CHIAA
An Interpreter

Submitted by:
Amol Shrestha
Binsan Khadka
Anup Chandra Poudyal

Kathmandu University
Dhulikhel, Nepal

Acknowledgement

We are greatly indebted to Fulbright Prof. Karen Lemmone for providing us with a problem to work with and her valuable suggestions and class lecture. This has been a great opportunity for the course Compiler Design that we learnt as a part of undergraduate course at Kathmandu University. We should also thank her for her time and valuable suggestions.

Of course, we would also like to acknowledge the Computer Department for giving us a chance to work for the project work. In particular Mr. Manish Pokharel.

Last but not the least, we are grateful to our class mates and all other friends who had helped us for the success of this project.

Abstract

Given a program written in high level language "Chiaa" as an input, the objective of this project is to compile the given program and output the equivalent code ready to assemble in a single resistor machine.

The Project has been divided into three sub division; the Scanner, Parser and Interpreter. An input program is initially scanned and list of tokens are extracted by the Scanner. Then those tokens are used to build an Abstract Syntax Tree (AST) with reference to the grammar (Rule) of the language. This is the job of Parser. This tree is traversed by an interpreter in a preorder manner to generate the assembly language code.

Table Of Content

1. Introduction

Given a program written in high level language "Chiaa" as an input, the objective of this project is to compile the given program and output the equivalent code ready to assemble in a single resistor machine.

2. The Scanner

For the first phase of this project, i.e. the Scanner, a popular Lexical Analyzer Generator software "FLEX" has been used.

Flex (Fast LEXical analyzer generator) is a tool for generating programs that perform pattern-matching on text. FLEX is a tool for generating scanners; programs which recognized lexical patterns in text. flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C Code, called rules, flex generates as output a C source files, lex.yy.c, which defines a routine yylex(). This file is compiled and linked with the -ll library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions.

2.1. The Tokens

Below are the List of Tokens used for Chiaa:

TYPE
"void", "int"

LOGICAL OPERATORS
"!", "||", "&&", "!=", "==", "<", ">", "<=", ">="

NUMERICAL OPERATORS
"+", "-", "*", "/", "%"

PUNCTUATION
"{", "}", "(", ")", ",", ";"

NAMES
Letter (Letter | Digit)*
where a Letter is either an uppercase of lowercase letter and Digit is one of the digits from 0-9.

INTEGERS
Sequences of 1 or more Digits.

2.2 The Lex file

For Scanner, list of tokens have to be identified. Below is the Listing of the Lex file that was used to find out tokens:

```
%{#include <stdio.h>
%}

NOT     [!]
PIPE    [|]
AND     [&]
EQUAL   [=]
LESS    [<]
GREATER [>]

PLUS    [+]
MINUS   [-]
STAR    [*]
SLASH   [/]
PERCENT [%]

MIDDLEOPEN  [{]
MIDDLECLOSE [}]
SMALLOPEN   [(]
SMALLCLOSE  [)]
COMMA       [,]
SEMICOLON   [;]

DIGIT [0-9]
LETTER [a-zA-Z]


%%

("void"|"int")  printf("<Type, \"%s\">",yytext);

({NOT}|{PIPE}{PIPE}|{AND}{AND}|{NOT}{EQUAL}|{EQUAL}{EQUAL}|{LESS}|{GREAT
ER}|{LESS}{EQUAL}|{GREATER}{EQUAL})   printf("<Logical operators,
\"%s\">",yytext);

({PLUS}|{MINUS}|{STAR}|{SLASH}|{PERCENT}) printf("<Numerical Operators,
\"%s\">", yytext);

({MIDDLEOPEN}|{MIDDLECLOSE}|{SMALLOPEN}|{SMALLCLOSE}|{COMMA}|{SEMICOLON}
) printf("<Punctuation, \"%s\">", yytext);

{DIGIT}+             printf("<Integers, \"%s\">", yytext);
{LETTER}({LETTER}|{DIGIT})*printf("<Names, \"%s\">", yytext);

%%
main ( argc, argv)
int argc;
```

```
char **argv;
{
++argv, --argc;
if (argc > 0)
      yyin = fopen(argv[0], "r");
else
      yyin = stdin;
yylex();
}
```

2.3. Inputs and Outputs

Input 1:

```
void input_a() {
 a = b3;
 xyz = a + b + c - p/q;
 a = xyz * (p +q);
 p = a - xyz - p;
}
```

Output 1:

```
<Type, "void"> <Names, "input">_<Names, "a"><Punctuation,
"("><Punctuation, ")"> <Punctuation, "{">
 <Names, "a"> = <Names, "b3"><Punctuation, ";">
 <Names, "xyz"> = <Names, "a"> <Numerical Operators, "+"> <Names, "b">
<Numerical Operators, "+"> <Names, "c"> <Numerical Operators, "-">
<Names, "p"><Numerical Operators, "/"><Names, "q"><Punctuation, ";">
 <Names, "a"> = <Names, "xyz"> <Numerical Operators, "*"> <Punctuation,
"("><Names, "p"> <Numerical Operators, "+"><Names, "q"><Punctuation,
")"><Punctuation, ";">
 <Names, "p"> = <Names, "a"> <Numerical Operators, "-"> <Names, "xyz">
<Numerical Operators, "-"> <Names, "p"><Punctuation, ";">
<Punctuation, "}">
```

Input 2:

```
void input_b() {
 if (i>j)
  i = i + i;
 else if (i<j)
  i = 1;
}
```

Output 2:

```
<Type, "void"> <Names, "input">_<Names, "b"><Punctuation,
"("><Punctuation, ")"> <Punctuation, "{">
 <Names, "if"> <Punctuation, "("><Names, "i"><Logical operators,
">"><Names, "j"><Punctuation, ")">
  <Names, "i"> = <Names, "i"> <Numerical Operators, "+"> <Names,
"i"><Punctuation, ";">
 <Names, "else"> <Names, "if"> <Punctuation, "("><Names, "i"><Logical
operators, "<"><Names, "j"><Punctuation, ")">
  <Names, "i"> = <Integers, "1"><Punctuation, ";">
<Punctuation, "}">
```

__Input 3:__

```
void input_c() {
 while (i<j && j >k) {
  k = k + 1;
   while ( i==j)
    i = i+2;
 }
}
```

__Output 3__:

```
<Type, "void"> <Names, "input">_<Names, "c"><Punctuation,
"("><Punctuation, ")"> <Punctuation, "{">
 <Names, "while"> <Punctuation, "("><Names, "i"><Logical operators,
"<"><Names,
"j"> <Logical operators, "&&"> <Names, "j"> <Logical operators,
">"><Names, "k"><Punctuation, ")"> <Punctuation, "{">
  <Names, "k"> = <Names, "k"> <Numerical Operators, "+"> <Integers,
"1"><Punctuation, ";">
   <Names, "while"> <Punctuation, "("> <Names, "i"><Logical operators,
"=="><Names, "j"><Punctuation, ")">
    <Names, "i"> = <Names, "i"><Numerical Operators, "+"><Integers,
"2"><Punctuation, ";">
 <Punctuation, "}"> <Punctuation, "}">
```

__Input 4:__

```
void input_c() {
 int a,b,c;
 a = b = c = 5;
  while ((a >=1 && c != 80) || !b) {
    a = b % 2;
    b = b - 1;
    c = c * 2;
}
```

__Output 4:__

```
<Type, "void"> <Names, "input">_<Names, "c"><Punctuation,
"("><Punctuation, ")"> <Punctuation, "{">
```

```
 <Type, "int"> <Names, "a"><Punctuation, ","><Names, "b"><Punctuation,
","><Names, "c"><Punctuation, ";">
 <Names, "a"> = <Names, "b"> = <Names, "c"> = <Integers,
"5"><Punctuation, ";">
 <Names, "while"> <Punctuation, "("><Punctuation, "("><Names, "a">
<Logical operators, ">="><Integers, "1"> <Logical operators, "&&">
<Names, "c"> <Logical operators, "!="> <Integers, "80"><Punctuation,
")"> <Logical operators, "||"> <Logical operators, "!"><Names,
"b"><Punctuation, ")"> <Punctuation, "{">
  <Names, "a"> = <Names, "b"> <Numerical Operators, "%"> <Integers,
"2"><Punctuation, ";">
  <Names, "b"> = <Names, "b"> <Numerical Operators, "-"> <Integers,
"1"><Punctuation, ";">
   <Names, "c"> = <Names, "c"> <Numerical Operators, "*"> <Integers,
"2"><Punctuation, ";">
  <Punctuation, "}">
```

3. The Parser

YACC (Yet Another Compiler-Compiler) has been used for this part of the project. Yacc reads the grammar specification and generates the LR(1) parser for it. Normally Yacc writes the parse tables and the driver routine to the file y.tab.c. For the Details of using Yacc: [2],[3],[4] and [5] were referred.

3.1 The Grammar

Below is EBNF Grammar that were provided for the Implementation of the Chiaa Grammar using yacc:

```
program -> method_declaration
method_declaration -> type name "(" ")" "{" statement_block "}"
type -> "void" | variable_type
variable_type -> "int"

statement_block -> { statement }
statement -> simple_statement  ";" | compound_statement | "{"  statement_block  "}"

simple_statement -> declarative_statement | assignment_statement
declarativestatement -> variable_type assignmentstatement
                    {"," assignmentstatement}
assignment_statement -> name [assignop expression]

expression -> or_expression
or_expression -> and_expression  { or and_expression }
and_expression -> relop_expression { and  relop_expression }
```

```
relop_expression -> ltgt_expression { relop ltgt_expression }
ltgt_expression -> addop_expression { ltgt addop_expression }
addop_expression -> mulop_expression { addop mulop_expression }
mulop_expression -> term { mulop term }
term -> not value | addop value | value
value -> name | number |   "("  expression  ")"

assignop -> "="
not -> "!"
or -> "||"
and -> "&&"
relop -> "!=" | "=="
ltgt -> ">" | "<" | ">=" | "<="
addop -> "+" | "-"
mulop -> "*" | "/" | "%"

compound_statement -> if_statement | loop_statement
if_statement -> "if" "(" expression ")" statement [ "else" statement ]
loop_statement -> while_statement | dowhile_statement | for_statement
while_statement -> "while" "(" expression ")" statement
do_whilestatement -> "do" statement "while" "(" expression ")" ";"
forstatement -> "for" "(" [for_expression] ";" [expression] ";"
                [for_expression] ")" statement
for_expression -> declarative_statement | assignment_statement
                 { "," assignment_statement }

name -> letter { letter | digit | "_" }
number -> digit { digit }
letter -> [a-zA-Z]
digit -> [0-9]
```

3.2 Modified Lex File

For the parser to work, it uses Tokens generated by the lexer. So, the lexer has to be made in such a way so that it can be synchronized with parser. Below is the list of Modified Lex File that returns the Token for the Parser with the value of token stored in corresponding member of structure yylval.

```
%{#include "y.tab.h"
    #include <string.h>
%}

DIGIT [0-9]
LETTER [a-zA-Z]
```

```
%%
"void"                 {yylval.cVal = "VOID"; return(VOID); }
"int"                  {yylval.cVal = "INT"; return(INT); }

"if"                   {yylval.cVal = "IF"; return(IF); }
"else"                 {yylval.cVal = "ELSE"; return(ELSE); }
"while"                {yylval.cVal = "WHILE"; return(WHILE); }
"do"                   {yylval.cVal = "DO"; return(DO); }
"for"                  {yylval.cVal = "FOR"; return(FOR); }

"="                    {yylval.cVal = "="; return(ASSIGNOP); }
"+"                    {yylval.cVal = "+"; return(ADDOP); }
"-"                    {yylval.cVal = "-"; return(ADDOP); }

"*"                    {yylval.cVal = "*"; return(MULOP); }
"/"                    {yylval.cVal = "/"; return(MULOP); }
"%"                    {yylval.cVal = "%"; return(MULOP); }

">"                    {yylval.cVal = ">"; return(LTGT); }
"<"                    {yylval.cVal = "<"; return(LTGT); }
"<="                   {yylval.cVal = "<="; return(LTGT); }
">="                   {yylval.cVal = ">="; return(LTGT); }

"!="                   {yylval.cVal = "!="; return(RELOP); }
"=="                   {yylval.cVal = "=="; return(RELOP); }


"&&"                   {yylval.cVal = "AND"; return(AND); }
"||"                   {yylval.cVal = "OR"; return(OR); }
"!"                    {yylval.cVal = "NOT"; return(NOT); }

"("                    {return(LPAREN); }
")"                    {return(RPAREN); }
"{"                    {return(OBRACE); }
"}"                    {return(EBRACE); }

","                    {return(COMMA); }
";"                    {return(SEMICOLON); }

{DIGIT}+                                {yylval.iVal = atoi(yytext); return(NUMBER); }
{LETTER}({LETTER}|{DIGIT}|"_")*
                                        {yylval.cVal = (char*)malloc(strlen(yytext)+1);
                                        strcpy(yylval.cVal, yytext); return(NAME); }

[ \t]                          {}
[\n]                           {}

%%
```

3.3 The Yacc File

This part creates an Abstract Syntax Tree (AST). and prints out in a way that while traversing the tree, each time it goes down, a bracket "(" is printed, and ")" will be printed after complete traverse of the tree.

For this part many pages were referred to. To solve the Shift/Reduce Error for the rule of if_statement in the Grammar, Page 35-36 of [3] was referred, where it tries to give higher precedence to IF-ELSE than a simple IF statement.

The Yacc file that generates AST and prints out is listed below:

```
%{
#include <stdio.h>
  typedef enum {NAME, NUMBER} myType;
  typedef union{
        char *c;
        int i;
  }types;

  typedef struct tnode{
        myType type;
        types info;
        struct tnode *left;
        struct tnode *right;
  } typenode, *typeptr;
  typeptr new_char_node(char *info);
  typeptr new_int_node(int info);
  typeptr add_char_node(char *info, typeptr t1, typeptr t2);
  typeptr add_child_node(typeptr t1, typeptr t2, typeptr t3);
%}

%union{
        typeptr t;
        char *cVal;
        int iVal;
}

%type <t> program method_declaration type variable_type statement_block
%type <t> statement simple_statement assignment_statement declarative_statement
declarative_statement2 compound_statement if_statement loop_statement
while_statement dowhile_statement for_statement
%type <t> expr or_expr and_expr relop_expr ltgt_expr addop_expr mulop_expr  term
value for_expr for_expr2

%type <cVal>  VOID INT NAME ASSIGNOP OR AND NOT RELOP LTGT ADDOP MULOP
%type <cVal> IF FOR WHILE DO
%type <iVal> NUMBER

%token VOID INT IF ELSE WHILE DO FOR ASSIGNOP ADDOP MULOP LTGT RELOP AND OR NOT
LPAREN RPAREN OBRACE EBRACE COMMA SEMICOLON NUMBER NAME

%nonassoc IFX
%nonassoc ELSE

%%
```

```
program         :       method_declaration
                                          {print_tree($1);printf("\n");}
                ;

method_declaration :    type NAME LPAREN RPAREN OBRACE statement_block EBRACE
                              {
                               typeptr temp = (typeptr)
                               malloc(sizeof(typenode));
                               temp = new_char_node($2);
                               $1 = add_child_node($1,temp,NULL);
                              $$ = add_char_node(".",$1, $6);
                              }
                ;

type            :       VOID              {$$= new_char_node($1); }
                |       variable_type     {$$=$1; }
                ;

variable_type   :       INT              {$$= new_char_node($1); }
                ;

statement_block :       /*empty*/                      {$$=NULL;}
                |       statement statement_block
                               {
                               typeptr temp = (typeptr)
                        malloc(sizeof(typenode));
                              temp = add_char_node(".",$1,$2);
                              $$= temp;
                              }
                ;

statement       :       simple_statement SEMICOLON      {$$ = $1; }
                |       compound_statement              {$$ = $1; }
                |       OBRACE statement_block EBRACE   {$$ =  $2; }
                ;

simple_statement :      assignment_statement            {$$= $1; }
                |       declarative_statement           {$$= $1; }
                ;

declarative_statement : variable_type assignment_statement
declarative_statement2
                              {
                               typeptr temp = (typeptr)
                               malloc(sizeof(typenode));
                              temp = add_char_node($1 ->info.c,$2,$3);
                              $$ = temp;
                              }
                ;

declarative_statement2 : /*empty*/                      {$$ = NULL; }
                |       COMMA assignment_statement declarative_statement2
                              {
                              typeptr temp = (typeptr)
malloc(sizeof(typenode));
                              temp = add_char_node(".",$2,$3);
                              $$ = temp;
                              }
                ;

assignment_statement :  NAME                    {$$ = new_char_node($1); }
                |       NAME ASSIGNOP expr
                             {
                             typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                             typeptr temp2 = (typeptr)
malloc(sizeof(typenode));
                             temp1 = new_char_node($2);
                             temp2 = new_cha r_node($1);
                             temp1 = add_child_node(temp1,temp2,$3);
                             $$ = temp1;
                             }
                ;

expr            :       or_expr                 {$$ =$1;}
                ;

or_expr         : or_expr OR and_expr
                              {
                              typeptr temp = (typeptr)
malloc(sizeof(typenode));
                              temp = add_char_node($2,$1,$3);
                              $$=temp;
                              }
                |       and_expr                  {$$ =$1; }
                ;

and_expr      :   and_expr AND relop_expr
                              {
                              typeptr temp = (typeptr)
malloc(sizeof(typenode));
                              temp = add_char_node($2,$1,$3);
                              $$=temp;
                              }
              |   relop_expr                {$$ =$1; }
              ;

relop_expr    :   relop_expr RELOP ltgt_expr
                              {
                              typeptr temp = (typept r)
malloc(sizeof(typenode));
                              temp = add_char_node($2,$1,$3);
                              $$=temp;
                              }
              |   ltgt_expr                 {$$ =$1; }
              ;

ltgt_expr     :   ltgt_expr LTGT addop_expr
                             {
                             typeptr temp = (typeptr) malloc(sizeof(typenode));
                             temp = add_char_node($2,$1,$3);
                             $$=temp;
                             }
              |   addop_expr                {$$ =$1; }
              ;
```

```
addop_expr    :   addop_expr ADDOP mulop_expr
                          {
                          typeptr temp = (typeptr)
                      malloc(sizeof(typenode));
                          temp = add_char_node($2,$1,$3);
                          $$=temp;
                          }
              |   mulop_expr               {$$= $1; }
              ;

mulop_expr    :   mulop_expr MULOP term
                          {
                          typeptr temp = (typeptr)
malloc(sizeof(typenode));
                          t emp = add_char_node($2,$1,$3);
                          $$=temp;
                          }
              |   term                    {$$= $1; }
              ;

term          :   NOT value
                          {
                          typeptr temp = (typeptr)
malloc(sizeof(typenode));
                          temp = add_char_node($1,$2,NULL);
                          $$=temp;
                          }
              |   ADDOP value
                          {
                          typeptr temp = (t ypeptr)
malloc(sizeof(typenode));
                          temp = add_char_node($1,$2,NULL);
                          $$=temp;
                          }
              |   value                   {$$= $1; }
              ;

value         :   NAME
                          {
                          typeptr temp = (typeptr)
malloc(sizeof(typenode));
                          temp = new_char_node($1);
                          $$ = temp;
                          }

              |   NUMBER
                          {
                          typeptr temp = (typeptr)
malloc(sizeof(typenode));
                          temp = new_int_node($1);
                          $$ = temp;
                          }

              |   LPAREN expr RPAREN             { $$ = $2; }
              ;

compound_statement :    if_statement           { $$ = $1; }
               |        loop_statement         { $$ = $1; }
               ;
```

```
if_statement    :       IF LPAREN expr RPAREN statemen t %prec IFX
                              {
                              typeptr temp = (typeptr)
malloc(sizeof(typenode));
                              temp = new_char_node($1);
                              temp = add_child_node(temp, $3, $5);
                              $$=temp;
                              }
                |       IF LPAREN expr RPAREN statement ELSE statement
                             {
                             typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                             typeptr tem p2 = (typeptr)
malloc(sizeof(typenode));
                             temp1 = add_char_node(".",$5,$7);
                             temp2 = add_char_node($1,$3,temp1);
                             $$=temp2;
                             }
                ;

loop_statement  :       while_statement                 {$$=$1; }
                |       dowhile_statement               {$$=$1; }
                |       for_statement                   {$$=$1; }
                ;

while_statement :       WHILE LPAREN  expr RPAREN statement
                             {
                             typeptr temp = (typeptr) malloc(sizeof(typenode));
                             temp = add_char_node($1, $3, $5);
                             $$=temp;
                             }
                ;

dowhile_statement :     DO statement WHILE LPAREN expr RPAREN SEMICOLON
                             {
                             typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                             typeptr temp2 =  (typeptr)
malloc(sizeof(typenode));
                             temp1 = add_char_node($3,$5,NULL);
                             temp2 = add_char_node($1,$2,temp1);
                             $$=temp2;
                             }
                ;

for_statement   :       FOR LPAREN SEMICOLON SEMICOLON RPAREN statement
                             {
                             typeptr temp = (typeptr) malloc(sizeof(typenode));
                             temp = add_char_node(".",NULL,NULL);
                             $$ = add_char_node($1,temp,$6);
                             }
         |       FOR LPAREN for_expr SEMICOLON SEMICOLON RPAREN statement
                             {
                             typeptr temp = (typeptr) malloc(sizeof (typenode));
                             temp = add_char_node(".",$3,NULL);
                             $$= add_char_node($1,temp,$7);
                             }
         |       FOR LPAREN SEMICOLON for_expr SEMICOLON RPAREN statement
```

```
                              {
                              typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                              typeptr temp2 = (typeptr)
malloc(sizeof(typenode));
                              temp1 = add_char_node(".",$4,NULL);
                              temp2 = add_char_node(".",NULL,temp1);
                              $$ = add_char_node($1,temp2,$7);
                              }
          |       FOR LPAREN SEMICOLON SEMICOLON for_expr RPAREN statement
                              {
                              typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                              typeptr temp2 = (typeptr)
malloc(sizeof(typenode));
                              temp1 = add_char_node(".",NULL,$5);
                              temp2 = add_char_node(".",NULL,temp1);
                              $$= add_char_node($1,temp2,$7);
                              }
          |  FOR LPAREN for_expr SEMICOLON for_expr SEMICOLON RPAREN statement
                              {
                              typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                              typeptr temp2 = (typeptr)
malloc(sizeof(typenode));
                              temp1 = add_char_node(".",$5,NULL);
                              temp2 = add_char_node(".",$3,temp1);
                              $$= add_char_node($1,temp2,$8);
                              }
          |  FOR LPAREN for_expr SEMICOLON SEMICOLON for_expr RPAREN statement
                              {
                              typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                              typeptr temp2 = (typeptr)
malloc(sizeof(typenode));
                              temp1 = add_char_node(".",NULL,$6);
                              temp2 = add_char_node(".",$3,temp1);
                              $$= add_char_node($1,temp2,$8);
                              }
          |  FOR LPAREN SEMICOLON for_expr SEMICOLON for_expr RPAREN statement
                              {
                              typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                              typeptr temp2 = (typeptr)
malloc(sizeof(typenode));
                              temp1 = add_char_node(".",$4,$6);
                              temp2 = add_char_node(".",NULL,temp1);
                              $$= add_char_node($1,temp2,$8);
                              }
     |  FOR LPAREN for_expr SEMICOLON for_expr SEMICOLON for_expr RPAREN
statement
                              {
                              typeptr temp1 = (typeptr)
malloc(sizeof(typenode));
                              typeptr temp2 = (typeptr)
malloc(sizeof(typenode));
                              temp1 = add_char_node(".",$5,$7);
                              temp2 = add_char_node(".",$3,temp1);
                              $$= add_char_node($1,temp2,$9);
```

```
                                    }
          ;

for_expr    :       declarative_statement           {$$= $1; }
            |       assignment_statement for_expr2
                                    {
                                    $$= add_char_node(".",$1,$2);
                                    }
            ;

for_expr2   :    /*empty*/                        {$$=NULL; }
            |    COMMA assignment_statement for_expr2
                                    {
                                    $$= add_char_node(".",$2,$3);
                                    }
            ;


%%
#include <stdio.h>
#include <stdlib.h>
#include "lex.yy.c"
#include "part3.c"

main(){
       yyparse();
}

yyerror(char *s) {
        fprintf(stderr, "%s\n",s);
}
```

3.4 Required Functions

Some of the function calls made from within the above listed yacc file was stored in a separate file. The source for those functions are listed below:

```
typeptr add_child_node(typeptr t1, typeptr t2, typeptr t3) {
        t1->left = t2;
        t1->right = t3;
        return t1;
}

typeptr new_char_node(char *info) {
        typeptr t;
        t = (typeptr) malloc(sizeof(typenode));
        t->info.c = (char*)malloc(strlen(info)+1);
        strcpy(t->info.c,info);
        t->type = NAME;
        t->left = NULL;
        t->right = NULL;
        return t;
}
```

```
typeptr new_int_node(int info){
        typeptr t;
        t = (typeptr) malloc(sizeof(typenode));
        t->info.i = (int) malloc (sizeof(int));
        t->info.i = info;
        t->type = NUMBER;
        t->left = NULL;
        t->right = NULL;
        return t;
}

typeptr add_char_node(char *info, typeptr t1, typeptr t2) {
        typeptr t;
        t = (typeptr)malloc(sizeof(typeptr));
        t->info.c = (char*)malloc(strlen(info)+1);
        strcpy(t->info.c,info);
        t->type = NAME;
        t->left = t1;
        t->right = t2;
        return t;
}

void print_tree (typeptr t) {
        if (t != NULL) {
                if (t->type == NAME)
                        printf("%s ", t->info.c);
                if (t->type == NUMBER)
                        printf("%d ", t->info.i);
                if (t->left != NULL || t->right != NULL) {
                        printf("\n( ");
                        print_tree(t->left);
                        print_tree(t->right);
                        printf(")");
                }
        }
        return ;
}
```

3.5 Inputs and outputs

The Following Steps were used for input and output :

```
 $ lex part3.lex
 $ yacc -d part3.yacc
 $ cc y.tab.c -ll
 $ ./a.out <input1
 $ ./a.out <input2
 $ ./a.out <input3
 $ ./a.out <input4
 $ ./a.out <input5
```

Inputs (input1, input2, input3, and input4) for this part is the same as used for the scanner. Listing of Input 5 is given below.
The "**.**" represents node with not a valid information, i.e. a Dummy Node. Each time a new Open brace is printed, it denotes the elements following Brace is a child of the preceeding element until a Close Brace is found.

Output 1:

```
.
( VOID
( input_a ).
( =
( a b3 ).
( =
( xyz -
( +
( +
( a b )c )/
( p q ))).
( =
( a *
( xyz +
( p q ))).
( =
( p -
( -
( a xyz )p )))))))
```

Output 2:

```
.
( VOID
( input_b ).
( IF
( >
( i j ).
( =
( i +
( i i ))IF
( <
( i j )=
( i 1 ))))))
```

Output 3:

```
.
( VOID
( input_c ).
( WHILE
( AND
( <
( i j )>
( j k )).
( =
( k +
( k 1 )).
( WHILE
( ==
( i j )=
( i +
( i 2 ))))))))
```

Output 4:

Syntax error

[Note : The Output of Input 4 give result as Syntax Error. Let's Look at Input 4 again:

1. void input_c() {
2.   int a,b,c;
3.   a = b = c = 5;
4.   while ((a >=1 && c != 80) || !b) {
5.   a = b % 2;
6.   b = b - 1;
7.   c = c * 2;
8.   }

In Line 3 There multiple assignment expression which is not defined in grammar. The given grammar is:

    assignment_statement -> name [assignop expression]

so, there is either 0 or 1 "assignop" followed by "expression"

Thus, this result in a syntax error

]

Input 5:

```
void input_e() {
        int a,b,c;

        for (;;) {
                c = a;
                a = b;
        }

        for (i=5;;j=j+1,j=j-1) {
                i=a+2;
        }
}
```

Output 5:

```
.
( VOID
( input_e ).
( INT
( a .
( b .
( c ))).
( FOR
( . .
( =
( c a ).
( =
( a b )))).
( FOR
( .
( .
( =
( i 5 )).
( .
( =
( j +
( j 1 )).
( =
( j -
( j 1 )))))).
( =
( i +
( a 2 )))))))
```

4. Interpreter

The Final Goal of this project is to convert the program written in language chiaa to an equivalent code ready to Assemble. After the generation of Abstract Syntax Tree, the task of interpreter is just a Tree Walk in a pre-order manner. So, a new procedure "generate()" is added which traverse the tree and print out equivalent assembly code. The Assembly code is generated assuming a single Resistor machine with sufficient memory to store intermediate variables. The Final source of the project are listed on section 6.

5. Conclusion & Further Recommendation

After completion of this project we have came up with an interpreter that can translate high level language program into an equivalent assembly language program.

There are still some more works that can be done. Though, the generated code perform equivalent translation, there are some unwanted code that can be optimized to make our compiler run faster than it does. Also, the Grammar does not deals with other data types like real, string, array and other enumerated data types.

6. Appendix: Chiaa

The Final Version of the project has three files which are listed below.

6.1. Chiaa.lex

```
/*
 *
 * File : chiaa.lex
 * Last Modified: 14 December 2002
 *
 *
 */


%{#include "y.tab.h"
  #include <string.h>

  int yylineno=1;   // used for line count.
%}

DIGIT [0-9]
LETTER [a-zA-Z]

%%
"void"                                  {yylval.cVal = "VOID"; return(VOID); }
"int"                                   {yylval.cVal = "INT"; return(INT); }

"if"                                    {yylval.cVal = "IF"; return(IF); }
"else"                                  {yylval.cVal = "ELSE"; return(ELSE); }
"while"                                 {yylval.cVal = "WHILE"; return(WHILE); }
"do"                                    {yylval.cVal = "DO"; return(DO); }
"for"                                   {yylval.cVal = "FOR"; return(FOR); }

"="                                     {yylval.cVal = "="; return(ASSIGNOP); }
"+"                                     {yylval.cVal = "+"; return(ADDOP); }
"-"                                     {yylval.cVal = "-"; return(ADDOP); }

"*"                                     {yylval.cVal = "*"; return(MULOP); }
"/"                                     {yylval.cVal = "/"; return(MULOP); }
"%"                                     {yylval.cVal = "%"; return(MULOP); }
```

```
">"                                     {yylval.cVal = ">"; return(LTGT); }
"<"                                     {yylval.cVal = "<"; return(LTGT); }
"<="                                    {yylval.cVal = "<="; return(LTGT); }
">="                                    {yylval.cVal = ">="; return(LTGT); }

"!="                                    {yylval.cVal = "!="; return(RELOP); }
"=="                                    {yylval.cVal = "=="; return(RELOP); }


"&&"                                    {yylval.cVal = "AND"; return(AND); }
"||"                                    {yylval.cVal = "OR"; return(OR); }
"!"                                     {yylval.cVal = "NOT"; return(NOT); }

"("                                     {return(LPAREN); }
")"                                     {return(RPAREN); }
"{"                                     {return(OBRACE); }
"}"                                     {return(EBRACE); }

","                                     {return(COMMA); }
";"                                     {return(SEMICOLON); }

{DIGIT}+                                {yylval.iVal = atoi(yytext); return(NUMBER); }
{LETTER}({LETTER}|{DIGIT}|"_")*         {yylval.cVal = (char*)malloc(strlen(yytext)+1); strcpy(yylval.cVal, yytext);
return(NAME); }

[ \t]                                   { /*ignore blank spaces*/ }
[\n]                                    {yylineno = yylineno +1;}

%%
```

6.2. Chiaa.yacc

```
/*
 *
 * File : chiaa.yacc
 * Last Modified : 14 December 2002
 *
 *
*/

%{
#include <stdio.h>
  typedef enum {STR, NUMBER, DUMMY, LOGOPER, NUMOPER, ASSIGNOPER, TYPE, RESWORD} myType;
  typedef union{
        char *c;
        int i;
  }types;

  typedef struct tnode{
        myType type;
        types info;
        struct tnode *left;
        struct tnode *right;
  } typenode, *typeptr;

  static int count = 1;
  static int label = 1;


  typeptr empty_node();
  typeptr new_node(char *info, myType t);
  typeptr new_int_node (int info);
  typeptr new_dummy_node();
  typeptr join_child_node(typeptr t1, typeptr t2, typeptr t3);

  void print_tree(typeptr t);
  void generate(typeptr t);

%}

%union{
        typeptr tNode;
        char *cVal;
        int iVal;
}

%type <tNode> program method_declaration type variable_type statement_block
%type <tNode> statement simple_statement assignment_statement declarative_statement declarative_statement2
%type <tNode> expr or_expr and_expr relop_expr ltgt_expr addop_expr mulop_expr term value
%type <tNode> compound_statement if_statement loop_statement while_statement dowhile_statement for_statement
%type <tNode> for_expr for_expr2

%token <cVal> VOID INT NAME ASSIGNOP OR AND NOT RELOP LTGT ADDOP MULOP
%token <cVal> IF FOR WHILE DO
%token <iVal> NUMBER

%token LPAREN RPAREN OBRACE EBRACE COMMA SEMICOLON

%nonassoc IFX
%nonassoc ELSE
```

```
%%

program         :       method_declaration               {
                                                         printf("Abstract Syntax Tree:\n(");
                                                         print_tree($1);
                                                         printf(")\n");
                                                         printf("\nTotal No Of Lines:%d\n\n",yylineno);
                                                         printf("Code Generated:\n");
                                                         generate($1);
                                                         }
                ;

method_declaration :    type NAME LPAREN RPAREN OBRACE statement_block EBRACE
                                                         {
                                                         $1 = join_child_node($1,new_node($2,STR),NULL);
                                                         $$ = join_child_node(new_dummy_node(),$1, $6);
                                                         }
                ;


type            :       VOID                             {$$= new_node($1,TYPE); }
                |       variable_type                    {$$=$1; }
                ;

variable_type   :       INT                              {$$= new_node($1,TYPE); }
                ;

statement_block :       /*empty*/                        {$$ = NULL;}
                |       statement statement_block        {$$ = join_child_node(new_dummy_node(),$1,$2);}
                ;

statement       :       simple_statement SEMICOLON       {$$ = $1; }
                |       compound_statement               {$$ = $1; }
                |       OBRACE statement_block EBRACE    {$$ = $2; }
                ;

simple_statement :      assignment_statement             {$$= $1; }
                |       declarative_statement            {$$= $1; }
                ;

declarative_statement : variable_type assignment_statement declarative_statement2
                                                         {$$ = join_child_node($1,$2,$3); }
                ;

declarative_statement2 : /*empty*/                       {$$ = NULL; }
                |       COMMA assignment_statement declarative_statement2
```

```
                                                        {$$ = join_child_node(new_dummy_node(),$2,$3); }
                ;


assignment_statement :  NAME                            {$$ = new_node($1,STR); }
                |       NAME ASSIGNOP expr              {$$ =
join_child_node(new_node($2,ASSIGNOPER),new_node($1,STR),$3); }
                ;

expr            :       or_expr                         {$$ =$1; }
                ;

or_expr         :   or_expr OR and_expr             {$$ = join_child_node(new_node($2,LOGOPER),$1,$3);}
                |       and_expr                        {$$ =$1; }
               ;

and_expr        :       and_expr AND relop_expr         {$$ = join_child_node(new_node($2,LOGOPER),$1,$3);}
                |       relop_expr                      {$$ =$1; }
                ;

relop_expr      :       relop_expr RELOP ltgt_expr      {$$ = join_child_node(new_node($2,LOGOPER),$1,$3);}
                |       ltgt_expr                       {$$ =$1; }
                ;

ltgt_expr       :       ltgt_expr LTGT addop_expr       {$$ = join_child_node(new_node($2,LOGOPER),$1,$3);}
                |       addop_expr                      {$$ =$1; }
                ;

addop_expr      :       addop_expr ADDOP mulop_expr     {$$ = join_child_node(new_node($2,NUMOPER),$1,$3);}
                |       mulop_expr                      {$$= $1; }
                ;

mulop_expr      :       mulop_expr MULOP term           {$$ = join_child_node(new_node($2,NUMOPER),$1,$3);}
                |       term                            {$$= $1; }
                ;

term           :    NOT value                       {$$ = join_child_node(new_node($1,LOGOPER),$2,NULL);}
                |       ADDOP value                     {$$ = join_child_node(new_node($1,NUMOPER),$2,NULL);}
                |       value                           {$$= $1; }
               ;

value          :    NAME                            {$$ = new_node($1,STR);}

                |       NUMBER                          {$$ = new_int_node($1);}

                |       LPAREN expr RPAREN              {$$ = $2; }
                ;
```

```
compound_statement :    if_statement                     { $$ = $1; }
                |       loop_statement                   { $$ = $1; }
                ;

if_statement    :       IF LPAREN expr RPAREN statement %prec IFX
                                                         {$$=join_child_node(new_node($1,RESWORD), $3, $5);}
                |       IF LPAREN expr RPAREN statement ELSE statement
                                                         {
                                                         typeptr temp = join_child_node(new_dummy_node(),$5,$7);
                                                         $$= join_child_node(new_node($1,RESWORD),$3,temp);
                                                         }
                ;

loop_statement  :       while_statement                  {$$=$1; }
                |       dowhile_statement                {$$=$1; }
                |       for_statement                    {$$=$1; }
                ;

while_statement :       WHILE LPAREN expr RPAREN statement
                                                         {$$=join_child_node(new_node($1,RESWORD),$3,$5);}
                ;

dowhile_statement :     DO statement WHILE LPAREN expr RPAREN SEMICOLON
                                                         {
                                                         //typeptr temp = join_child_node(new_node($3,RESWORD),$5,NULL);
                                                         $$=join_child_node(new_node($1,RESWORD),$5,$2);
                                                         }
                ;

for_statement   :       FOR LPAREN SEMICOLON SEMICOLON RPAREN statement
                                                         {$$ =
join_child_node(new_node($1,RESWORD),new_dummy_node(),$6); }
                |       FOR LPAREN for_expr SEMICOLON SEMICOLON RPAREN statement
                                                         {
                                                         typeptr temp = join_child_node(new_dummy_node(),$3,NULL);
                                                         $$ = join_child_node(new_node($1,RESWORD),temp,$7);
                                                         }
                |       FOR LPAREN SEMICOLON expr SEMICOLON RPAREN statement
                                                         {
                                                         typeptr temp1 = join_child_node(new_dummy_node(),$4,NULL);
                                                         typeptr temp2 = join_child_node(new_dummy_node(),NULL,temp1);
                                                         $$ = join_child_node(new_node($1,RESWORD),temp2,$7);
                                                         }
                |       FOR LPAREN SEMICOLON SEMICOLON for_expr RPAREN statement
                                                         {
                                                         typeptr temp1 = join_child_node(new_dummy_node(),NULL,$5);
```

```
                                                typeptr temp2 = join_child_node(new_dummy_node(),NULL,temp1);
                                                $$= join_child_node(new_node($1,RESWORD),temp2,$7);
                                                }
                |       FOR LPAREN for_expr SEMICOLON expr SEMICOLON RPAREN statement
                                                {
                                                typeptr temp1 = join_child_node(new_dummy_node(),$5,NULL);
                                                typeptr temp2 = join_child_node(new_dummy_node(),$3,temp1);
                                                $$= join_child_node(new_node($1,RESWORD),temp2,$8);
                                                }
                |       FOR LPAREN for_expr SEMICOLON SEMICOLON for_expr RPAREN statement
                                                {
                                                typeptr temp1 = join_child_node(new_dummy_node(),NULL,$6);
                                                typeptr temp2 = join_child_node(new_dummy_node(),$3,temp1);
                                                $$= join_child_node(new_node($1,RESWORD),temp2,$8);
                                                }
                |       FOR LPAREN SEMICOLON expr SEMICOLON for_expr RPAREN statement
                                                {
                                                typeptr temp1 = join_child_node(new_dummy_node(),$4,$6);
                                                typeptr temp2 = join_child_node(new_dummy_node(),NULL,temp1);
                                                $$= join_child_node(new_node($1,RESWORD),temp2,$8);
                                                }
                |       FOR LPAREN for_expr SEMICOLON expr SEMICOLON for_expr RPAREN statement
                                                {
                                                typeptr temp1 = join_child_node(new_dummy_node(),$5,$7);
                                                typeptr temp2 = join_child_node(new_dummy_node(),$3,temp1);
                                                $$= join_child_node(new_node($1,RESWORD),temp2,$9);
                                                }
                ;

for_expr        :       declarative_statement           {$$= $1; }
                |       assignment_statement for_expr2
                                                {$$= join_child_node(new_dummy_node(),$1,$2);}
                ;

for_expr2       :       /*empty*/                       {$$=NULL; }
                |       COMMA assignment_statement for_expr2
                                                {$$= join_child_node(new_dummy_node(),$2,$3);}
                ;


%%
#include <stdio.h>
#include <stdlib.h>
#include "lex.yy.c"
#include "chiaa.c"
```

```
main(){
       yyparse();
}

yyerror(char *s) {
        fprintf(stderr, "Line: %d ;%s\n",yylineno,s);
}
```

6.3. Chiaa.c

```
/*
 * File : chiaa.c
 * Last Modified: 14 December 2002
 *
 *
 */


typeptr empty_node (){
// returns an empty node with its left and right pointers as NULL

        typeptr t1 = (typeptr) malloc(sizeof(typenode));
        t1->left = NULL;
        t1->right = NULL;

        return t1;
}


typeptr new_node(char *info, myType t){
//returns a new node of type t

        typeptr t1 = empty_node();

        t1->info.c = (char*)malloc(strlen(info)+1);
        strcpy(t1->info.c,info);
        t1->type = t;

        return t1;
}

typeptr new_int_node(int info) {
//returns a new node of type NUMBER
```

```
        typeptr t1 = empty_node();

        t1->info.i = info;
        t1->type = NUMBER;

        return t1;
}

typeptr new_dummy_node() {
//returns a new dummy node with "." filled in its info field

        typeptr t1= empty_node();

        t1->type = DUMMY;
        t1->info.c = ".";
}

typeptr join_child_node(typeptr t1, typeptr t2, typeptr t3) {
//return pointer to the root node that points the left subtree t2 and right subtree t3

        t1->left = t2;
        t1->right = t3;
        return t1;
}


void print_tree (typeptr t) {
//prints the abstract syntax tree formed

        if (t != NULL) {
                if ((t->type == STR) || (t->type == NUMOPER) || (t->type == TYPE) || (t->type == LOGOPER)
                     || (t->type == ASSIGNOPER) || (t->type == DUMMY) || (t->type == RESWORD))
                        printf("%s ", t->info.c);
                if (t->type == NUMBER)
                        printf("%d ", t->info.i);
                if (t->left != NULL || t->right != NULL) {
                        printf("\n( ");
                        print_tree(t->left);
                        print_tree(t->right);
                        printf(")");
                }
        }
        return ;
}

int isLeaf(typeptr t){
//return 1 if the node is leaf node, otherwise 0
```

```
        if (t->left==NULL && t->right==NULL)
                return 1;
        else
                return 0;
}

void generate(typeptr t){
//generate equivalent interpreted code from the abstract syntax tree

        int leftCount,rightCount;
        int leftlabel, rightlabel, slabel,scount;

        if (t == NULL) return;
        switch(t->type) {
                case DUMMY:
                        generate(t->left);
                        generate(t->right);
                        break;
                case TYPE:
                        //printf("Procedure %s: \n",t->left->info.c);
                        if (t->left->type == ASSIGNOPER)
                                generate(t->left);
                        generate(t->right);
                        break;
                case ASSIGNOPER:
                        generate(t->right);
                        printf("STORE R1, %s\n",t->left->info.c);
                        break;
                case STR:
                        if (isLeaf(t))
                                printf("LOAD %s, R1\n",t->info.c);
                        break;
                case NUMBER:
                        if (isLeaf(t))
                                printf("LOAD #%d, R1\n",t->info.i);
                        break;
                case LOGOPER:
                        if (strcmp(t->info.c,"<") == 0) {
                                scount = count++;
                                generate(t->left);
                                printf("STORE R1, temp%d\n",scount);
                                generate(t->right);
                                printf("L temp%d, R1\n",scount);
                        }

                        if (strcmp(t->info.c,"<=") == 0) {
```

```
        scount = count++;
        generate(t->left);
        printf("STORE R1, temp%d\n",scount);
        generate(t->right);
        printf("LE temp%d, R1\n",scount);
}

if (strcmp(t->info.c,">") == 0) {
        scount = count++;
        generate(t->left);
        printf("STORE R1, temp%d\n",scount);
        generate(t->right);
        printf("G temp%d, R1\n",scount);
}

if (strcmp(t->info.c,">=") == 0) {
        scount = count++;
        generate(t->left);
        printf("STORE R1, temp%d\n",scount);
        generate(t->right);
        printf("GE temp%d, R1\n",scount);
}

if (strcmp(t->info.c,"!=") == 0) {
        scount = count++;
        generate(t->left);
        printf("STORE R1, temp%d\n",scount);
        generate(t->right);
        printf("NE temp%d, R1\n",scount);
}

if (strcmp(t->info.c,"==") == 0) {
        scount = count++;
        generate(t->left);
        printf("STORE R1, temp%d\n",scount);
        generate(t->right);
        printf("CMP temp%d, R1\n",scount);
}

if (strcmp(t->info.c,"AND") == 0) {
        scount = count++;
        generate(t->left);
        printf("STORE R1, temp%d\n",scount);
        generate(t->right);
        printf("AND R1, temp%d\n",scount);
}
```

```
        if (strcmp(t->info.c,"OR") == 0) {
                scount = count++;
                generate(t->left);
                printf("STORE R1, temp%d\n",scount);
                generate(t->right);
                printf("OR R1, temp%d\n",scount);
        }

        if (strcmp(t->info.c,"NOT") == 0) {
                scount = count++;
                generate(t->left);
                printf("NOT R1\n");
        }


        break;
case RESWORD:
        slabel = label;

        if (strcmp(t->info.c,"IF") == 0) {
                if (t->right->type == DUMMY){
                        if (t->right->left != NULL){
                                leftlabel = label++;
                                slabel = leftlabel;
                        }
                        if (t->right->right != NULL)
                                rightlabel = label++;
                }

                generate(t->left);
                printf("IFFALSE GOTO Label%d\n",slabel);
                if (t->right->type != DUMMY){
                        generate(t->right);
                        printf("Label%d:\n",slabel);
                }
                else{
                        generate(t->right->left);
                        printf("GOTO Label%d\n",rightlabel);
                        printf("Label%d:\n",leftlabel);
                        generate(t->right->right);
                        printf("Label%d:\n",rightlabel);
                }
        }

        if (strcmp(t->info.c,"WHILE") == 0) {
                leftlabel = label++;
                rightlabel = label++;
                printf("LABEL%d:\n",leftlabel);
                generate(t->left);
                printf("IFFALSE GOTO Label%d\n",rightlabel);
                generate(t->right);
                printf("GOTO Label%d\n",leftlabel);
                printf("LABEL%d:\n",rightlabel);
        }

        if (strcmp(t->info.c,"DO") == 0) {
                slabel = label++;
                printf("LABEL%d:\n",slabel);
                generate(t->right);
                generate(t->left);
                printf("IFTRUE GOTO Label%d\n",slabel);
        }

        if (strcmp(t->info.c,"FOR") == 0) {
                slabel = label++;
                if (t->left->left != NULL)
                        generate(t->left->left);
                printf("Label%d:\n",slabel);
                generate(t->right);
                if (t->left->type == DUMMY){
                        if (t->left->right != NULL){
                                if (t->left->right->right != NULL)
                                        generate(t->left->right->right);
                                if (t->left->right->left != NULL){
                                        generate(t->left->right->left);
                                        printf("IFTRUE GOTO Label%d\n",slabel);
                                }
                                else {
                                        printf("GOTO Label%d\n",slabel);
                                }
                        }
                }
        }

        break;
case NUMOPER:
        if (!isLeaf(t->left) || !isLeaf(t->right))
                leftCount = count++;
        if (!isLeaf(t->right))
                rightCount = count++;

        generate(t->left);
        if (!isLeaf(t->left) || !isLeaf(t->right))
        printf("STORE R1, Temp%d\n",leftCount);

if (!isLeaf(t->right)){
        generate(t->right);
        printf("STORE R1, Temp%d\n",rightCount);
}

if (!isLeaf(t->left) || !isLeaf(t->right))
        printf("LOAD Temp%d, R1\n",leftCount);

if (strcmp(t->info.c,"+") == 0)
        if (!isLeaf(t->right))
                printf("ADD Temp%d, R1\n",rightCount);
        else
                if (t->right->type == STR)
                        printf("ADD %s, R1\n",t->right->info.c);
                else
                        printf("ADD #%d, R1\n",t->right->info.i);

if (strcmp(t->info.c,"-") == 0){
        if (!isLeaf(t->right)){
                printf("NEG Temp%d\n",rightCount);
                printf("ADD Temp%d, R1\n",rightCount);
        }
        else
                if (t->right->type == STR){
                        printf("NEG %s\n",t->right->info.c);
                        printf("ADD %s, R1\n",t->right->info.c);
                }
                else{
                        printf("NEG #%d\n",t->right->info.i);
                        printf("ADD #%d, R1\n",t->right->info.i);
                }
}

if (strcmp(t->info.c,"*") == 0)
        if (!isLeaf(t->right))
                printf("MULT Temp%d, R1\n",rightCount);
        else
                if (t->right->type == STR)
                        printf("MULT %s, R1\n",t->right->info.c);
                else
                        printf("MULT #%d, R1\n",t->right->info.i);
if (strcmp(t->info.c,"/") == 0)
        if (!isLeaf(t->right))
                printf("DIV R1, Temp%d\n",rightCount);
        else
```

```
                                        if (t->right->type == STR)
                                                printf("DIV R1, %s\n",t->right->info.c);
                                        else
                                                printf("DIV R1, #%d\n",t->right->info.i);

                        if (strcmp(t->info.c,"%") == 0)
                                if (!isLeaf(t->right))
                                        printf("MOD R1, Temp%d\n",rightCount);
                                else
                                        if (t->right->type == STR)
                                                printf("MOD R1, %s\n",t->right->info.c);
                                        else
                                                printf("MOD R1, #%d\n",t->right->info.i);

                        break;
        }
        return;
}
```

6.4. The Make File

An Executable file is used to compile and generate the executable “chiaa” interpreter, which is listed below:

```
lex chiaa.lex
yacc -d chiaa.yacc
cc -o chiaa y.tab.c -ll
```

6.5. Input and Output

Input1:

```
void input_a() {
 int a = 3;
 int b = 4;
 int mult, d;

 mult = a * b;
 d = a * (b + mult);
 b = mult - a - d;
}
```

Output1:

```
Total No Of Lines:9

Code Generated:
LOAD #3, R1
STORE R1, a
LOAD #4, R1
STORE R1, b
LOAD d, R1
LOAD a, R1
MULT b, R1
STORE R1, mult
LOAD a, R1
STORE R1, Temp1
LOAD b, R1
ADD mult, R1
STORE R1, Temp2
LOAD Temp1, R1
MULT Temp2, R1
STORE R1, d
LOAD mult, R1
NEG a
ADD a, R1
STORE R1, Temp3
LOAD Temp3, R1
NEG d
ADD d, R1
STORE R1, b
```

Input2:

```
void input_b() {
 if (i>j)
  i = i + i;
 else if (i<j)
  i = 1;
  else i = 0;
}
```

Output2:

```
Total No Of Lines:7

Code Generated:
LOAD i, R1
STORE R1, temp1
LOAD j, R1
G temp1, R1
IFFALSE GOTO Label1
LOAD i, R1
ADD i, R1
STORE R1, i
GOTO Label2
Label1:
LOAD i, R1
STORE R1, temp2
LOAD j, R1
L temp2, R1
IFFALSE GOTO Label3
LOAD #1, R1
STORE R1, i
GOTO Label4
Label3:
LOAD #0, R1
STORE R1, i
Label4:
Label2:
```

Input3:

```
void input_c() {
 while (i<j && j >k) {
  k = k + 1;
   while ( i==j)
    i = i+2;
 }
}
```

Output3:

```
Total No Of Lines:7

Code Generated:
```

```
LABEL1:
LOAD i, R1
STORE R1, temp2
LOAD j, R1
L temp2, R1
STORE R1, temp1
LOAD j, R1
STORE R1, temp3
LOAD k, R1
G temp3, R1
AND R1, temp1
IFFALSE GOTO Label2
LOAD k, R1
ADD #1, R1
STORE R1, k
LABEL3:
LOAD i, R1
STORE R1, temp4
LOAD j, R1
CMP temp4, R1
IFFALSE GOTO Label4
LOAD i, R1
ADD #2, R1
STORE R1, i
GOTO Label3
LABEL4:
GOTO Label1
LABEL2:
```

7. Reference

[1]. Flex man page
[2]. Yacc man page
[3]. Thomas Niemann, "A Compact Guide to Lex & Yacc", ePaper Press.
[4]. "Lex and YACC primer/HOWTO"
[5]. Stephen C. Johnson, "Yacc : Yet Another Compiler Compiler"
[6]. http://www.unet.unieve.ac.at/aix/aixprggd/genprogc/ie_prog_4lex_yacc.htm#A26F073
[7]. http://cs.wpi.edu/~kal/comp409