INTERMEDIATE CODE GENERATOR (ICG) FOR C-LANGUAGE



NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL

**Date**:

7th November, 2020.

**Submitted To**:

 P.Santhi Thilagam

**Group Members**:

Sai Nishanth  K R(181CO145)

Srikrishna G Yaji(181CO153)

Shashank D(181CO248)

Tarun S Anur(181CO255)

**ABSTRACT**

This report is presented as part of the fourth and the final phase of the Compiler Design Lab Course. In this whole process of the mini-compiler design, we have been able to generate the Three-Address code for a given sample input program. The Lex tool detects all the tokens in the program and gives it to the YACC tool. YACC is able to generate the parse trees and also do semantic checks in parallel to the parse tree generation. In this phase an Intermediate code is to be generated. The various forms of Intermediate Codes include Postfix notation, Three-Address Code, Syntax trees, Directed Acyclic Graphs (DAG) etc. Now, in this phase the Three-Address codes are generated as a part of Intermediate Code in parallel to the previous two phases in order to present the final output of the Compiler Design Lab course. The Three-Address code generated is a machine independent linear representation of the Intermediate Code and comes in handy when we have to convert it into the machine code which is the final output expected from a compiler.

**INDEX**

* Introduction
* Code
* Explanation
* Sample Programs
* Implementation
* Future Work
* Bibliography

**TABLE OF CONTENTS**

* Sample Program 1
* Sample Output 1
* Sample Program 2
* Sample Output 2
* Sample Program 3
* Sample Output 3
* Sample Program 4
* Sample Output 4
* Sample Program 5
* Sample Output 5

**INTRODUCTION**

**What is Intermediate Code:**

* If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
* Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion the same for all the compilers.
* The second part of the compiler, synthesis, is changed according to the target machine.
* It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code). Intermediate code generator receives input from its predecessor phase, semantic analyser, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code.

**Three-Address code**

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in a temporary variable generated by the compiler. The compiler decides the order of operation given by three address code.

**General representation:** x = y op z

x, y, z are operands

op is an operator (arithmetic or logical)

* Assignments:  x = y op z (or) x = op y.
* Copy: x = y.
* Unconditional Jump: goto L. (L is a label)
* Conditional Jumps: IF x relop y goto L.
* Indexed Statements: x[i] = y (or) x = y[i] (where i is an index)
* Address/Pointer: x =&y (or) x = \*y (or) \*x = y
* Parameters: param x.
* Function call: y = call p

**Advantages of Three Address Code:**

* It resembles many machines
* Introduces a new set of names (temporary variables and labels)
* Compact form
* Control Flow is easily understandable.

**Implementation of Three-Address Code:**

A three-address code can be represented in two forms: quadruples and triples. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (registers) to generate code. Triples are more of a space-saving method and more useful when memory is tight. Quadruples are simpler and if space allows, it is the author’s opinion that you use them.

**Back-Patching:**

Back patching is the activity of filling up the unspecified information of labels by using the appropriate semantic expression during the code generation process. It is done by a boolean expression.

Back patching Algorithms perform three types of operations

1) makelist (i) – creates a new list containing only i, an index into the array of quadruples and returns a pointer to the list it has made.

2) Merge (i, j) – concatenates the lists pointed to by i and j, and returns a pointer to the concatenated list.

3) Backpatch (p, i) – inserts i as the target label for each of the statements on the list pointed to by p.

**CODE**

Scanner.l (Updated version)

%{

    #include<stdio.h>

    #include<string.h>

    #include"y.tab.h"

    const int INT\_MAX = 1 << 30;

    int cur\_scope;

    int params\_cnt;

    int funccall\_params\_cnt;

    struct symbol\_table {

        char type[100];

        char name[100];

        char Class[100];

        char value[100];

        int line\_number;

        int valid;

        int scope;

        int param\_cnt;

    }st[501];

    struct constant\_table {

        char type[100];

        char name[100];

        int valid;

    }ct[501];

    struct func\_table {

        char params[100];

        char name[100];

        int valid;

    } funct[501];

    char get\_identifier\_type(char\* matchstr) {

        for(int i = 0; i < 501; i++) {

            if(strcmp(st[i].name, matchstr) == 0) {

                if(st[i].scope > cur\_scope) {

                    printf("LINE: %d ", yylineno);

                    puts("ERROR: Undeclared Variable.");

                    exit(0);

                }

                return st[i].type[0];

            }

        }

    }

    void insert\_symbol\_table\_line(char \*str1, int line)

    {

        for(int i = 0 ; i < 501 ; i++)

        {

            if(strcmp(st[i].name,str1)==0)

            {

                st[i].line\_number = line;

            }

        }

    }

    int hash\_function(char \*str)

    {

        int value = 0;

        for(int i = 0 ; i < strlen(str) ; i++)

        {

            value = 10\*value + (str[i] - 'A');

            value = value % 501;

            while(value < 0)

                value = value + 501;

        }

        return value;

    }

    int lookup\_symbolTable(char \*str)

    {

        int value = hash\_function(str);

        if(st[value].valid == 0)

        {

            return 0;

        }

        else if(strcmp(st[value].name,str)==0)

        {

            return value;

        }

        else

        {

            for(int i = value + 1 ; i!=value ; i = (i+1)%501)

            {

                if(strcmp(st[i].name,str)==0)

                {

                    return i;

                }

            }

            return 0;

        }

    }

    int lookup\_constantTable(char \*str)

    {

        int value = hash\_function(str);

        if(ct[value].valid == 0)

            return 0;

        else if(strcmp(ct[value].name,str)==0)

            return 1;

        else

        {

            for(int i = value + 1 ; i!=value ; i = (i+1)%501)

            {

                if(strcmp(ct[i].name,str)==0)

                {

                    return 1;

                }

            }

            return 0;

        }

    }

    void insert\_symbolTable(char \*str1, char \*str2)

    {

        if(lookup\_symbolTable(str1))

        {

            return;

        }

        else

        {

            int value = hash\_function(str1);

            if(st[value].valid == 0)

            {

                strcpy(st[value].name,str1);

                strcpy(st[value].Class, str2);

                st[value].valid = strlen(str1);

                st[value].scope = INT\_MAX;

                insert\_symbol\_table\_line(str1, yylineno);

                return;

            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%501)

            {

                if(st[i].valid == 0)

                {

                    pos = i;

                    break;

                }

            }

            strcpy(st[pos].name,str1);

            strcpy(st[pos].Class,str2);

            st[pos].valid = strlen(str1);

            st[pos].scope = INT\_MAX;

        }

    }

    void insert\_symbol\_table\_scope(char\* str, int scope) {

        int pos = lookup\_symbolTable(str);

        if(pos && st[pos].scope != INT\_MAX) {

            int val = hash\_function(str);

            for(int i = val, loopCnt = 0; loopCnt < 502; i = (i + 1) % 501, loopCnt++) {

                if(st[i].valid == 1) {

                    if(st[i].scope == scope && strcmp(str, st[i].name) == 0) {

                        printf("LINE: %d ", yylineno);

                        puts("ERROR: DUPLICATE declaration");

                        exit(-1);

                        return;

                    }

                }

            }

            for(int i = val + 1; i != val; i = (i + 1) % 501) {

                if(st[i].valid == 0) {

                    strcpy(st[i].name,str);

                    strcpy(st[i].Class, str);

                    st[i].valid = strlen(str);

                    st[i].scope = scope;

                    st[i].line\_number = yylineno;

                    return;

                }

            }

        } else {

            for(int i = 0; i < 501; i++) {

                if(strcmp(st[i].name, str) == 0) {

                    st[i].scope = scope;

                    return;

                }

            }

        }

    }

    int remove\_scope(int scope) {

        for(int i = 0; i < 501; i++) {

            if(st[i].valid && st[i].scope == scope) {

                st[i].scope = INT\_MAX;

            }

        }

    }

    void insert\_func\_table(char\* func) {

        int val = hash\_function(func);

        if(funct[val].valid == 0) {

            strcpy(funct[val].name, func);

            funct[val].valid = strlen(func);

        } else {

            printf("LINE: %d ", yylineno);

            puts("ERROR: Duplicate Function declaration");

        }

    }

    void insert\_symbol\_table\_params\_cnt(char\* str, int param\_count) {

        int pos = lookup\_symbolTable(str);

        st[pos].param\_cnt = param\_count;

    }

    int verify\_funccall\_cnt(char\* str, int cnt) {

        int pos = lookup\_symbolTable(str);

        return st[pos].param\_cnt == cnt;

    }

    void insert\_arg\_type(char\* type, char\* func, int pos) {

        int posi = lookup\_symbolTable(func);

        funct[posi].params[pos] = type[0];

        if(type[0] == 'v') {

            puts("ERROR: void type parameter!");

            exit(-1);

        }

    }

    int check\_arg\_type(int typid, char\* func, int pos) {

        int posi = lookup\_symbolTable(func);

        if(posi == 0 || funct[posi].valid == 0) {

            printf("LINE: %d ", yylineno);

            puts("ERROR: Function Not Declared");

            exit(-1);

        }

        if(typid == 5 && funct[posi].params[pos] == 'i') {

        } else if(typid == 6 && funct[posi].params[pos] == 'c') {

        }  else {

            printf("LINE: %d ", yylineno);

            puts("ERROR: Arguments mismatch");

            exit(-1);

        }

    }

    void insert\_symbol\_table\_type(char \*str1, char \*str2)

    {

        for(int i = 0 ; i < 501 ; i++)

        {

            if(strcmp(st[i].name,str1)==0)

            {

                strcpy(st[i].type,str2);

            }

        }

    }

    void insert\_symbol\_table\_value(char \*str1, char \*str2)

    {

        for(int i = 0 ; i < 501 ; i++)

        {

            if(strcmp(st[i].name,str1)==0)

            {

                strcpy(st[i].value,str2);

            }

        }

    }

    void insert\_constantsTable(char \*str1, char \*str2)

    {

        if(lookup\_constantTable(str1))

            return;

        else

        {

            int value = hash\_function(str1);

            if(ct[value].valid == 0)

            {

                strcpy(ct[value].name,str1);

                strcpy(ct[value].type,str2);

                ct[value].valid = strlen(str1);

                return;

            }

            int pos = 0;

            for (int i = value + 1 ; i!=value ; i = (i+1)%501)

            {

                if(ct[i].valid == 0)

                {

                    pos = i;

                    break;

                }

            }

            strcpy(ct[pos].name,str1);

            strcpy(ct[pos].type,str2);

            ct[pos].valid = strlen(str1);

        }

    }

    void print\_symbol\_table()

    {

        printf("%10s | %15s | %10s | %10s | %10s  \n","SYMBOL", "CLASS", "TYPE","VALUE", "LINE NO");

        for(int i=0;i<81;i++) {

            printf("-");

        }

        printf("\n");

        for(int i = 0 ; i < 501 ; i++)

        {

            if(st[i].valid == 0)

            {

                continue;

            }

            printf("%10s | %15s | %10s | %10s | %10d  \n",st[i].name, st[i].Class, st[i].type, st[i].value, st[i].line\_number);

        }

    }

    void print\_func\_table() {

        printf("%10s | %10s | %10s \n","Name", "Parameters Count", "PARAM TYPE");

        for(int i=0;i<61;i++) {

            printf("-");

        }

        puts("");

        for(int i = 0; i < 501; i++) {

            if(funct[i].valid) {

                printf("%10s | %d ", funct[i].name, st[i].param\_cnt);

                printf("%17s", " ");

                for(int j = 0; j < st[i].param\_cnt; j++) {

                    printf(" %c, ", funct[i].params[j]);

                }

                puts("");

            }

        }

    }

    void print\_constant\_table()

    {

        printf("%10s | %15s\n","NAME", "TYPE");

        for(int i=0;i<81;i++) {

            printf("-");

        }

        printf("\n");

        for(int i = 0 ; i < 501 ; i++)

        {

            if(ct[i].valid == 0)

                continue;

            printf("%10s | %15s\n",ct[i].name, ct[i].type);

        }

    }

    int cbracketsopen = 0;

    int cbracketsclose = 0;

    int bbracketsopen = 0;

    int bbracketsclose = 0;

    int fbracketsopen = 0;

    int fbracketsclose = 0;

    char Match\_str[20];

    char Match\_type[20];

    char curval[20];

    char cur\_identifier[20];

    char cur\_function[20];

    char cur\_type[20];

%}

identifier [a-zA-Z\_][a-zA-Z0-9\_]\*

numerical\_constants (([0-9]\*\.[0-9]+)|([0-9]+\.[0-9]\*)|([0-9]+))

char\_constants  [\t\n ]\*((\'[a-zA-Z0-9]\')|\'\\a\'|\'\\b\'|\'\\e\'|'\\f\'|\'\\n\'|\'\\n\'|\'\\r\'|\'\\t\'|\'\\v\'|\'\\\'|\'\'\'|\'\"\'|\'\?\')

string\_constants [\t\n ]\*\"(.)\*\"

keywords\_1 auto|double|int|struct|break|else|long|switch|case|enum|register|typedef|char|extern|return|union|continue|for|signed

keywords\_2 void|do|if|static|while|default|goto|volatile|const|float|short|unsigned

multiline\_comment \/\\*([^(\\*/)]\*|(\n)\*)\\*\/

singleline\_comment \/\/(.)\*

binary\_operators \+|\-|\\*|\/|\%

unary\_operators \+\+|\-\-

relational\_operators \=\=|\!\=|\>|\<|\>\=|\<\=

logical\_operators \&\&|\|\||\!

bitwise\_operators \&|\||\^|\<\<|\>\>

assignment\_operators \+\=|\-\=|\\*\=|\/\=|\%\=|\=

special\_operators sizeof

special\_symbols \[|\]|\{|\}|\(|\)|\,|\;

header\_files #(.)\*

constants {numerical\_constants}|{char\_constants}|{string\_constants}

operators {unary\_operators}|{special\_operators}|{logical\_operators}|{relational\_operators}|{bitwise\_operators}|{binary\_operators}|{assignment\_operators}

%%

\n {yylineno++;}

[\n\t' ']\* {

    for(int i = 0; i < strlen(yytext); i++) {

        if(yytext[i] == '\n') yylineno++;

    }

}

{singleline\_comment} ;

{multiline\_comment} {

     for(int i = 0; i < strlen(yytext); i++) {

        if(yytext[i] == '\n') yylineno++;

    }

}

{header\_files} {printf("%s is a header declaration\n", yytext);}

":"             { return(':'); }

"."

{keywords\_1}|{keywords\_2} {

    printf("%s is a keyword\n", yytext);

    if(strcmp(yytext, "auto") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(AUTO);

    }

    else if(strcmp(yytext, "double") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(DOUBLE);

    }

    else if(strcmp(yytext, "int") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(INT);

    }

    else if(strcmp(yytext, "struct") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(STRUCT);

    }

    else if(strcmp(yytext, "break") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(BREAK);

    }

    else if(strcmp(yytext, "else") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(ELSE);

    }

    else if(strcmp(yytext, "long") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(LONG);

    }

    else if(strcmp(yytext, "switch") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(SWITCH);

    }

    else if(strcmp(yytext, "case") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(CASE);

    }

    else if(strcmp(yytext, "enum") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(ENUM);

    }

    else if(strcmp(yytext, "register") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(REG);

    }

    else if(strcmp(yytext, "typedef") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(TYPEDEF);

    }

    else if(strcmp(yytext, "char") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(CHAR);

    }

    else if(strcmp(yytext, "extern") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(EXTERN);

    }

    else if(strcmp(yytext, "return") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(RETURN);

    }

    else if(strcmp(yytext, "union") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(UNION);

    }

    else if(strcmp(yytext, "continue") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(CONTINUE);

    }

    else if(strcmp(yytext, "for") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(FOR);

    }

    else if(strcmp(yytext, "signed") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(SIGNED);

    }

    else if(strcmp(yytext, "void") == 0){

        strcpy(Match\_type, yytext);

        insert\_symbolTable(yytext, "KEYWORD");

        return(VOID);

    }

    else if(strcmp(yytext, "do") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(DO);

    }

    else if(strcmp(yytext, "if") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(IF);

    }

    else if(strcmp(yytext, "static") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(STATIC);

    }

    else if(strcmp(yytext, "while") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(WHILE);

    }

    else if(strcmp(yytext, "default") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(DEFAULT);

    }

    else if(strcmp(yytext, "goto") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(GOTO);

    }

    else if(strcmp(yytext, "volatile") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(VOLATILE);

    }

    else if(strcmp(yytext, "const") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(CONST);

    }

    else if(strcmp(yytext, "float") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(FLOAT);

    }

    else if(strcmp(yytext, "short") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(SHORT);

    }

    else if(strcmp(yytext, "unsigned") == 0){

        insert\_symbolTable(yytext, "KEYWORD");

        return(UNSIGNED);

    }

    }

({numerical\_constants}){identifier} {

  printf("In LineNo: %d, ERROR: Invalid Identifier : %s\n", yylineno, yytext);  exit(1);

}

(\")(\s|{identifier}|{numerical\_constants}|{operators})\* {

    printf("In LineNo: %d, ERROR: String usage error in %s\n", yylineno, yytext);  exit(1);

}

[\n\t ]\*\'(\s|{identifier}|{numerical\_constants}|{operators})\* {

    printf("In LineNo: %d, ERROR: Character usage error: %s\n", yylineno, yytext);  exit(1);

}

{identifier} {

    printf("%s is a identifier\n", yytext);

    strcpy(Match\_str, yytext);

    strcpy(cur\_identifier, yytext);

    insert\_symbolTable(yytext, "Identifier");

    return(IDENTIFIER);

}

{numerical\_constants} {

    printf("%s is a constant\n", yytext);

    strcpy(curval, yytext);

    // insert\_constantsTable(yytext, "Constant");

    for(int i = 0; i < strlen(yytext); i++) {

        if(yytext[i] == '\n') yylineno++;

    }

    insert\_constantsTable(yytext, "NUMERICAL CONSTANT");

    return(NUM\_CONSTANT);

    }

{char\_constants} {

    printf("%s is a constant\n", yytext);

    // insert\_constantsTable(yytext, "Constant");

    strcpy(curval, yytext);

    for(int i = 0; i < strlen(yytext); i++) {

        if(yytext[i] == '\n') yylineno++;

    }

    insert\_constantsTable(yytext, "CHAR CONSTANT");

    return(CHAR\_CONSTANT);

    }

{string\_constants} {

    printf("%s is a constant\n", yytext);

    // insert\_constantsTable(yytext, "Constant");

    strcpy(curval, yytext);

    for(int i = 0; i < strlen(yytext); i++) {

        if(yytext[i] == '\n') yylineno++;

    }

    insert\_constantsTable(yytext, "STRING CONSTANT");

    return(STRING\_CONSTANT);

    }

{special\_symbols} {

    printf("%s is a special symbol\n", yytext);

    if(yytext[0] == ';') { return(';'); }

    else

    if(yytext[0] == ',') { return(','); }

    else

    if(yytext[0] == '{'){

        fbracketsopen++;

        return('{');

    }

    else if(yytext[0] == '}'){

        fbracketsclose++;

        return('}');

    }

    else if(yytext[0] == '('){

        cbracketsopen++;

        return('(');

    }

    else if(yytext[0] == ')'){

        cbracketsclose++;

         return(')');

    }

    else if(yytext[0] == '['){

        bbracketsopen++;

        return('[');

    }

    else if(yytext[0] == ']'){

        bbracketsclose++;

        return(']');

    }

    }

{unary\_operators}|{special\_operators}|{logical\_operators}|{relational\_operators}|{bitwise\_operators}|{binary\_operators}|{assignment\_operators} {

    printf("%s is an operator\n", yytext);

    if(strcmp(yytext, "++") == 0) return increment;

    else if(strcmp(yytext, "--") == 0) return decrement;

    else if(strcmp(yytext, "<<") == 0) return leftshift;

    else if(strcmp(yytext, ">>") == 0) return rightshift;

    else if(strcmp(yytext, "<=") == 0) return lessthanAssignment;

    else if(strcmp(yytext, "<") == 0) return lessthan;

    else if(strcmp(yytext, ">=") == 0) return greaterthanAssignment;

    else if(strcmp(yytext, ">") == 0) return greaterthan;

    else if(strcmp(yytext, "==") == 0) return equality;

    else if(strcmp(yytext, "!=") == 0) return inequality;

    else if(strcmp(yytext, "&&") == 0) return and;

    else if(strcmp(yytext, "||") == 0) return or;

    else if(strcmp(yytext, "^") == 0) return xor;

    else if(strcmp(yytext, "\*=") == 0) return multiplicationAssignment;

    else if(strcmp(yytext, "/=") == 0) return divisionAssignment;

    else if(strcmp(yytext, "%=") == 0) return moduloAssignment;

    else if(strcmp(yytext, "+=") == 0) return additionAssignment;

    else if(strcmp(yytext, "-=") == 0) return subtractionAssignment;

    else if(strcmp(yytext, "<<=") == 0) return leftshiftAssignment;

    else if(strcmp(yytext, ">>=") == 0) return rightshiftAssignment;

    else if(strcmp(yytext, "&=") == 0) return andAssignment;

    else if(strcmp(yytext, "|=") == 0) return orAssignment;

    else if(strcmp(yytext, "&") == 0) return bitAnd;

    else if(strcmp(yytext, "!") == 0) return not;

    else if(strcmp(yytext, "~") == 0) return negation;

    else if(strcmp(yytext, "|") == 0) return bitOr;

    else if(strcmp(yytext, "-") == 0) return subtract;

    else if(strcmp(yytext, "+") == 0) return add;

    else if(strcmp(yytext, "\*") == 0) return multiplication;

    else if(strcmp(yytext, "/") == 0) return divide;

    else if(strcmp(yytext, "%") == 0) return modulo;

    else if(strcmp(yytext, "=") == 0) return assignment;

    }

%%

int yywrap(){

    return(1);

}

Parser.y (Updated version)

%{

    #define \_RED        "\x1b[31m"

    #define ANSI\_COLOR\_GREEN    "\x1b[32m"

    #define ANSI\_COLOR\_CYAN     "\x1b[36m"

    #define \_RESET  "\x1b[0m"

    void yyerror(char\* s);

    int yylex();

    #include <stdio.h>

    #include <stdlib.h>

    #include <string.h>

    void ins();

    void insV();

    int flag=0;

    FILE\* fp;

    int T\_cnt = 0, valtop = 0, lbltop = 0, L\_cnt = 0;

    extern char Match\_str[20];

    extern char Match\_type[20];

    extern char curval[20];

    extern char cur\_identifier[20];

    extern char cur\_function[20];

    extern int cur\_scope;

    extern int params\_cnt;

    extern int funccall\_params\_cnt;

    void insert\_symbol\_table\_scope(char\*, int);

    void insert\_symbol\_table\_params\_cnt(char\*, int);

    void remove\_scope(int);

    int verify\_funccall\_cnt(char\*, int);

    int check\_arg\_type(int , char\* , int);

    void insert\_arg\_type(char\*, char\*, int);

    void insert\_func\_table(char\* );

    char get\_identifier\_type(char\* );

    struct labl {

        char value[500];

        int id;

    } val\_stack[100], labl\_stack[100];

    void val\_push(char\* );

    void TAC();

    void reassign\_TAC();

    void identifier\_TAC();

    void constant\_TAC();

    void if\_not\_goto();

    void if\_end\_goto();

    void if\_end\_label();

    void iter\_label();

    void iter\_loop\_label();

    void TAC\_assign();

%}

%nonassoc IF

%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT

%token RETURN MAIN

%token VOID

%token WHILE FOR DO

%token BREAK

%token ENDIF

%token AUTO SWITCH CASE ENUM REG TYPEDEF EXTERN UNION CONTINUE STATIC DEFAULT GOTO VOLATILE CONST IDENTIFIER NUM\_CONSTANT CHAR\_CONSTANT STRING\_CONSTANT

%nonassoc ELSE

%right leftshiftAssignment rightshiftAssignment

%right xorAssignment orAssignment

%right andAssignment moduloAssignment

%right multiplicationAssignment divisionAssignment

%right additionAssignment subtractionAssignment

%right assignment

%left or

%left and

%left bitOr

%left xor

%left bitAnd

%left equality inequality

%left lessthanAssignment lessthan greaterthanAssignment greaterthan

%left leftshift rightshift

%left add subtract

%left multiplication divide modulo

%right SIZEOF

%right negation not

%left increment decrement

%start program

%%

program

            : declaration\_list;

declaration\_list

            : declaration D

D

            : declaration\_list

            | ;

declaration

            : variable\_declaration

            | function\_declaration

            | structure\_definition;

variable\_declaration

            : type\_specifier variable\_declaration\_list ';' {

            }

            | structure\_declaration;

variable\_declaration\_list

            : variable\_declaration\_identifier V ;

V

            : ',' variable\_declaration\_list {

                $$ = $2;

            }

            | ;

variable\_declaration\_identifier

            : IDENTIFIER {ins(), insert\_symbol\_table\_scope(cur\_identifier, cur\_scope); val\_push(cur\_identifier);} vdi {

                if($3 != 127) reassign\_TAC();

                char type = get\_identifier\_type(cur\_identifier);

                if(type == 'i' && $3 == 5) $$ = 5;

                else if(type == 'c' && $3 == 6) $$ = 6;

                else if($3 != 127) {

                    puts("ERROR:  Declaration type Mismatch.\n");

                    yyerror("");

                }

            };

vdi : identifier\_array\_type {$$ = 127;} | assignment expression {

    $$ = $2;

};

identifier\_array\_type

            : '[' initilization\_params

            | ;

initilization\_params

            : subtract NUM\_CONSTANT ']' initilization {puts("ERROR: Array size negative!!"); yyerror("");}

            | NUM\_CONSTANT {if(atoi(curval) == 0) {puts("ERROR: Array Size is 0!!"); yyerror("");}} ']' initilization

            | ']' string\_initilization;

initilization

            : string\_initilization

            | array\_initialization

            | ;

type\_specifier

            : INT {$$ = 5;}| CHAR {$$ = 6;}| FLOAT {$$ = 5;}| DOUBLE

            | LONG long\_grammar

            | SHORT short\_grammar

            | UNSIGNED unsigned\_grammar

            | SIGNED signed\_grammar

            | VOID ;

unsigned\_grammar

            : INT | LONG long\_grammar | SHORT short\_grammar | ;

signed\_grammar

            : INT | LONG long\_grammar | SHORT short\_grammar | ;

long\_grammar

            : INT | ;

short\_grammar

            : INT | ;

structure\_definition

            : STRUCT IDENTIFIER { ins(); } '{' V1  '}' ';';

V1 : variable\_declaration V1 | ;

structure\_declaration

            : STRUCT IDENTIFIER variable\_declaration\_list;

function\_declaration

            : function\_declaration\_type function\_declaration\_param\_statement;

function\_declaration\_type

            : type\_specifier IDENTIFIER '('  { params\_cnt = 0; ins(); strcpy(cur\_function, cur\_identifier); insert\_symbol\_table\_scope(cur\_identifier, cur\_scope); insert\_func\_table(cur\_function); fprintf(fp, "function %s:\n ", cur\_function);};

function\_declaration\_param\_statement

            : params ')' statement;

params

            : parameters\_list | ;

parameters\_list

            : type\_specifier {insert\_arg\_type(Match\_type, cur\_function, params\_cnt);} parameters\_identifier\_list {insert\_symbol\_table\_params\_cnt(cur\_function, params\_cnt);};

parameters\_identifier\_list

            : param\_identifier parameters\_identifier\_list\_breakup;

parameters\_identifier\_list\_breakup

            : ',' parameters\_list

            | ;

param\_identifier

            : IDENTIFIER { ins(); insert\_symbol\_table\_scope(cur\_identifier, cur\_scope+1); params\_cnt++;} param\_identifier\_breakup;

param\_identifier\_breakup

            : '[' ']'

            | ;

statement

            : expression\_statment | compound\_statement

            | conditional\_statements | iterative\_statements

            | return\_statement | break\_statement

            | variable\_declaration;

compound\_statement

            : {cur\_scope++;} '{' statment\_list '}' {remove\_scope(cur\_scope); cur\_scope--;};

statment\_list

            : statement statment\_list

            | ;

expression\_statment

            : expression ';'

            | ';' ;

conditional\_statements

            : IF '(' simple\_expression {if\_not\_goto();} ')' statement {if\_end\_goto();} conditional\_statements\_breakup {if\_end\_label();};

conditional\_statements\_breakup

            : ELSE statement

            | ;

iterative\_statements

            : WHILE '(' {iter\_label();} simple\_expression {if\_not\_goto();} ')' statement {iter\_loop\_label();}

            | FOR '(' expression ';' {iter\_label();} simple\_expression {if\_not\_goto();}';' expression ')' statement {iter\_loop\_label();}

            | {iter\_label();} DO statement WHILE '(' simple\_expression {if\_not\_goto(); iter\_loop\_label();} ')' ';';

return\_statement

            : RETURN return\_statement\_breakup {

                if($2 == 5 && get\_identifier\_type(cur\_function) == 'i') {

                } else if($2 == 6 && get\_identifier\_type(cur\_function) == 'c') {

                } else if(!($2 == 127 && get\_identifier\_type(cur\_function) == 'v')){

                    puts("ERROR: RETURN Type mismatch!");

                    yyerror(cur\_function);

                }

            };

return\_statement\_breakup

            : ';' {$$ = 127;}

            | expression ';' {$$ = $1;};

break\_statement

            : BREAK ';' ;

string\_initilization

            : assignment STRING\_CONSTANT { insV(); };

array\_initialization

            : assignment '{' array\_int\_declarations '}';

array\_int\_declarations

            : NUM\_CONSTANT array\_int\_declarations\_breakup;

array\_int\_declarations\_breakup

            : ',' array\_int\_declarations

            | ;

expression

            : mutable expression\_breakup {

                if($1 != $2) {

                    printf("ERROR: Type Mismatch.\n");

                    yyerror("");

                } else if($1 == 5) {

                    $$ = 5;

                } else if($1 == 6) {

                    $$ = 6;

                }

            }

            | simple\_expression {

                // reassign\_TAC();

                $$ = $1;

            };

expression\_breakup

            : assignment expression {

                reassign\_TAC();

                $$ = $2;

            }

            | additionAssignment {val\_push("+");} expression {

                TAC\_assign();

                $$ = $2;

            }

            | subtractionAssignment {val\_push("-");} expression {

                TAC\_assign();

                $$ = $2;

            }

            | multiplicationAssignment {val\_push("\*");} expression {

                TAC\_assign();

                $$ = $2;

            }

            | divisionAssignment {val\_push("/");} expression {

                $$ = $2;

            }

            | moduloAssignment {val\_push("/");} expression {

                TAC\_assign();

                $$ = $2;

            }

            | increment {val\_push("+"); val\_push("1");}{

                TAC\_assign();

                $$ = 5;

            }

            | decrement {val\_push("-"); val\_push("1");}{

                TAC\_assign();

                $$  = 5;

            };

simple\_expression

            : and\_expression simple\_expression\_breakup {

                if($1 != -98 && $2 != -98) TAC();

                $$ = $1;

            };

simple\_expression\_breakup

            : or {val\_push("||");} and\_expression simple\_expression\_breakup {}| {$$ = -98;};

and\_expression

            : unary\_relation\_expression and\_expression\_breakup {

                if($2 != -98 && $1 != -98) TAC();

                $$ = $1;

            };

and\_expression\_breakup

            : and {val\_push("&&");} unary\_relation\_expression and\_expression\_breakup

            | {$$ = -98;};

unary\_relation\_expression

            : not unary\_relation\_expression

            | regular\_expression {

                $$ = $1;

            };

regular\_expression

            : sum\_expression regular\_expression\_breakup {

                {

                    if($2 != -98 && $1 != -98) TAC();

                }

                if($1 == $2) {

                    $$ = $1;

                }

            };

regular\_expression\_breakup

            : relational\_operators sum\_expression {

                $$ = $2;

            }

            | {$$ = -98;};

relational\_operators

            : greaterthanAssignment {val\_push(">=");} | lessthanAssignment {val\_push("<=");}| greaterthan {val\_push(">");}

            | lessthan {val\_push("<");}| equality {val\_push("==");}| inequality {val\_push("!=");};

sum\_expression

            : sum\_expression sum\_operators term {

                TAC();

                if($1 == $3)

                    $$ = $1;

                else {

                    printf("ERROR: Type mismatch.\n");

                    yyerror("");

                }

            }

            | term {$$ = $1;};

sum\_operators

            : add {val\_push("+");};

            | subtract {val\_push("-");};

term

            : term MULOP factor {

                TAC();

                if($1 == $3)

                    $$ = $1;

                else {

                    printf("ERROR: Type mismatch");

                    yyerror("");

                };

            }

            | factor {$$ = $1;};

MULOP

            : multiplication {val\_push("\*");}| divide {val\_push("/");} | modulo {val\_push("%");};

factor

            : immutable {$$ = $1;}| mutable ;

mutable

            : IDENTIFIER {

                // check identifire type and return;

                val\_push(cur\_identifier);

                char type = get\_identifier\_type(cur\_identifier);

                if(type == 'i') $$ = 5;

                if(type == 'c') $$ = 6;

            }

            | mutable mutable\_breakup {

                if($2 == 5 || $1 == 5)

                    $$ = 5;

                else

                    printf("ERROR: Type Mismatch");

                    yyerror("");

            };

mutable\_breakup

            : '[' expression ']'

            | '.' IDENTIFIER {if( $2 == 5) $$ = 5;};

immutable

            : '(' expression ')' {

                if($2 == 5) $$ = 5;

            }

            | call {

                if($1 == 5) $$ = 5;

            }

            | constant {

                if($1 == 5) $$ = 5;

            };

call

            : IDENTIFIER '(' {strcpy(cur\_function, cur\_identifier);} arguments ')' {

                char type = get\_identifier\_type(cur\_function);

                fprintf(fp, "goto %s\n", cur\_function);

                if(type == 'i') $$ = 5;

                if(type == 'c') $$ = 6;

                if(!verify\_funccall\_cnt(cur\_function, funccall\_params\_cnt)) {

                    puts("ERROR: Function Call arguments count mismatch");

                    yyerror(cur\_function);

                }

                char tmp[100] = {0};

                sprintf(tmp, "%d", funccall\_params\_cnt);

                fprintf(fp, "pop\_params %s\n", tmp);

            };

arguments

            : arguments\_list | ;

arguments\_list

            : {funccall\_params\_cnt = 0;} expression {fprintf(fp, "push\_param %s\n", val\_stack[valtop].value); check\_arg\_type($2, cur\_function, funccall\_params\_cnt);funccall\_params\_cnt++;} A;

A

            : ',' expression {fprintf(fp, "push\_param %s\n", val\_stack[valtop].value); check\_arg\_type($2, cur\_function, funccall\_params\_cnt);;funccall\_params\_cnt++;} A

            | ;

constant

            : NUM\_CONSTANT  { insV(); constant\_TAC(); $$=5;}

            | STRING\_CONSTANT   { insV(); constant\_TAC();}

            | CHAR\_CONSTANT{ insV(); constant\_TAC(); $$=6;};

%%

extern FILE \*yyin;

extern int yylineno;

extern char \*yytext;

extern int cbracketsopen;

extern int cbracketsclose;

extern int bbracketsopen;

extern int bbracketsclose;

extern int fbracketsopen;

extern int fbracketsclose;

void insert\_symbol\_table\_type(char \*,char \*);

void insert\_symbol\_table\_value(char \*, char \*);

void insert\_constantsTable(char \*, char \*);

void print\_constant\_table();

void print\_symbol\_table();

void print\_func\_table();

void val\_push(char\* str) {

    strcpy(val\_stack[++valtop].value, str);

}

void if\_end\_label() {

    char code[100] = {0};

    strcpy(code, "L");

    sprintf(code + 1, "%d", labl\_stack[lbltop].id);

    printf(\_RED "%s: \n" \_RESET, code);

    fprintf(fp, "%s: \n", code);

    lbltop--;

}

void if\_not\_goto() {

    char code[100] = {0};

    strcpy(code, "L");

    sprintf(code + 1, "%d", L\_cnt);

    printf(\_RED "if not %s goto %s\n" \_RESET, val\_stack[valtop].value, code);

    fprintf(fp, "if not %s goto %s\n", val\_stack[valtop].value, code);

    labl\_stack[++lbltop].id = L\_cnt++;

}

void if\_end\_goto() {

    char code[100] = {0};

    strcpy(code, "L");

    sprintf(code + 1, "%d", L\_cnt);

    printf(\_RED "goto %s\n" \_RESET, code);

    fprintf(fp,  "goto %s\n" , code);

    code[0] = 'L';

    sprintf(code + 1, "%d", labl\_stack[lbltop].id);

    printf(\_RED "%s: \n" \_RESET, code);

    fprintf(fp, "%s: \n", code);

    labl\_stack[lbltop].id = L\_cnt++;

}

void iter\_label() {

    char code[100] = {0};

    strcpy(code, "L");

    sprintf(code + 1, "%d", L\_cnt);

    fprintf(fp, "%s:\n", code);

    labl\_stack[++lbltop].id = L\_cnt++;

}

void iter\_loop\_label()  {

    char code[100] = {0};

    strcpy(code, "L");

    sprintf(code + 1, "%d", labl\_stack[lbltop-1].id);

    fprintf(fp, "goto %s\n", code);

    sprintf(code + 1, "%d", labl\_stack[lbltop].id);

    fprintf(fp, "%s: \n", code);

    lbltop -= 2;

}

void identifier\_TAC()  {

    char code[100] = {0};

    strcpy(code, "T");

    sprintf(code + 1, "%d", T\_cnt);

    printf(\_RED "%s = %s\n" \_RESET , code, cur\_identifier);

    fprintf(fp, " %s = %s\n" , code, cur\_identifier);

    T\_cnt++;

    val\_push(code);

}

void constant\_TAC() {

    char code[100] = {0};

    strcpy(code, "T");

    sprintf(code + 1, "%d", T\_cnt);

    printf(\_RED "%s = %s\n" \_RESET , code, curval);

    fprintf(fp,  " %s = %s\n"  , code, curval);

    T\_cnt++;

    val\_push(code);

}

void reassign\_TAC() {

    if(valtop-1 < 0) return;

    // if(val\_stack[valtop].value[0] == 0 || val\_stack[valtop-1].value[0] == 0) return;

    printf(\_RED "%s = %s\n" \_RESET, val\_stack[valtop-1].value, val\_stack[valtop].value);

    fprintf(fp, " %s = %s\n", val\_stack[valtop-1].value, val\_stack[valtop].value);

    valtop -= 2;

}

void TAC() {

    char code[100] = {0};

    strcpy(code, "T");

    if(valtop-2 < 0) return;

    sprintf(code + 1, "%d", T\_cnt);

    printf(\_RED "%s = %s %s %s\n" \_RESET, code, val\_stack[valtop-2].value, val\_stack[valtop-1].value, val\_stack[valtop].value);

    fprintf(fp, "%s = %s %s %s\n", code, val\_stack[valtop-2].value, val\_stack[valtop-1].value, val\_stack[valtop].value);

    valtop -= 2;

    strcpy(val\_stack[valtop].value, code);

    T\_cnt++;

}

void TAC\_assign() {

    char code[100] = {0};

    strcpy(code, "T");

    sprintf(code + 1, "%d", T\_cnt);

    printf(\_RED "%s = %s %s %s\n" \_RESET, code, val\_stack[valtop-2].value, val\_stack[valtop-1].value, val\_stack[valtop].value);

    fprintf(fp, "%s = %s %s %s\n", val\_stack[valtop-2].value, val\_stack[valtop-2].value, val\_stack[valtop-1].value, val\_stack[valtop].value);

    valtop -= 2;

}

int main(int argc , char \*\*argv)

{

    fp = fopen("TAC.txt", "w");

    yyin = fopen(argv[1], "r");

    yyparse();

    if((bbracketsopen-bbracketsclose)){

        printf("ERROR: brackets error [\n");

        // yyerror("ERROR: brackets error [\n");

        flag = 1;

    }

    if((fbracketsopen-fbracketsclose)){

        printf("ERROR: brackets error {\n");

        // yyerror("ERROR: brackets error {\n");

        flag = 1;

    }

    if((cbracketsopen-cbracketsclose)){

        printf("ERROR: brackets error (\n");

        // yyerror("ERROR: brackets error (\n");

        flag = 1;

    }

    if(flag == 0)

    {

        printf("Status: Parsing Complete - Valid\n");

        printf("SYMBOL TABLE\n");

        printf("%30s %s\n", " ", "------------");

        print\_symbol\_table();

        printf("\n\nCONSTANT TABLE\n");

        printf("%30s %s\n", " ", "--------------");

        print\_constant\_table();

        printf("%30s %s\n", " ", "--------------");

        print\_func\_table();

    }

    fclose(fp);

}

void yyerror(char \*s)

{

    puts("=========================================================================");

    printf("Parsing Failed at line no: %d\n", yylineno);

    printf("Error: %s\n", yytext);

    // exit(0);

    // flag=1;

}

void ins()

{

    insert\_symbol\_table\_type(Match\_str,Match\_type);

}

void insV()

{

    insert\_symbol\_table\_value(Match\_str,curval);

}

**EXPLANATION**

In Phase one of the compiler design we implemented a scanner which detected all the tokens in the language. Any unmatched token would lead to lexical errors. These tokens detected were input to the Phase two of our design which defined all the production rules of the language. Phase three of the design checks the semantic structure of the language by embedding the semantic rules with the productions defined in Phase two. The Syntax Directed Translation (SDT) actions detect for errors and also perform necessary tasks if any. Now in Phase four of our design, we generate an Intermediate Representation (IR) for our language. The IR we use here is the Three-Address Code which is very much similar to machine language.

**Declaration Section**

In this section we have included all the header files, function declaration and variables required for the program. This is followed by all the tokens as detected by the Scanner according to its precedence. Operators are declared according to their precedence as well as their associativity. This is done to eliminate all the Shift/Reduce and Reduce/Reduce conflicts in the parse table as generated by the YACC compiler.

**Rules Section**

This section contains all the production rules in accordance with the C-Language. The grammar rules written are detected by the YACC Compiler. Here, we have also defined the semantic rules which define the semantics of the C-Language. Any errors related to either the Grammar Productions or the Semantic Rules are reported in this part of the program.

**C-Program Section**

This section of the code has the main function form which the execution part starts. A few variables are declared to work with some functions that are needed accordingly. Here we have defined functions necessary to generate the temporary variables and the temporary labels in order to keep track of the control flow in the program compilation part. Some functions necessary for back-patching are also defined to complete the entire frontend of the compiler in a single pass only. The external files generated by the Lexer are linked here and the Symbol Table as well as the Constant Table are printed out as a result of this phase.

**SAMPLE PROGRAMS**

Sample Program 1:

Sample Output 1:

Sample Program 2:

Sample Output 2:

Sample Program 3:

Sample Output 3:

Sample Program 4;

Sample Output 4:

Sample Program 5:

Sample Output 5:

**IMPLEMENTATION**

There are a few extra functions implemented in order to generate the three-address code for the given program. The functions implemented are as follows:

1. **val\_push()** : Pushes temp variables, identifiers to stack for further processing.

2. **TAC()** : Generated Three-Address code using the contents of the stack.

3. **identifier\_TAC()** : Three-Address code for identifier.

4. **constant\_TAC()** : Three-Address code for constants assignment.

5. **if\_not\_goto()** : Pushes label to stack for conditional statements.

6. **if\_end\_goto()** : Goto label at the end of conditional statements.

7. **if\_end\_label()** : Label at the end for conditional statements.

8. **iter\_label()** : Label for iterative statements (for, while).

9. **iter\_loop\_label()** : Goto label for looping.

10. **TAC\_assign()** : Handle +=1, ++ etc..

**FUTURE DEVELOPMENTS**

The program written can be made even more robust so as to even optimise the three-address code generated as the output of this phase. Code can be refactored to make it even more understandable.

**BIBLIOGRAPHY**

1. http://dinosaur.compilertools.net/

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