SEMANTIC ANALYSER FOR C-LANGUAGE



NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA SURATHKAL

**Date**:

21st October, 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**

Semantic analysis provides meaning to the constructs of the language like tokens. Grammar written is verified against some other semantic rules so as to preserve the semantics of the language. Semantics of a language help in interpreting symbols, their types and their relations with each other. Context Free Grammar (CFG) along with a set of semantic rules forms the final version of the semantic analyser. Both the parse tree of the previous phase and symbol table are used to check the semantics of the given language**.**

The input to this phase is the parse tree generated by the previous phase. The output of this phase is the Syntax Directed Translation (SDT) Tree where each of the grammar symbols are associated with a set of attributes and each grammar production is associated with a set of Semantic Rules. Attributes include the type, scope or even a small programming snippet. The Semantic Rules are rules to compute the attribute characteristics of each grammar production in the form of an Annotated Parse Tree. Annotated Parse Trees are Parse Trees attached with their corresponding attribute info.

**INDEX**

**TABLE OF CONTENTS**

**INTRODUCTION**

After the lexical analysis stage, we get the stream of tokens from source C code which is given as input to the parser. Parser verifies that a string of token names can be generated by the grammar of the source language. We expect the parser to check the structure of the input program and report any syntax errors. Semantic analysis phase checks the semantics of the language. Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

**Functions of Semantic Analyser:**

1. **Type Checking –** Expressions having the same data types on either side of the assignment operators are accepted. In case of different data types, the semantic analyser throws a type error.
2. **Array Bound Checking** – Array declaration with size less than 1 is thrown as an error.
3. **Multiple declaration -** Identifiers declared multiple times in the same scope leads to Duplicate declaration error.
4. **Function Semantics –** Type or count mismatch in the formal and actual parameters of the function declaration throws an Error. Function calls without declaration are also reported as errors.

**Semantic Analyzer:**

It uses syntax tree and symbol table to check whether the given program is semantically consistent with language definition rules. It gathers type information and stores it in either a syntax tree or symbol table. This type information is the output of this phase and the Intermediate-Code Generation (ICG) phase follows this semantic analysis phase.

**CODE**

Scanner.l

%{

    #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) {

                puts(st[i].type);

                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) {

                        puts("ERROR: DUPLICATE declaration=============");

                        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) {

                printf("SCOPE: +++ %d\n", scope);

                puts(st[i].name);

                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 {

            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);

        puts("INSERTING Function ==================");

        puts(func);

        puts(type);

        printf("POS: %d\n", pos);

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

    }

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

        int posi = lookup\_symbolTable(func);

        puts("FOR: Function ==================");

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

            puts("ERROR: Function Not Declared");

        }

        puts(func);

        printf("TYPE: %d, actual type: %s\n", typid, funct[posi].params);

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

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

        }  else {

            puts("ERROR: ARguments Type mismatch");

        }

    }

    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 | %10s | %10s \n","SYMBOL", "CLASS", "TYPE","VALUE", "LINE NO", "SCOPE", "PARAM\_CNT");

        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 | %10d | %10d \n",st[i].name, st[i].Class, st[i].type, st[i].value, st[i].line\_number, st[i].scope, st[i].param\_cnt);

        }

    }

    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

%{

    void yyerror(char\* s);

    int yylex();

    #include "stdio.h"

    #include "stdlib.h"

    #include "string.h"

    void ins();

    void insV();

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

    int flag=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\* );

%}

%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 ';' {

                if($1 != $2) {

                    puts("variable\_declaration mismatch");

                }

            }

            | 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);} vdi {

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

                printf("%d ->\_\_\_$2\n", $3);

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

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

                else if($3 != 127) {

                    puts("ERROR\_MISMATCH DECL ");

                }

            };

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

    printf("EXP\_\_GIVES %d\n", $2);

    $$ = $2;

    printf("$$ is\_\_\_%d\n", $$);

};

identifier\_array\_type

            : '[' initilization\_params

            | ;

initilization\_params

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

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

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 ')' statement conditional\_statements\_breakup;

conditional\_statements\_breakup

            : ELSE statement

            | ;

iterative\_statements

            : WHILE '(' simple\_expression ')' statement

            | FOR '(' expression ';' simple\_expression ';' expression ')'

            | DO statement WHILE '(' simple\_expression ')' ';';

return\_statement

            : RETURN return\_statement\_breakup;

return\_statement\_breakup

            : ';'

            | expression ';' ;

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("EXPR\_\_\_mismatch\n");

                } else if($1 == 5) {

                    $$ = 5;

                } else if($1 == 6) {

                    $$ = 6;

                }

            }

            | simple\_expression {

                puts("MATCHING HERE");

                $$ = $1;

            };

expression\_breakup

            : assignment expression {

                $$ = $2;

            }

            | additionAssignment expression {

                $$ = $2;

            }

            | subtractionAssignment expression {

                $$ = $2;

            }

            | multiplicationAssignment expression {

                $$ = $2;

            }

            | divisionAssignment expression {

                $$ = $2;

            }

            | moduloAssignment expression {

                $$ = $2;

            }

            | increment

            | decrement ;

simple\_expression

            : and\_expression simple\_expression\_breakup {

                $$ = $1;

            };

simple\_expression\_breakup

            : or and\_expression simple\_expression\_breakup | ;

and\_expression

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

                $$ = $1;

            };

and\_expression\_breakup

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

            | ;

unary\_relation\_expression

            : not unary\_relation\_expression

            | regular\_expression {

                $$ = $1;

            };

regular\_expression

            : sum\_expression regular\_expression\_breakup {

                $$ = $1;

            };

regular\_expression\_breakup

            : relational\_operators sum\_expression {

                $$ = $2;

            }

            | ;

relational\_operators

            : greaterthanAssignment | lessthanAssignment | greaterthan

            | lessthan | equality | inequality ;

sum\_expression

            : sum\_expression sum\_operators term {

                if($1 == 5 && $3 == 5)

                    $$ = 5;

                else

                    printf("Type mismatch");

            }

            | term {$$ = $1;};

//

sum\_operators

            : add

            | subtract ;

term

            : term MULOP factor {

                if($1 == $3)

                    $$ = $1;

                else

                    {

                        printf("Type mismatch");

                    };

            }

            | factor {$$ = $1;};

MULOP

            : multiplication | divide | modulo ;

factor

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

mutable

            : IDENTIFIER {

                // check identifire type and return;

                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("EERROR");

            };

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 ')' {

                puts(cur\_identifier);

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

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

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

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

                    puts("ERROR: \_\_\_\_FUnction Call argumetns mismatch");

                }

            };

arguments

            : arguments\_list | ;

arguments\_list

            : {funccall\_params\_cnt = 0;} expression {check\_arg\_type($2, cur\_function, funccall\_params\_cnt);funccall\_params\_cnt++;} A;

A

            : ',' expression {check\_arg\_type($2, cur\_function, funccall\_params\_cnt);;funccall\_params\_cnt++;} A

            | ;

constant

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

            | STRING\_CONSTANT   { insV(); }

            | CHAR\_CONSTANT{ insV(); $$=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();

int main(int argc , char \*\*argv)

{

    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();

    }

}

void yyerror(char \*s)

{

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

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

    printf("Error: %s\n", yytext);

    flag=1;

}

void ins()

{

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

}

void insV()

{

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

}

**EXPLANATION**

In Phase 1 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 2 of our design which defined all the production rules of the language. This phase checks the structure of the program with respect to its grammar productions. The parse tree generated in this phase becomes input to the next phase which is the semantic phase. Here, the grammar productions defined are checked for specific attributes or semantics of the program. The output of this phase is the Syntax Directed Translation (SDT) Tree which has information about attributes in the grammar.

**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 defines 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. The external files generated by the Lexer is linked here and the Symbol Table as well as the Constant Table are printed out as a result of this phase.

**SAMPLE PROGRAMS**

**ERROR HANDLING**

1. The unary operations and their semantics are yet to be defined.

**FUTURE DEVELOPMENTS**

The program written can be made even more robust to detect all the semantic errors defined in the C- Language. Code can be refactored to make it even more understandable.

**BIBLIOGRAPHY**

1. https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/

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

3.