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 defined as a non-positive integer 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) {
                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, lo
opCnt++) {

                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].Cl
ass, 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|ty
pedef|char|extern|return|union|continue|for|signed
keywords_2 void|do|if|static|while|default|goto|volatile|const|float|short|uns
igned
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}|{relationa
l_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, yytex
t);  exit(1);
}

[\n\t ]*\'(\s|{identifier}|{numerical_constants}|{operators})* {
    printf("In LineNo: %d, ERROR: Character usage error: %s\n", yylineno, yyte
xt);  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_operator
s}|{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();
    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* );
    char get_identifier_type(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 ';' {

            }
            | 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);
                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(c
ur_function, cur_identifier); insert_symbol_table_scope(cur_identifier, cur_sc
ope); 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_functio
n, 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, cu
r_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); c
ur_scope--;};

statment_list
            : statement statment_list
            | ;

expression_statment
            : expression ';'
            | ';' ;

conditional_statements
            : IF '(' simple_expression ')' statement conditional_statements_br
eakup;

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 {
                // printf(get_ideni)
                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 {
                $$ = $1;
            };

expression_breakup
            : assignment expression {
                $$ = $2;
            }
            | additionAssignment expression {
                $$ = $2;
            }
            | subtractionAssignment expression {
                $$ = $2;
            }
            | multiplicationAssignment expression {
                $$ = $2;
            }
            | divisionAssignment expression {
                $$ = $2;
            }
```

```
            | moduloAssignment expression {
                $$ = $2;
            }
            | increment {
                $$ = 5;
            }
            | decrement {
                $$  = 5;
            };

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)
                    $$ = 5;
                else
                    printf("ERROR: Type mismatch.\n");
                    yyerror("");
            }
            | term {$$ = $1;};

sum_operators
            : add
            | subtract ;

term
            : term MULOP factor {
                if($1 == $3)
                    $$ = $1;
                else
                    {
                        printf("ERROR: Type mismatch");
                        yyerror("");
                    };
            }
            | factor {$$ = $1;};

MULOP
            : multiplication | divide | modulo ;

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

mutable
            : IDENTIFIER {
                // check identifire type and return;
                puts(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
 ')' {
                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 arguments count mismatch");
                    yyerror(cur_function);
                }
            };

arguments
            : arguments_list | ;

arguments_list
            : {funccall_params_cnt = 0;} expression {check_arg_type($2, cur_fu
nction, 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();
void print_func_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();
        printf("%30s %s\n", " ", "--------------");
        print_func_table();
```

```
    }
}


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 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

Sample Program 1:

```c
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int i, sum = 0;
    for (i = 0;i < 10;i++)
    {
       sum += i;
    }

}
```

Sample Output 1:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
#include<stdio.h> is a header declaration
void is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
i is a identifier
, is a special symbol
sum is a identifier
= is an operator
0 is a constant
; is a special symbol
for is a keyword
( is a special symbol
i is a identifier
= is an operator
i
0 is a constant
; is a special symbol
i is a identifier
< is an operator
i
10 is a constant
; is a special symbol
i is a identifier
++ is an operator
i
) is a special symbol
{ is a special symbol
sum is a identifier
+= is an operator
sum
i is a identifier
; is a special symbol
i
} is a special symbol
} is a special symbol
Status: Parsing Complete - Valid
SYMBOL TABLE
                                ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         i |      Identifier |        int |         10 |          6
      main |      Identifier |       void |            |          4
       sum |      Identifier |        int |          0 |          6
       for |         KEYWORD |            |            |          7
      void |         KEYWORD |            |            |          4
       int |         KEYWORD |            |            |          6
```

Sample Program 2:

```c
#include<stdio.h>
#include<stdlib.h>
int square(int a) {
    return a*a;
}
int cube(int b) {
    return b*b*b;
}
void main() {
    int num = 10;
    int s = square(num);
    int c = cube(num);
}
```

Sample Output 2:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
int is a keyword
square is a identifier
( is a special symbol
int is a keyword
a is a identifier
) is a special symbol
{ is a special symbol
return is a keyword
a is a identifier
* is an operator
a
a is a identifier
; is a special symbol
a
} is a special symbol
int is a keyword
cube is a identifier
( is a special symbol
int is a keyword
b is a identifier
) is a special symbol
{ is a special symbol
return is a keyword
b is a identifier
* is an operator
b
b is a identifier
* is an operator
b
b is a identifier
; is a special symbol
b
} is a special symbol
void is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
num is a identifier
= is an operator
10 is a constant
; is a special symbol
int is a keyword
s is a identifier
= is an operator
square is a identifier
```

```
( is a special symbol
num is a identifier
) is a special symbol
num
num
; is a special symbol
int is a keyword
c is a identifier
= is an operator
cube is a identifier
( is a special symbol
num is a identifier
) is a special symbol
num
num
; is a special symbol
} is a special symbol
Status: Parsing Complete - Valid
SYMBOL TABLE
                                ------------
    SYMBOL |           CLASS |       TYPE |      VALUE |    LINE NO
-------------------------------------------------------------------------------
         a |      Identifier |        int |            |          4
         b |      Identifier |        int |            |          9
         c |      Identifier |        int |            |         18
      main |      Identifier |       void |            |         14
         s |      Identifier |        int |            |          0
       num |      Identifier |        int |         10 |         16
    return |         KEYWORD |            |            |          6
    square |      Identifier |        int |            |          4
      void |         KEYWORD |            |            |         14
      cube |      Identifier |        int |            |          9
       int |         KEYWORD |            |            |          4


CONSTANT TABLE
                                ------------
      NAME |            TYPE
-------------------------------------------------------------------------------
        10 | NUMERICAL CONSTANT
                               -------------
      Name | Parameters Count | PARAM TYPE
-----------------------------------------------------------
      main | 0
    square | 1                  i,
      cube | 1                  i,
```

Sample Program 3:

```c
#include<stdio.h>
#include<stdlib.h>
void main()
{
    int a;
    char c;
    a = 10;
    c = a;
}
```

Sample Output 3:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
void is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
a is a identifier
; is a special symbol
char is a keyword
c is a identifier
; is a special symbol
a is a identifier
= is an operator
a
10 is a constant
; is a special symbol
c is a identifier
= is an operator
c
a is a identifier
; is a special symbol
a
ERROR: Type Mismatch.
================================================================================
Parsing Failed at line no: 9
```

Sample Program 4:

```c
#include<stdio.h>
#include<stdlib.h>

int main(void a, void b)
{
    printf("Hello, World!")
}
```

Sample Output 4:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
int is a keyword
main is a identifier
( is a special symbol
void is a keyword
ERROR: void type parameter!
```

Sample Program 5:

```c
#include<stdio.h>
#include<stdlib.h>

void func()
{
    int dfa =10;
}
int main(int argc)
{
    int x = dfa;
    printf("Value of x = %d", dfa);
}
```

Sample Output 5:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
void is a keyword
func is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
dfa is a identifier
= is an operator
10 is a constant
; is a special symbol
} is a special symbol
int is a keyword
main is a identifier
( is a special symbol
int is a keyword
argc is a identifier
) is a special symbol
{ is a special symbol
int is a keyword
x is a identifier
= is an operator
dfa is a identifier
; is a special symbol
dfa
LINE: 10 ERROR: Undeclared Variable.
```

Sample Program 6:

```c
#include<stdio.h>
#include<stdlib.h>

int square(int n)
{
    return n*n;
}
int main()
{
    square(2, 3);
    return 0;
}
```

Sample Output 6:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
int is a keyword
square is a identifier
( is a special symbol
int is a keyword
n is a identifier
) is a special symbol
{ is a special symbol
return is a keyword
n is a identifier
* is an operator
n
n is a identifier
; is a special symbol
n
} is a special symbol
int is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
square is a identifier
( is a special symbol
2 is a constant
, is a special symbol
3 is a constant
) is a special symbol
LINE: 10 ERROR: Arguments mismatch
```

Sample Program 7:

```c
#include<stdio.h>
#include<stdlib.h>

void cube(int n)
{
    return n*n*n;
}
int main()
{
    x = cube(4);
    return x;
}
```

Sample Output 7:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
void is a keyword
cube is a identifier
( is a special symbol
int is a keyword
n is a identifier
) is a special symbol
{ is a special symbol
return is a keyword
n is a identifier
* is an operator
n
n is a identifier
* is an operator
n
n is a identifier
; is a special symbol
n
ERROR: RETURN Type mismatch!
================================================================================
Parsing Failed at line no: 6
```

Sample Program 8:

```c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int x = 10;
    int twice = 2 *x;
    int x = x + 10;
    return x;
}
```

Sample Output 8:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
int is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
x is a identifier
= is an operator
10 is a constant
; is a special symbol
int is a keyword
twice is a identifier
= is an operator
2 is a constant
* is an operator
x is a identifier
; is a special symbol
x
int is a keyword
x is a identifier
LINE: 8 ERROR: DUPLICATE declaration
```

Sample Program 9:

```c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int a[-1] = {};
    printf("Wrong Array Size");
}
```

Sample Output 9:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
int is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
int is a keyword
a is a identifier
[ is a special symbol
- is an operator
1 is a constant
] is a special symbol
= is an operator
{ is a special symbol
} is a special symbol
================================================================================
Parsing Failed at line no: 6
```

Sample Program 10:

```c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    char x = 'a';
    x = x--;
    printf("Unary Operations Error")
}
```

Sample Output 10:

```
tarun@DESKTOP-2J4SUBH:~/CompilerDesign/SemanticPhase$ ./parser sample.c
#include<stdio.h> is a header declaration
#include<stdlib.h> is a header declaration
int is a keyword
main is a identifier
( is a special symbol
) is a special symbol
{ is a special symbol
char is a keyword
x is a identifier
= is an operator
 'a' is a constant
; is a special symbol
x is a identifier
= is an operator
x
x is a identifier
-- is an operator
x
ERROR: Type Mismatch.
================================================================================
Parsing Failed at line no: 7
Error: --
```

IMPLEMENTATION

We have added some new functions into our code to increase the functionality provided by the compiler. Scope attributes, function attributes, type checking are some of the functionalities taken care of in this phase of the complier design. The following are the names of the functions included for the implementation of this phase of the compiler design.

1. func_table: This is a structure definition that exclusively stores functions and their properties like number and type of parameters.
2. get_identifier_type: This function returns the type of the identifier given with their respective function calls if any.
3. insert_symbol_table_scope: This function adds a scope with respect to the number of opening and closing braces it has encountered until that part of the code and is put into an array which stores the scope of all the symbols.
4. remove_scope: This function removes the identifier scope data after it encounters its end of scope.
5. insert_func_table: This function inserts the function and its associated attributes into the function table structure.
6. insert_symbol_table_params_cnt: This function adds the count of parameters associated with the function into the function table structure.
7. verify_funccall_cnt: This function is called to verify if the number of parameters are same in both function call and the function definition.
8. insert_arg_type: This function adds the datatype of the function parameters into the function table structure.
9. check_arg_type: This function checks the argument's datatype in function call and function definition.
10. print_func_table: This function prints the function table.

ERROR HANDLING

In this phase of the compiler design, the program can detect the following errors:

1. Array size to be a positive integer.
2. Type checking in expressions.
3. Duplicate declaration of an identifier in the same scope.
4. Usage of identifier outside the defined scope.
5. Mismatch of function parameters in function definition or function call.
6. Mathematical operations performed on Non-Integer datatypes.
7. Function calls without its definition.
8. Void types parameters in function definition
9. Return datatype mismatch in function definition and function call.

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