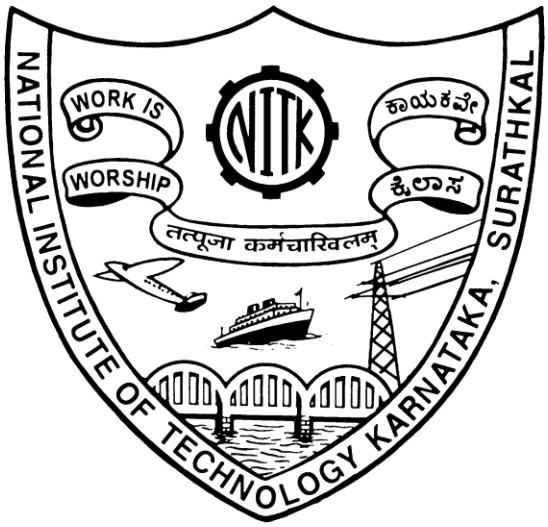
Semantic Analysis for the C Language



National Institute of Technology Karnataka Surathkal

Date: 21/10/2020

Submitted To: Dr. P. Santhi Thilagam

Submitted By:
1. Pranav Vigneshwar Kumar 181CO239
2. Mohammed Rushad 181CO232
3. Ankush Chandrashekar 181CO206
4. Akshat Nambiar 181CO204

Abstract

A compiler is a special program that processes statements written in a particular programming language (high-level language) and turns them into machine language (low-level language) that a computer's processors use. Apart from this, the compiler is also responsible for detecting and reporting any errors in the source program during the translation process.

The file used for writing code in a specific language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

This report specifies the details related to the third stage of the compiler, the parsing stage. We have developed a parser for the C programming language using the lex and yacc tools. The parser makes use of the tokens outputted by the lexer developed in the previous stage to parse the C input file. The lexical analyzer can detect only lexical errors like unmatched comments etc. but cannot detect syntactical errors like missing semi-colon etc. These syntactical errors are identified by the parser i.e. the syntax analysis phase is done by the parser. After parser checks if the code is structured correctly, semantic analysis phase checks if that syntax structure constructed in the source program derives any meaning. The output of the syntax analysis phase is parse tree whereas that of Semantic phase is annotated parse tree. Semantic analysis is done by modifications in the parser code only. Some of the tasks performed during semantic analysis are:

1. Scope Resolution
2. Type Checking
3. Array Bounds Checking

Contents Page No

Introduction

Semantics Analysis

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. Semantic analysis can compare information in one part of a parse tree to that in another part (e.g compare reference to variable agrees with its declaration, or that parameters to a function call match the function definition). Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures.

Semantic analysis typically involves in following tasks:
1. Data types are used correctly according to their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
2. Labels should exist
3. When an array is declared, subscript should be defined properly.

Some of the semantic errors that a general compiler is expected to recognize are given below
1. Type mismatch
    a. Type mismatch of variables
    b. Operands in operations having different types
2. Undeclared variable
    a. Check if variable is undeclared globally.
    b. Check if variable is visible in current scope.
3. Reserved identifier misuse.
    a. Function name and variable name cannot be same.
    b. Declaration of keyword as variable name.
4. Multiple declaration of variable in a scope.
5. Accessing an out of scope variable.
6. Actual and formal parameter mismatch.

Yacc Script

Yacc stands for Yet Another Compiler-Compiler. Yacc is essentially a parser generator. Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these

rules are recognized, and a low-level routine to do the basic input. A function is then generated by Yacc to control the input process. This function is called the parser which calls the lexical analyzer to get a stream of tokens from the input.

Based on the input structure rules, called grammar rules, the tokens are organized. When one of these rules has been recognized, then user code supplied for this rule, an action, is invoked. Actions have the ability to return values and make use of the values of other actions.

Yacc is written in portable C. The class of specifications accepted is a very general one, LALR(1) grammars with disambiguating rules.

The structure of our yacc script is divided into three sections, separated by lines that contain only two percent signs, as follows:

DECLARATIONS

%%

RULES

%%

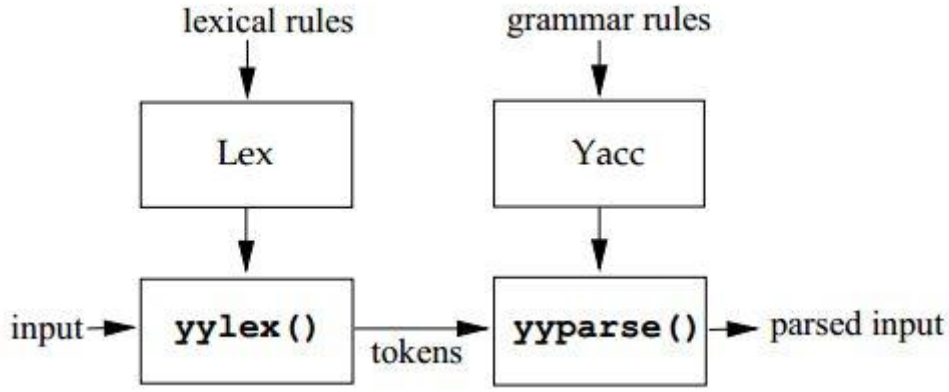
AUXILIARY FUNCTIONS

The **Declarations Section** defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied directly into the generated source file. We also define all parameters related to the parser here, specifications like using leftmost derivations or rightmost derivations, precedence, left and right associativity are declared here, data types and tokens which will be used by the lexical analyzer are also declared at this stage.

The **Rules Section** contains the entire grammar which is used for deciding if the input text is legally correct according to the specifications of the language. Yacc uses these rules for reducing the token stream received from the lexical analysis stage. All rules are linked to each other from the start state.

Yacc generates C code for the rules specified in the Rules section and places this code into a single function called yyparse(). The **Auxiliary Functions Section** contains C statements and functions that are copied directly to the generated source

file. These statements usually contain code called by the different rules. This section essentially allows the programmer to add to the generated source code.



C Program

The parser takes C source files as input for parsing. The input file is specified in the auxiliary functions section of the yacc script.
The workflow for testing the parser is as follows:

1. Compile the yacc script using the yacc tool
   $ yacc -d parser.y
2. Compile the flex script using the flex tool
   $ lex lexer.l
3. The first two steps generate lex.yy.c, y.tab.c, and y.tab.h. The header file is included in lexer.l file. Then, lex.yy.c and y.tab.c are compiled together.
   $ gcc lex.yy.c y.tab.c
4. Run the generated executable file
   $ ./a.out

Design of Programs

Updated Lexical Analyzer Code

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

    struct ConstantTable{
        char constant_name[100];
        char constant_type[100];
        int exist;
    }CT[1000];

    struct SymbolTable{
        char symbol_name[100];
        char symbol_type[100];
        char array_dimensions[100];
        char class[100];
        char value[100];
        char parameters[100];
        int line_number;
        int exist;
        int nested_val;
        int params_count;
    }ST[1000];

    int current_nested_val = 0;
    int params_count = 0;


    unsigned long hash(unsigned char *str)
    {
        unsigned long hash = 5381;
        int c;

        while (c = *str++)
            hash = ((hash << 5) + hash) + c;

        return hash;
    }

```

```
40      int search_ConstantTable(char* str){
41          unsigned long temp_val = hash(str);
42          int val = temp_val%1000;
43
44          if(CT[val].exist == 0){
45              return 0;
46          }
47
48          else if(strcmp(CT[val].constant_name, str) == 0)
49          {
50              return 1;
51          }
52          else
53          {
54              for(int i = val+1 ; i!=val ; i = (i+1)%1000)
55              {
56                  if(strcmp(CT[i].constant_name,str)==0)
57                  {
58                      return 1;
59                  }
60              }
61              return 0;
62          }
63      }
64
65
66      int search_SymbolTable(char* str){
67          unsigned long temp_val = hash(str);
68          int val = temp_val%1000;
69
70          if(ST[val].exist == 0){
71              return 0;
72          }
73
74          else if(strcmp(ST[val].symbol_name, str) == 0)
75          {
76              return val;
```

```
77          }
78          else
79          {
80              for(int i = val+1 ; i!=val ; i = (i+1)%1000)
81              {
82                  if(strcmp(ST[i].symbol_name,str)==0)
83                  {
84                      return i;
85                  }
86              }
87              return 0;
88          }
89      }
90
91
92      void insert_ConstantTable(char* name, char* type){
93          int index = 0;
94           if(search_ConstantTable(name)){
95              return;
96          }
97          else{
98              unsigned long temp_val = hash(name);
99              int val = temp_val%1000;
100             if(CT[val].exist == 0){
101                 strcpy(CT[val].constant_name, name);
102                 strcpy(CT[val].constant_type, type);
103                 CT[val].exist = 1;
104                 return;
105             }
106
107             for(int i = val+1; i != val; i = (i+1)%1000){
108                 if(CT[i].exist == 0){
109                     index = i;
110                     break;
111                 }
112             }
113             strcpy(CT[index].constant_name, name);
114             strcpy(CT[index].constant_type, type);
```

```
            CT[index].exist = 1;
        }
    }

    void insert_SymbolTable(char* name, char* class){
        int index = 0;
        //printf("BBBB");
         if(search_SymbolTable(name)){
            //printf("AAAAAA");
            return;
        }
        else{
            unsigned long temp_val = hash(name);
            int val = temp_val%1000;
            if(ST[val].exist == 0){
                strcpy(ST[val].symbol_name, name);
                strcpy(ST[val].class, class);
                ST[val].nested_val = 100;
                //ST[val].params_count = -1;
                ST[val].line_number = yylineno;
                ST[val].exist = 1;
                return;
            }

            for(int i = val+1; i != val; i = (i+1)%1000){
                if(ST[i].exist == 0){
                    index = i;
                    break;
                }
            }
            strcpy(ST[index].symbol_name, name);
            strcpy(ST[val].class, class);
            ST[index].nested_val = 100;
            //ST[index].params_count = -1;
            ST[index].exist = 1;
        }
    }
```

```
153     void insert_SymbolTable_type(char *str1, char *str2)
154     {
155         for(int i = 0 ; i < 1000 ; i++)
156         {
157             if(strcmp(ST[i].symbol_name,str1)==0)
158             {
159                 strcpy(ST[i].symbol_type,str2);
160             }
161         }
162     }
163 
164     void insert_SymbolTable_value(char *str1, char *str2)
165     {
166         for(int i = 0 ; i < 1001 ; i++)
167         {
168             if(strcmp(ST[i].symbol_name,str1)==0 && ST[i].nested_val != current_nested_val)
169             {
170                 strcpy(ST[i].value,str2);
171             }
172         }
173     }
174 
175     void insert_SymbolTable_arraydim(char *str1, char *dim)
176     {
177         for(int i = 0 ; i < 1000 ; i++)
178         {
179             if(strcmp(ST[i].symbol_name,str1)==0)
180             {
181                 strcpy(ST[i].array_dimensions,dim);
182             }
183         }
184     }
185 
186     void insert_SymbolTable_funcparam(char *str1, char *param)
187     {
188         for(int i = 0 ; i < 1000 ; i++)
189         {
```

```
190             if(strcmp(ST[i].symbol_name,str1)==0)
191             {
192                 strcat(ST[i].parameters," ");
193                 strcat(ST[i].parameters,param);
194             }
195         }
196     }
197 
198     void insert_SymbolTable_line(char *str1, int line)
199     {
200         for(int i = 0 ; i < 1000 ; i++)
201         {
202             if(strcmp(ST[i].symbol_name,str1)==0)
203             {
204                 ST[i].line_number = line;
205             }
206         }
207     }
208 
209     void insert_SymbolTable_nest(char *s, int nest)
210     {
211         //printf("mlkjhad %d", nest);
212         if(search_SymbolTable(s) && ST[search_SymbolTable(s)].nested_val != 100)
213         {
214             //printf("mlkjhad %d\n", nest);
215             int pos = 0;
216             int value = hash(s);
217             value = value%1001;
218             for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
219             {
220                 if(ST[i].exist == 0)
221                 {
222                     pos = i;
223                     break;
224                 }
225             }
226 
```

```
227             strcpy(ST[pos].symbol_name,s);
228             strcpy(ST[pos].class,"Identifier");
229             ST[pos].nested_val = nest;
230             //printf("afafa %s\n", ST[pos].symbol_name);
231             //ST[pos].params_count = -1;
232             ST[pos].line_number = yylineno;
233             ST[pos].exist = 1;
234         }
235         else
236         {
237             for(int i = 0 ; i < 1001 ; i++)
238             {
239                 if(strcmp(ST[i].symbol_name,s)==0 )
240                 {
241                     ST[i].nested_val = nest;
242                 }
243             }
244         }
245     }
246
247     int check_scope(char *s)
248     {
249         int flag = 0;
250         for(int i = 0 ; i < 1000 ; i++)
251         {
252             if(strcmp(ST[i].symbol_name,s)==0)
253             {
254                 if(ST[i].nested_val > current_nested_val)
255                 {
256                     flag = 1;
257                 }
258                 else
259                 {
260                     flag = 0;
261                     break;
262                 }
263             }
264         }
```

```
265         if(!flag)
266         {
267             return 1;
268         }
269         else
270         {
271             return 0;
272         }
273     }
274
275     void remove_scope (int nesting)
276     {
277         for(int i = 0 ; i < 1000 ; i++)
278         {
279             if(ST[i].nested_val == nesting)
280             {
281                 ST[i].nested_val = 100;
282             }
283         }
284     }
285
286     void insert_SymbolTable_function(char *s)
287     {
288         for(int i = 0 ; i < 1001 ; i++)
289         {
290             if(strcmp(ST[i].symbol_name,s)==0 )
291             {
292                 strcpy(ST[i].class,"Function");
293                 return;
294             }
295         }
296
297     }
298
299     int check_function(char *s)
300     {
301         for(int i = 0 ; i < 1000 ; i++)
302         {
```

```
303             if(strcmp(ST[i].symbol_name,s)==0)
304             {
305                 if(strcmp(ST[i].class,"Function")==0)
306                     return 1;
307             }
308         }
309         return 0;
310     }
311
312     int check_array(char *s)
313     {
314         for(int i = 0 ; i < 1000 ; i++)
315         {
316             if(strcmp(ST[i].symbol_name,s)==0)
317             {
318                 if(strcmp(ST[i].class,"Array Identifier")==0)
319                 {
320                     return 0;
321                 }
322             }
323         }
324         return 1;
325     }
326
327     int duplicate(char *s)
328     {
329         for(int i = 0 ; i < 1000 ; i++)
330         {
331             if(strcmp(ST[i].symbol_name,s)==0)
332             {
333                 if(ST[i].nested_val == current_nested_val)
334                 {
335                     return 1;
336                 }
337             }
338         }
339
340         return 0;
```

```
343     int check_duplicate(char* str)
344     {
345         for(int i=0; i<1000; i++)
346         {
347             if(strcmp(ST[i].symbol_name, str) == 0 && strcmp(ST[i].class, "Function") == 0)
348             {
349                 printf("ERROR: Cannot Redeclare same function!\n");
350                 printf("\nUNSUCCESSFUL: INVALID PARSE\n");
351                 exit(0);
352             }
353         }
354     }
355
356     int check_declaration(char* str, char *check_type)
357     {
358         for(int i=0; i<1000; i++)
359         {
360             if(strcmp(ST[i].symbol_name, str) == 0 && strcmp(ST[i].class, "Function") == 0
361             {
362                 return 1;
363             }
364         }
365         return 0;
366     }
367
368     int check_params(char* type_specifier)
369     {
370         if(!strcmp(type_specifier, "void"))
371         {
372             printf("ERROR: Here, Parameter cannot be of void type\n");
373             printf("\nUNSUCCESSFUL: INVALID PARSE\n");
374             exit(0);
375         }
376         return 0;
377     }
378
```

```
379     void insert_SymbolTable_paramscount(char *s, int count)
380     {
381         for(int i = 0 ; i < 1000 ; i++)
382         {
383             if(strcmp(ST[i].symbol_name,s)==0 )
384             {
385                 ST[i].params_count = count;
386             }
387         }
388     }
389
390     int getSTparamscount(char *s)
391     {
392         for(int i = 0 ; i < 1000 ; i++)
393         {
394             if(strcmp(ST[i].symbol_name,s)==0 )
395             {
396                 return ST[i].params_count;
397             }
398         }
399         return -2;
400     }
401
402     char gettype(char *s, int flag)
403     {
404             for(int i = 0 ; i < 1001 ; i++ )
405             {
406                 if(strcmp(ST[i].symbol_name,s)==0)
407                 {
408                     return ST[i].symbol_type[0];
409                 }
410             }
411
412     }
413
414     void printConstantTable(){
415         printf("%20s | %20s\n", "CONSTANT","TYPE");
```

```
416         for(int i = 0; i < 1000; ++i){
417             if(CT[i].exist == 0)
418                 continue;
419 
420             printf("%20s | %20s\n", CT[i].constant_name, CT[i].constant_type);
421         }
422     }
423 
424     void printSymbolTable(){
425         printf("%10s | %18s | %10s | %10s | %10s | %10s | %10s | %10s\n"
426         for(int i = 0; i < 1000; ++i){
427             if(ST[i].exist == 0)
428                 continue;
429             printf("%10s | %18s | %10s | %10s | %10s | %10s | %15d | %10d | %d\
430         }
431     }
432     char current_identifier[20];
433     char current_type[20];
434     char current_value[20];
435     char current_function[20];
436     char previous_operator[20];
437     int flag;
438 
439 %}
440 
441 num                 [0-9]
442 alpha               [a-zA-Z]
443 alphanum            {alpha}|{num}
444 escape_sequences    0|a|b|f|n|r|t|v|"\\"|"\""|"\'"
445 ws                  [ \t\r\f\v]+
446 %x MLCOMMENT
447 DE "define"
448 IN "include"
449 
450 %%
```

```
"char"          { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword"); return CHAR;}
"double"        { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword"); return DOUBLE;}
"else"          { insert_SymbolTable_line(yytext, yylineno); insert_SymbolTable(yytext, "Keyword"); return ELSE;}
"float"         { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");return FLOAT;}
"while"         { insert_SymbolTable(yytext, "Keyword"); return WHILE;}
"do"            { insert_SymbolTable(yytext, "Keyword"); return DO;}
"for"           { insert_SymbolTable(yytext, "Keyword"); return FOR;}
"if"            { insert_SymbolTable(yytext, "Keyword"); return IF;}
"int"           { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");return INT;}
"long"          { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return LONG;}
"return"        { insert_SymbolTable(yytext, "Keyword");  return RETURN;}
"short"         { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return SHORT;}
"signed"        { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return SIGNED;}
"sizeof"        { insert_SymbolTable(yytext, "Keyword");  return SIZEOF;}
"struct"        { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return STRUCT;}
"unsigned"      { insert_SymbolTable(yytext, "Keyword");  return UNSIGNED;}
"void"          { strcpy(current_type,yytext); insert_SymbolTable(yytext, "Keyword");  return VOID;}
"break"         { insert_SymbolTable(yytext, "Keyword");  return BREAK;}
"continue"      { insert_SymbolTable(yytext, "Keyword");  return CONTINUE;}
"goto"          { insert_SymbolTable(yytext, "Keyword");  return GOTO;}
"switch"        { insert_SymbolTable(yytext, "Keyword");  return SWITCH;}
"case"          { insert_SymbolTable(yytext, "Keyword");  return CASE;}
"default"       { insert_SymbolTable(yytext, "Keyword");  return DEFAULT;}

("\"")[^\n\"]*("\"")                        {strcpy(current_value,yytext); insert_ConstantTable(yytext,"String Constant"); return string_constant;}
("\"")[^\n\"]*                              { printf("Line No. %d ERROR: UNCLOSED STRING - %s\n", yylineno, yytext); return 0;}
("\'")((\"({escape_sequences}))|.)("\'")    {strcpy(current_value,yytext); insert_ConstantTable(yytext,"Character Constant"); return character_cons
("\'")((((\")[^0abfnrtv\\\"\'][^\n\']*))|[^\n\'][^\n\'']+)("\'") {printf("Line No. %d ERROR: NOT A CHARACTER - %s\n", yylineno, yytext); return 0
{num}+(\.{num}+)?e{num}+                {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Floating Constant"); return float_constant;}
{num}+\.{num}+                          {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Floating Constant"); return float_constant;}
{num}+                                  {strcpy(current_value,yytext); insert_ConstantTable(yytext, "Number Constant"); yylval = atoi(yytext); retur
(_|{alpha})({alpha}|{alpha}|_)*         {strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Identifier");  return identifier;}
(_|{alpha})({alpha}|{alpha}|_)*/\[      {strcpy(current_identifier,yytext);insert_SymbolTable(yytext,"Array Identifier");  return array_identifier;}
{ws}                                        ;

"+"                                     {return *yytext;}
"-"                                     {return *yytext;}
```

```
"+"                                     {return *yytext;}
"-"                                     {return *yytext;}
"*"                                     {return *yytext;}
"/"                                     {return *yytext;}
"="                                     {return *yytext;}
"%"                                     {return *yytext;}
"&"                                     {return *yytext; }
"^"                                     {return *yytext; }
"++"                                    {return INCREMENT;}
"--"                                    {return DECREMENT;}
"!"                                     {return NOT;}
"+="                                    {return ADD_EQUAL;}
"-="                                    {return SUBTRACT_EQUAL;}
"*="                                    {return MULTIPLY_EQUAL;}
"/="                                    {return DIVIDE_EQUAL;}
"%="                                    {return MOD_EQUAL;}
"&&"                                    {return AND_AND;}



"||"                                    {return OR_OR;}
">"                                     {return GREAT;}
"<"                                     {return LESS;}
">="                                    {return GREAT_EQUAL;}
"<="                                    {return LESS_EQUAL;}
"=="                                    {return EQUAL;}
"!="                                    {return NOT_EQUAL;}
.                                       { flag = 1;
                                          if(yytext[0] == '#')
                                              printf("Line No. %d PREPROCESSOR ERROR - %s\n", yylineno, yytext);
                                          else
                                              printf("Line No. %d ERROR: ILLEGAL CHARACTER - %s\n", yylineno, yytext);
                                         return 0;}

%%
```

Updated Parser Code

```
%{
    void yyerror(char* s);
    int yylex();
    #include "stdio.h"
    #include "stdlib.h"
    #include "ctype.h"
    #include "string.h"
    void insert_type();
    void insert_value();
    void insert_dimensions();
    void insert_parameters();
    void remove_scope (int );
    int check_scope(char*);
    int check_function(char *);
    void insert_SymbolTable_nest(char*, int);
    void insert_SymbolTable_paramscount(char*, int);
    int getSTparamscount(char*);
    int check_duplicate(char*);
    int check_declaration(char*, char *);
    int check_params(char*);
    int duplicate(char *s);
    int check_array(char*);
    void insert_SymbolTable_function(char*);
    char gettype(char*,int);

    extern int flag=0;
    int insert_flag = 0;

    extern char current_identifier[20];
    extern char current_type[20];
    extern char current_value[20];
    extern char current_function[20];
    extern char previous_operator[20];
    extern int current_nested_val;
    char currfunctype[100];
    char currfunccall[100];
    extern int params_count;
    int call_params_count;
```

```
42  %nonassoc IF
43  %token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
44  %token RETURN MAIN
45  %token VOID
46  %token WHILE FOR DO
47  %token BREAK CONTINUE GOTO
48  %token ENDIF
49  %token SWITCH CASE DEFAULT
50  %expect 2
51
52  %token identifier array_identifier
53  %token integer_constant string_constant float_constant character_constant
54
55  %nonassoc ELSE
56
57  %right MOD_EQUAL
58  %right MULTIPLY_EQUAL DIVIDE_EQUAL
59  %right ADD_EQUAL SUBTRACT_EQUAL
60  %right '='
61
62  %left OR_OR
63  %left AND_AND
64  %left '^'
65  %left EQUAL NOT_EQUAL
66  %left LESS_EQUAL LESS GREAT_EQUAL GREAT
67  %left '+' '-'
68  %left '*' '/' '%'
69
70  %right SIZEOF
71  %right NOT
72  %left INCREMENT DECREMENT
73
74
75  %start begin_parse
76
77  %%
```

```
begin_parse
            : declarations;

declarations
            : declaration declarations
            |
            ;

declaration
            : variable_dec
            | function_dec
            | structure_dec;

structure_dec
            : STRUCT identifier { insert_type(); } '{' structure_content  '}' ';';

structure_content : variable_dec structure_content | ;

variable_dec
            : datatype variables ';'
            | structure_initialize;

structure_initialize
            : STRUCT identifier variables;

variables
            : identifier_name multiple_variables;

multiple_variables
            : ',' variables
            | ;

identifier_name
            : identifier { if(check_function(current_identifier))
                           {yyerror("ERROR: Identifier cannot be same as function name!\n"); exit(8);}
                           if(duplicate(current_identifier)){yyerror("Duplicate value!\n");exit(0);}insert_SymbolTable_nest(curre
            | array_identifier {if(duplicate(current_identifier)){yyerror("Duplicate value!\n");exit(0);}insert_SymbolTable_nest
extended_identifier : array_iden | '='{strcpy(previous_operator,"=");} simple_expression ;

array_iden
            : '[' array_dims
            | ;

array_dims
            : integer_constant {insert_dimensions();} ']' initilization{if($$ < 1) {yyerror("Array must have size greater than 1!\n"); exit(0);} }
            | ']' string_initilization;

initilization
            : string_initilization
            | array_initialization
            | ;

string_initilization
            : '='{strcpy(previous_operator,"=");} string_constant { insert_value(); };

array_initialization
            : '='{strcpy(previous_operator,"=");} '{' array_values '}';

array_values
            : integer_constant multiple_array_values;

multiple_array_values
            : ',' array_values
            | ;


datatype
            : INT | CHAR | FLOAT | 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 | ;

function_dec
            : function_datatype function_parameters;

function_datatype
            : datatype identifier '('  {strcpy(currfunctype, current_type); check_duplicate(current_identifier); insert_SymbolTable_function(current

function_parameters
            : parameters ')' statement;

parameters
            : datatype { check_params(current_type); } all_parameter_identifiers {insert_SymbolTable_paramscount(current_function, params_count);} |

all_parameter_identifiers
            : parameter_identifier multiple_parameters;

multiple_parameters
            : ',' parameters
            | ;

parameter_identifier
            : identifier {insert_parameters(); insert_type(); insert_SymbolTable_nest(current_identifier,1); params_count++;} extended_parameter;

extended_parameter
            : '[' ']'
            | ;
statement
            : expression_statment | multiple_statement
            | conditional_statements | iterative_statements
            | return_statement | break_statement
            | variable_dec;

multiple_statement
            :{current_nested_val++;} '{' statments '}' {remove_scope(current_nested_val);current_nested_val--;} ;

statments
            : statement statments
            | ;

expression_statment
            : expression ';'
            | ';' ;

conditional_statements
            : IF '(' simple_expression ')' {if($3!=1){yyerror("ERROR: Here, condition must have integer value!\n");exit(0);}} statement extended_con

extended_conditional_statements
            : ELSE statement
            | ;

iterative_statements
            : WHILE '(' simple_expression ')'{if($3!=1){printf("ERROR: Here, condition must have integer value!\n");exit(0);}} statement
            | FOR '(' for_initialization simple_expression ';' {if($5!=1){printf("Here, condition must have integer value!\n");exit(0);}} expression
            | DO statement WHILE '(' simple_expression ')' {if($5!=1){printf("ERROR: Here, condition must have integer value!\n");exit(0);}} ';';

for_initialization
            : variable_dec
            | expression ';'
            | ';' ;

return_statement
            : RETURN ';' {if(strcmp(currfunctype,"void")) {yyerror("ERROR: Cannot have void return for non-void function!\n"); exit(0);}}
            | RETURN expression ';' {   if(!strcmp(currfunctype, "void"))
```

```
                                    {
                                        yyerror("Non-void return for void function!"); exit(0);
                                    }

                                    if((currfunctype[0]=='i' || currfunctype[0]=='c') && $2!=1)
                                    {
                                        yyerror("Expression doesn't match return type of function\n"); exit(0);
                                    }

                                };

break_statement
            : BREAK ';' ;


expression
            : mutable '=' expression             {                strcpy(previous_operator,"=");
                                                                    if($1==1 && $3==1)
                                                                    {
                                                                    $$=1;
                                                                    }
                                                                    else
                                                                    {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                  }
            | mutable ADD_EQUAL expression     {                 strcpy(previous_operator,"+=");
                                                                    if($1==1 && $3==1)
                                                                    $$=1;
                                                                    else
                                                                    {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                  }
            | mutable SUBTRACT_EQUAL expression  {                 strcpy(previous_operator,"-=");
                                                                    if($1==1 && $3==1)
                                                                    $$=1;
                                                                    else
                                                                    {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                  }
              | mutable MULTIPLY_EQUAL expression {                    strcpy(previous_operator,"*=");
                                                                        if($1==1 && $3==1)
                                                                        $$=1;
                                                                        else
                                                                        {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                      }
              | mutable DIVIDE_EQUAL expression       {                strcpy(previous_operator,"/=");
                                                                        if($1==1 && $3==1)
                                                                        $$=1;
                                                                        else
                                                                        {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                      }
              | mutable MOD_EQUAL expression      {                    strcpy(previous_operator,"%=");
                                                                        if($1==1 && $3==1)
                                                                        $$=1;
                                                                        else
                                                                        {$$=-1; yyerror("Type Mismatch\n"); exit(0);}
                                                                      }

              | mutable INCREMENT                                   {if($1 == 1) $$=1; else $$=-1;}
              | mutable DECREMENT                                   {if($1 == 1) $$=1; else $$=-1;}
              | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;


simple_expression
            : simple_expression OR_OR and_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;}
            | and_expression {if($1 == 1) $$=1; else $$=-1;};

and_expression
            : and_expression AND_AND unary_relation_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;}
              |unary_relation_expression {if($1 == 1) $$=1; else $$=-1;} ;

unary_relation_expression
            : NOT unary_relation_expression {if($2==1) $$=1; else $$=-1;}
            | regular_expression {if($1 == 1) $$=1; else $$=-1;} ;

```

```
regular_expression
            : regular_expression relational_operators sum_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;}
            | sum_expression {if($1 == 1) $$=1; else $$=-1;} ;

relational_operators
            : GREAT_EQUAL{strcpy(previous_operator,">=");}
            | LESS_EQUAL{strcpy(previous_operator,"<=");}
            | GREAT{strcpy(previous_operator,">");}
            | LESS{strcpy(previous_operator,"<");}
            | EQUAL{strcpy(previous_operator,"==");}
            | NOT_EQUAL{strcpy(previous_operator,"!=");} ;

sum_expression
            : sum_expression sum_operators term  {if($1 == 1 && $3==1) $$=1; else $$=-1;}
            | term {if($1 == 1) $$=1; else $$=-1;};

sum_operators
            : '+'
            | '-' ;

term
            : term MULOP factor {if($1 == 1 && $3==1) $$=1; else $$=-1;}
            | factor {if($1 == 1) $$=1; else $$=-1;} ;

MULOP
            : '*' | '/' | '%' ;

factor
            : immutable {if($1 == 1) $$=1; else $$=-1;}
            | mutable {if($1 == 1) $$=1; else $$=-1;} ;

mutable
            : identifier {
                          if(!check_scope(current_identifier))
                          {printf("%s\n",current_identifier);yyerror("Identifier undeclared\n");exit(0);}
                          if(!check_array(current_identifier))
                          {printf("%s\n",current_identifier);yyerror("Array Identifier has No Subscript\n");exit(0);}
                         if(gettype(current_identifier,0)=='i' || gettype(current_identifier,1)== 'c')
                         $$ = 1;
                         else
                         $$ = -1;
                         }
            | array_identifier {if(!check_scope(current_identifier)){printf("%s\n",current_identifier);yyerror("Identifier undeclared\n");exit(0);}}
                          {if(gettype(current_identifier,0)=='i' || gettype(current_identifier,1)== 'c')
                           $$ = 1;
                           else
                           $$ = -1;
                           };

immutable
           : '(' expression ')' {if($2==1) $$=1; else $$=-1;}
           | call
           | constant {if($1==1) $$=1; else $$=-1;};

call
           : identifier '('{ strcpy(previous_operator,"(");
                      if(!check_declaration(current_identifier, "Function"))
                      { yyerror("Function not declared"); exit(0);}
                      insert_SymbolTable_function(current_identifier);
                      strcpy(currfunccall,current_identifier);
                      } arguments ')'
                      { if(strcmp(currfunccall,"printf"))
                        {
                            if(getSTparamscount(currfunccall)!=call_params_count)
                            {
                                yyerror("Number of parameters not same as number of arguments during function call!");
                                //printf("Number of arguments in function call %s doesn't match number of parameters\n", currfunccall);
                                exit(8);
                            }
                        }
                      };

arguments
           : arguments_list | ;
```

```
376   arguments_list
377               : expression { call_params_count++; } A ;
378
379   A
380               : ',' expression { call_params_count++; } A
381               | ;
382
383   constant
384               : integer_constant  {  insert_type(); $$=1; }
385               | string_constant   {  insert_type(); $$=-1;}
386               | float_constant    {  insert_type(); }
387               | character_constant{  insert_type();$$=1; };
388
389
390   %%
391
392   extern FILE *yyin;
393   extern int yylineno;
394   extern char *yytext;
395   void insert_SymbolTable_type(char *,char *);
396   void insert_SymbolTable_value(char *, char *);
397   void insert_ConstantTable(char *, char *);
398   void insert_SymbolTable_arraydim(char *, char *);
399   void insert_SymbolTable_funcparam(char *, char *);
400   void printSymbolTable();
401   void printConstantTable();
402
403
404   int main()
405   {
406       yyin = fopen("test21.c", "r");
407       yyparse();
408
409       if(flag == 0)
410       {
411           printf("VALID PARSE\n");
412           printf("%30s SYMBOL TABLE \n", " ");
```

```
414             printSymbolTable();
415
416             printf("\n\n%30s CONSTANT TABLE \n", " ");
417             printf("%30s %s\n", " ", "--------------");
418             printConstantTable();
419         }
420     }
421
422     void yyerror(char *s)
423     {
424         printf("Line No. : %d %s %s\n",yylineno, s, yytext);
425         flag=1;
426         printf("\nUNSUCCESSFUL: INVALID PARSE\n");
427     }
428
429     void insert_type()
430     {
431         insert_SymbolTable_type(current_identifier,current_type);
432     }
433
434     void insert_value()
435     {
436         if(strcmp(previous_operator, "=") == 0)
437         {   insert_SymbolTable_value(current_identifier,current_value);
438         }
439     }
440
441     void insert_dimensions()
442     {
443         insert_SymbolTable_arraydim(current_identifier, current_value);
444     }
445
446     void insert_parameters()
447     {
448         insert_SymbolTable_funcparam(current_function, current_identifier);
449     }
```

Explanation

The lex code is used to detect tokens and generate a stream of tokens from the input C source code. In the first phase of the project, we only stored the different symbols and constants their respective tables and printed out the different tokens with their corresponding line numbers. For the second stage, we return the tokens identified by the lexer to the parser so that the parser is able to use it for further computation. In addition to the functions used in the previous stage, we added functions to help the parser insert the type, value, function parameter, and array dimensions into the symbol table. In the 3rd stage we are using the symbol table and constant table of the previous phase only. We added functions to insert the nesting value of a variable which is essential to check for the scope of a variable, to insert the number of function parameters, to check the scope matches that of the variable etc., in order to check the semantics . In the production rules of the grammar semantic actions are written and these are performed by the functions listed above.

Definition Section

In the definition section of the yacc program, we include all the required header files, function definitions and other variables. All the tokens which are returned by the lexical analyzer are also listed in the order of their precedence in this section. Operators are also declared here according to their associativity and precedence. This helps ensure that the grammar given to the parser is unambiguous.

Rules Section

In this section, grammar rules for the C Programming Language is written. The grammar rules are written in such a way that there is no left recursion and the grammar is also deterministic. Non deterministic grammar is converted by applying left factoring. The grammar productions does the syntax analysis of the source code. When the complete statement with proper syntax is matched by the parser, the parser recognizes that it is a valid parse and prints the symbol and constant table. If the statement is not matched, the parser recognizes that there is an error and outputs the error along with the line number.

C Code Section

The yyparse() function was called to run the program on the given input file. After that, both the symbol table and the constant table were printed in order to show the result.

Test Cases (Error Free)

Test Case 1

```c
//ERROR FREE - This test includes a declaration and a print statement
#include<stdio.h>

int main()
{
        //This is the first test program.
        int a;
        /* This is the declaration
        of an integer value */

        printf("Hello World");
        return 0;
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |        100 | 4
      main |          Function |        int |            |            |            |               0 |        100 | 4
    printf |          Function |        int |            |            |            |               0 |        100 | 10
         a |        Identifier |        int |            |            |            |               0 |        100 | 7
    return |           Keyword |            |            |            |            |               0 |        100 | 11


                            CONSTANT TABLE
                            --------------
          CONSTANT |                TYPE
      "Hello World" |      String Constant
                  0 |      Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 2

```c
//ERROR FREE - This test case includes a function
#include<stdio.h>

int multiply(int a)
{
        return 2*a;
}

int main()
{
        int a = 5;
        int b = multiply(a);
        printf("%d ", b);
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
  multiply |          Function |        int |            |            |          a |               1 |        100 | 4
       int |           Keyword |            |            |            |            |               0 |        100 | 4
      main |          Function |        int |            |            |            |               0 |        100 | 9
    printf |          Function |        int |            |            |            |               0 |        100 | 13
         a |        Identifier |        int |            |            |            |               0 |        100 | 4
         b |        Identifier |        int |            |            |            |               0 |        100 | 12
    return |           Keyword |            |            |            |            |               0 |        100 | 6


                            CONSTANT TABLE
                            --------------
          CONSTANT |                TYPE
                 2 |      Number Constant
                 5 |      Number Constant
              "%d " |      String Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 3

```c
//ERROR FREE - This test case includes array declarations, and conditional statements
#include<stdio.h>

int main()
{
	int A[5] = {1,2,3,4,5};
	char B[10] = "Hello";

	if(B[0] == 'H'){
		if(A[0] == '1')
			printf("Hello 1");
		else
			printf("Hello 2");
	}
	else
		printf("Not Hello");
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
      else |            Keyword |            |            |            |            |               0 |        100 | 15
       int |            Keyword |            |            |            |            |               0 |        100 | 4
      char |            Keyword |            |            |            |            |               0 |        100 | 7
      main |           Function |        int |            |            |            |               0 |        100 | 4
    printf |           Function |       char |            |            |            |               0 |        100 | 11
        if |            Keyword |            |            |            |            |               0 |        100 | 9
         A |   Array Identifier |       char |            |          5 |            |               0 |        100 | 6
         B |   Array Identifier |       char |            |         10 |            |               0 |        100 | 7


                            CONSTANT TABLE
                            --------------
          CONSTANT |                 TYPE
           "Hello" |      String Constant
               '1' |   Character Constant
       "Not Hello" |      String Constant
                10 |      Number Constant
                 0 |      Number Constant
                 1 |      Number Constant
                 2 |      Number Constant
                 3 |      Number Constant
                 4 |      Number Constant
                 5 |      Number Constant
               'H' |   Character Constant
         "Hello 1" |      String Constant
         "Hello 2" |      String Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 4

```c
//ERROR FREE - This test case includes for and while loops
#include<stdio.h>

int main()
{
	int num = 3;

	for(int i = 0; i < num; i++)
		printf("Hello");

	while(num > 0)
	{
		printf("Hello");
		num--;
	}
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |            Keyword |            |            |            |            |               0 |        100 | 4
      main |           Function |        int |            |            |            |               0 |        100 | 4
    printf |           Function |        int |            |            |            |               0 |        100 | 9
         i |         Identifier |        int |            |            |            |               0 |        100 | 8
       num |         Identifier |        int |            |            |            |               0 |        100 | 6
     while |            Keyword |            |            |            |            |               0 |        100 | 11
       for |            Keyword |            |            |            |            |               0 |        100 | 8


                            CONSTANT TABLE
                            --------------
          CONSTANT |                 TYPE
           "Hello" |      String Constant
                 0 |      Number Constant
                 3 |      Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 5

```c
//ERROR FREE - This test case includes nested loops
#include<stdio.h>

int main()
{
        int num = 3;

        for(int i = 0; i<num; i++)
        {
                for(int j = 0; j < num; j++)
                        printf("Hello");
        }
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |            Keyword |            |            |            |            |               0 |        100 | 4
      main |           Function |        int |            |            |            |               0 |        100 | 4
    printf |           Function |        int |            |            |            |               0 |        100 | 11
         i |         Identifier |        int |            |            |            |               0 |        100 | 8
         j |         Identifier |        int |            |            |            |               0 |        100 | 10
       num |         Identifier |        int |            |            |            |               0 |        100 | 6
       for |            Keyword |            |            |            |            |               0 |        100 | 8


                            CONSTANT TABLE
                            --------------
           CONSTANT |                TYPE
            "Hello" |      String Constant
                  0 |      Number Constant
                  3 |      Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 6

```c
//ERROR FREE - This test case includes declaration of a structure
#include<stdio.h>

struct book
{
        char name[10];
        char author[10];
};

int main()
{
        int num = 3;
        printf("Hello");

}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |            Keyword |            |            |            |            |               0 |        100 | 10
      char |            Keyword |            |            |            |            |               0 |        100 | 6
      main |           Function |        int |            |            |            |               0 |        100 | 10
      name |    Array Identifier |       char |            |         10 |            |               0 |          0 | 6
    printf |           Function |        int |            |            |            |               0 |        100 | 13
      book |         Identifier |     struct |            |            |            |               0 |        100 | 4
       num |         Identifier |        int |            |            |            |               0 |        100 | 12
    author |    Array Identifier |       char |            |         10 |            |               0 |          0 | 7
    struct |            Keyword |            |            |            |            |               0 |        100 | 4


                            CONSTANT TABLE
                            --------------
           CONSTANT |                TYPE
            "Hello" |      String Constant
                 10 |      Number Constant
                  3 |      Number Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 7

```c
//ERROR FREE - This test case includes escape sequences
#include<stdio.h>

int main()
{
        char es = '\a';
        printf("Hello");

}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |        100 | 4
      char |           Keyword |            |            |            |            |               0 |        100 | 6
      main |          Function |        int |            |            |            |               0 |        100 | 4
    printf |          Function |       char |            |            |            |               0 |        100 | 7
        es |        Identifier |       char |            |            |            |               0 |        100 | 6


                            CONSTANT TABLE
                            --------------
          CONSTANT |                TYPE
           "Hello" |      String Constant
              '\a' |   Character Constant
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 8

```c
//ERROR FREE - This test case includes nested comments
#include<stdio.h>

int main()
{
        /*This is /* nested comment */!!*/
        /*This is a
        normal comment*/
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |        100 | 4
      main |          Function |        int |            |            |            |               0 |        100 | 4


                            CONSTANT TABLE
                            --------------
          CONSTANT |                TYPE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Cases With Error

Test Case 9

```c
//WITH ERROR - This test case includes duplicate declaration of identifier
#include<stdio.h>

void main()
{
        int a = 1;
        int a = 2;
        printf("%d", a);

}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 7 Duplicate value!
 a

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 10

```c
//WITH ERROR - This test case includes array size less than 1
#include<stdio.h>

void main()
{
        int a[0];
        printf("hello\n");

}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 6 Array must have size greater than 1!
 ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 11

```c
//WITH ERROR - This test case includes duplicate function declaration
#include<stdio.h>
void func()
{
        printf("hello\n");
}
void func()
{
        printf("hello\n");
}
void main()
{
        printf("hello\n");
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
ERROR: Cannot Redeclare same function!

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 12

```c
//WITH ERROR - This test case includes void parameter for function
#include<stdio.h>
void func(void x)
{
        printf("hello\n");
}
void main()
{
        printf("hello\n");
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
ERROR: Here, Parameter cannot be of void type

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 13

```c
//WITH ERROR - This test case includes a function call to undeclared function
#include<stdio.h>
void main()
{
        func();
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 5 Function not declared (

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 14

```
//WITH ERROR - This test case includes a function call to function declared after main
#include<stdio.h>
void main()
{
	func();
}
void func()
{
	printf("hello\n");
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 5 Function not declared (

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 15

```
//WITH ERROR - This test case includes void return for int function
#include<stdio.h>
int func()
{
	printf("hello\n");
	return;
}
void main()
{
	func();
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 6 ERROR: Cannot have void return for non-void function!
 ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 16

```
//WITH ERROR - This test case includes int return for void function
#include<stdio.h>
void func()
{
	printf("hello\n");
	return 0;
}
void main()
{
	func();
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 6 Non-void return for void function! ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 17

```
//WITH ERROR - This test case includes function call with incorrect number of parameters during call
#include<stdio.h>
int func(int a, int b)
{
	return a+b;
}
void main()
{
	int a = 1;
	int x;
	x = func(a);
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 11 Number of parameters not same as number of arguments during function call! )

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 18

```
//WITH ERROR - This test case includes if condition not of type int
#include<stdio.h>
void main()
{
        float x = 1.0;
        if(x)
                print("hello\n");
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 6 ERROR: Here, condition must have integer value!
 )

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 19

```
//WITH ERROR - This test case includes case where array identifier has no subscript
#include<stdio.h>
void main()
{
        int ar[2] = {1,2};
        ar = 3;
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
ar
Line No. : 6 Array Identifier has No Subscript
 =

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 20

```
//WITH ERROR - This test case includes case value in subscript not integer
#include<stdio.h>
void main()
{
        int ar[2] = {1, 2};
        float y = 1.0;
        ar[y] = 1;

}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 7 Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 21

```
//WITH ERROR - This test case includes case where lhs of assignment has more than 1 single variable
#include<stdio.h>
void main()
{
        int x = 1;
        int y = 1;
        int z = 1;
        x + y = z;

}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
Line No. : 8 syntax error =

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Test Case 22

```c
//WITH ERROR- This test case includes use of out of scope id
#include <stdio.h>
void main()
{
        int x = 1;
        if(1)
        {
                int y = 2;
        }
        y = 3;
}
```

```
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$ ./a.out
y
Line No. : 10 Identifier undeclared
 =

UNSUCCESSFUL: INVALID PARSE
mrushad@mrushad-HP-Notebook:~/Desktop/COmpilerLab/Compiler Project/3-Semantic/new$
```

Implementation

The regular expressions used to identify the different tokens of the C language are fairly straightforward. Similar lexer code to the one submitted in the previous phase was used. A few features require a significant amount of thought are mentioned below:

- The Regex for Identifiers
- Support for Multi-line and Nested Comments and Error Handling for Unmatched Comments
- Literals
- Error Handling for Incomplete String

The parser uses a number of grammar production rules to implement the C programming language grammar. The parser also takes the help from the lexer for its functioning. The lexer outputs tokens one at a time and these tokens are used by the parser. The parser then applies the corresponding production rules on the token to insert the type, value, array dimensions, function parameters etc. into the symbol table. Along with this, semantic actions were also added to each
Production rule to check if the structure created has some meaning or not.

The following functions are used for semantics analysis. The functions used in the previous stage remain the same.

1. insert_SymbolTable_nest()

This function takes care of the insertion of the nesting value into the Symbol Table. It is also used to insert variables which are already there in the symbol table but have a different nesting value.

2. insert_SymbolTable_paramscount()

This function is used to insert the number of parameters for a function identifier. This number is calculated by the parser and passed to this function.

3. getSTparamscount()

This function is used to get the number of parameters of a function identifier.

4. remove_scope()

This function indicates that the scope of a variable is over by assigning the default value (default value used here is 100) to the nested_val of the variable in the symbol table.

5. check_scope()

This function is used to check if the identifier which calls this function is declared within the current scope indicated by the current_nested_val.

6. check_function()

This function is used to check if the identifier passed is a function or not.

7. check_array()

This function is used to check if the identifier is an array identifier or not.

8. duplicate()

This function is used to check if the identifier passed is already in scope.

9. check_duplicate()

This function is used to check if there is a re-declared function identifier.

10. check_declaration()

This function is used to check if a function is declared.

11. check_params()

This function is used to check if the parameters are of type void which are invalid.

12. gettype()

This function is used to return the first character of the data type identifier which is used while performing Type Check.

Functionality

The section below describes how the code identifies different constructs of the C language and by executing code blocks for these constructs determines if the program is semantically correct. The handling of various semantic errors that have been accounted are shown below.

1. Variable Declaration

- Undeclared Identifier: An undeclared identifier is identified by passing the identifier to the check_scope() function. This function checks if the variable exists in the symbol table and if it does, it also ensures that the scope of the variable is valid.
- Duplicate Declaration: A duplicate identifier is identified by passing the identifier to the duplicate() function. This function checks if a variable with the same name and scope exists. If such a variable exists, it returns error.

- Array Size less than 1: Array size less than 1 is not permitted in C. During parsing, we check if the integer_constant passed as the array dimension is below 1 and if it is, we report an error.

2. Functions

- Function Re-declaration: This function identifier is passed to the check_duplicate() function which checks the symbol table for the same function identifier.
- void Parameters: The datatype of each function parameter is passed to the function check_params() which returns an error if the datatype passed is void.
- Function not Declared: This function identifier is passed to the check_declaration() function which checks the symbol table for the same function identifier.
- Return type mismatch: When we encounter a function identifier in a declaration, the return type of the function is stored in the variable currfunctype. This variable is used to check for return type mismatch. If we encounter a "return;" at the end of the function declaration and the currfunctype is not void, we return an error. Similarly, we check for other return statements as well.
- Parameter Count: We store the number of arguments used in the function call of a particular function in the call_params_count variable. Also, during declaration of the function, we store the parameter count in the symbol table. By comparing these two values, we determine if there is an error in the number of parameters and arguments.

3. Expressions

- The program does not allow operations on operators of different data types. Every production rule for an expression check if the type of the left hand side matches the type of the right hand side. The type of the LHS and RHS are determined when we encounter a constant and we store what the type of the constant is.

Future work

The yacc script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

Results

The parser was able to be successfully parse the tokens recognized by the flex script. The program gives us the list of the symbols and constants. The program also detects syntactical errors and semantics errors if found. Thus, the semantic stage is an essential part of the compiler and is needed for the simplifications of the compiler. It makes the compiler more efficient and robust

References

- Compiler Principles, Techniques and Tool by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jefferey D. Ullman
- https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis