Semantic Checker for C Language

Project - 3 (CO351)
14th February 2019

Submitted to :
Dr. P. Santhi Thilagam
Faculty, Dept of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal

Group members :
Ayush Kumar - 16CO208
Gauri Baraskar - 16CO209
Sanjana Krishnam - 16CO139

Abstract

Objective

The main goal of this project is to design a mini compiler for a subset of the C language as part of the Compiler Design Lab(CO351) course. The compiler is to be built in four phases finishing at the Intermediate Code Generation Phase. The subset of the C language chosen is to include certain data types, constructs and functions as mentioned in the specifications below. The implementation will be carried out with LEX and YACC.

Phases of the project

- Implementation of Scanner/Lexical Analyser
- Implementation of Parser
- Implementation of Semantic Checker for C language
- Intermediate Code generation for C language

Mini-compiler Specifications

The compiler is going to support the following cases :

1. **Keywords** - int, break, continue, else, for, void, goto, char, do, if, return, while
2. **Identifiers** - identified by the regular expression (_ |{letter})({letter}|{digit}| _){0,31}
3. **Comments** - single line comments (specified with // or /* … */), multi-line comments (specified with /* … */)
4. **Strings** - can identify strings mentioned in double quotes
5. **Preprocessor directives** - can identify filenames (stdio.h) after #include
6. **Data types** - int,char (supports comma declaration)
    - Syntax:
        - int a=5;
        - Char newchar = ‘a’
7. **Arrays** - int A[n]
    - Syntax:
        - int A[3];
        - Array initialization through for loops
8. **Punctuators** - [ ] , <> , {} , , , : , = , ; , # , ” ” ,’ ’
9. **Operators** - arithmetic (+, - , * , /) ,increment(++) and decrement(--), assignment (=)
10. **Condition** - if else
    - Syntax:
        - if (condition == true){
            //code
            }
            else{
            //code }

11. **Loops -**
    - Syntax
        - while(condition){
          //code
          }
        - for(initialization;condition;increment/decrement){
          //code
          }
        - while(condition){
          //code
          }
        - do{
          //code
          }while(condition);

    The loop control structures that are supported are break,continue and goto.

12. **Functions-**
    Void functions with no return type and a single parameter will be implemented
    - Syntax
        - void sample_function(int a){
          //code
          }

The mini compiler will be implemented in a straightforward fashion , using Lex and YACC tools starting off with the Scanner as the first module. If time permits we will add more features to cover a larger subset of the C language.

Contents

Abstract

Introduction

Semantic Checker

Yacc Script

C Program

Design of Programs

Updated Lexer Code

Updated Parser Code

Test Cases

Functionality and Implementation

Results and Future Work

Results

Future work

References

List of Figures and Tables

1. **Table 1: Test Cases without errors**
2. **Table 2: Test cases with errors**
3. **Figure 1: Input : Sample program with no errors**
4. **Figure 2: Output : Sample program with no errors**
5. **Figure 3: Input : Sample program based on variable scopes**
6. **Figure 4: Output : Sample program based on variable scopes**
7. **Figure 5: Input : Sample program for error due to type matching**
8. **Figure 6: Output : Sample program for error due to type matching**
9. **Figure 7: Input : Sample program for duplicate functions**
10. **Figure 8: Output : Sample program for duplicate functions**
11. **Figure 9: Input : Sample program for undeclared function**
12. **Figure 10: Output : Sample program for undeclared function**
13. **Figure 11: Input : Sample program for function call argument mismatch**
14. **Figure 12: Output : Sample program for function call argument mismatch**
15. **Figure 13: Input : Sample program for incorrect function return type**
16. **Figure 14: Output : Sample program for incorrect function return type**
17. **Figure 15: Input: Sample program for incorrect use of loop control structures**
18. **Figure 16: Output: Sample program for incorrect use of loop control structu**

Introduction

Semantic Checker

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.

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called coercions. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number or report an error.

Yacc Script

Yacc stands for Yet Another Compiler-Compiler. Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

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 given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

Definition section
%%

Rules section

%%

C code section

The definition section is used to define any parameters for the C program, any header files to be included and global variable declarations. We also define all parameters related to the parser here, specifications like using Leftmost derivations or Rightmost derivations, precedence and left right associativity are declared here, data types and tokens which will be used by the lexical analyser are also declared in this stage. We also declare all the variables and types associated for non-terminals to assist with semantic checking.

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. It also consists of code blocks with semantic rules. Yacc uses this rules for reducing the token stream received from the scanner, all rules are linked to each other from the start state, which is declared in the rules section.

In the C code section the parser is called, and the symbol table and constant tables are initialised in this section. After parsing and semantic checking the tables are filled with nesting level, function flag, array flag, array dimension, etc. The lex.yy.c file created by the lex script is also included from here which the parser calls. In this section we also define the error function used by the parser/semantic checker to report syntactical and semantic errors along with line numbers.

C Program

The parser(yacc script) takes C source files as input for parsing and semantic checking, the files are supplied using command line arguments, the path is given as argument.

For testing the parser the following commands have to be executed

```
lex lexer_parser.l
yacc -d parser.y
gcc y.tab.c -ll -ly
./a.out testcases/test-file-name.c
```

We have used various C programs for testing. We have induced errors in some programs, such as mismatching of arguments with parameters of functions or undeclared variables. This is done in order to verify that the semantic checker works and outputs the error with the line number. We have also

used semantically correct C programs for testing to verify the correct functioning of the semantic checker.

Design of Programs

Updated Lex Code

```
 /* This file describes the rules for the lexer */
%{
       #include<stdio.h>
       #include<stdlib.h>
       #include<limits.h>
       int ErrFlag = 0;

       extern int is_declaration;

       #include "y.tab.h"
%}


 /* Declarations */

letter [a-zA-Z]
digit [0-9]
whitespace [ \t\r\f\v]+
identifier (_|{letter})({letter}|{digit}|_)*
hex [0-9a-fA-F]


 /* States */

%x PREPROCESSOR
%x MACROPREPROCESSOR
%x COMMENT
%x SLCOMMENT

%%
/* Keywords */

"int"                               {return INT;}
"short"                             {return SHORT;}
"long"                              {return LONG;}
"char"                              {return CHAR;}
"void"                              {return VOID;}
"if"                                {return IF;}
"else"                              {return ELSE;}
"for"                               {return FOR;}
"do"                                {return DO;}
```

```
"while"                             {return WHILE;}
"break"                             {return BREAK;}
"continue"                          {return CONTINUE;}
"return"                            {return RETURN;}
"float"                             {return FLOAT;}


 /* Constants */

[0][x|X]{hex}+              {yylval.tbEntry = InsertEntry(ConstantTable, yytext ,
(int)strtol(yytext, NULL, 16),"HEX",yylineno,0);return HEX_CONSTANT;}
{digit}+                    {yylval.tbEntry = InsertEntry(ConstantTable, yytext , (int)
atoi(yytext),"INT",yylineno,0);return INT_CONSTANT;}
({digit}*)["."]({digit}+)   {yylval.tbEntry = InsertEntry(ConstantTable, yytext ,
atof(yytext),"FLOAT",yylineno,0);return DEC_CONSTANT;}
({digit}+)["."]({digit}*)   {yylval.tbEntry = InsertEntry(ConstantTable, yytext ,
atof(yytext),"FLOAT",yylineno,0);return DEC_CONSTANT;}

{identifier} {
  if(strlen(yytext) <= 32)
       {
       yylval.str = yytext;
       if(is_declaration)
       {
       yylval.tbEntry =
InsertEntry(SymbolTable,yytext,INT_MAX,curr_data_type,yylineno,curr_nest_level);
       }
       return IDENTIFIER;
       }
  else
  {
       printf("Error %d: Identifier too long,must be between 1 to 32 characters\n", yylineno);
       ErrFlag = 1;
  }
}


{digit}+({letter}|_)+               {printf("Error %d: Illegal identifier format\n", yylineno);
ErrFlag = 1;}
{whitespace}                        ;


 /* Preprocessor Directives */

^"#include"                         {BEGIN PREPROCESSOR;}
<PREPROCESSOR>{whitespace}          ;
<PREPROCESSOR>"<"[^<>\n]*">"        {BEGIN INITIAL;}
<PREPROCESSOR>\"[^<>\n]*\"          {BEGIN INITIAL;}
<PREPROCESSOR>"\n"                  { yylineno++; BEGIN INITIAL; ErrFlag=1;}
<PREPROCESSOR>.                     {yyerror("Improper Header");}
```

```
 /* Macropreprocessor Directives */

^"#define"                                          {BEGIN MACROPREPROCESSOR;}
<MACROPREPROCESSOR>{whitespace}                     ;
<MACROPREPROCESSOR>({letter})({letter}|{digit})*    {BEGIN INITIAL;}
<MACROPREPROCESSOR>\n                               {yylineno++; BEGIN INITIAL;}
<MACROPREPROCESSOR>.                                {BEGIN INITIAL;ErrFlag=1;}


 /* Comments */

"/*"                                {BEGIN COMMENT;}
<COMMENT>.|{whitespace}             ;
<COMMENT>\n                         {yylineno++;}
<COMMENT>"*/"                       {BEGIN INITIAL;}
<COMMENT>"/*"                       {yyerror("Improper Comment");yyterminate();}
<COMMENT><<EOF>>                    {yyerror("Improper Comment");yyterminate();}
"//"                                {BEGIN SLCOMMENT;}
<SLCOMMENT>.                        ;
<SLCOMMENT>\n                       {yylineno++; BEGIN INITIAL;}


 /* Operators */

"+"                                 {return ADD;}
"-"                                 {return SUBTRACT;}
"*"                                 {return MULTIPLY;}
"/"                                 {return DIVIDE;}
"="                                 {return ASSIGN;}
"--"                                {return DECREMENT;}
"++"                                {return INCREMENT;}

"+="                                {return ADD_ASSIGN;}
"-="                                {return SUB_ASSIGN;}
"*="                                {return MUL_ASSIGN;}
"/="                                {return DIV_ASSIGN;}

">"                                 {return GREATER_THAN;}
"<"                                 {return LESSER_THAN;}
">="                                {return GREATER_EQ;}
"<="                                {return LESS_EQ;}
"=="                                {return EQUAL;}

"||"                                {return LG_OR;}
"&&"                                {return LG_AND;}
"!"                                 {return NOT;}
"!="                                {return NOT_EQ;}
```

```
 /* Strings and Characters */

\"[^\"\n]*$                         {ErrFlag=1; yyterminate();}
\"[^\"\n]*\" {
  if(yytext[yyleng-2]=='\\') {
       yyless(yyleng-1);
       yymore();
  }
  else
  {
       yylval.tbEntry = InsertEntry(ConstantTable,yytext,yylval.dval,"CHAR",yylineno,0);
       return STRING;
  }
}

\'[^\'\n]\'                         { yylval.tbEntry =
InsertEntry(ConstantTable,yytext,yylval.dval,"CHAR",yylineno,0); return STRING;}


 /* Punctuators */


"\n"                                {yylineno++;}
.                                   {return yytext[0];}

%%
```

There is no change in lex code with respect to identifying of tokens. But while inserting constants and variables it stores the pointer to the location of constant or the variable in yylval which is passed to the parser. Thus parser , lexical analyser , and semantic analyser share the symbol table. The script still identifies lexical errors and reports them to the parser which then throws a syntax error.

Updated Parser Code

```
/* The file describes the grammar for our parser */
%{
    // Header files
    #include<stdio.h>
    #include<stdlib.h>
    #include "tables.h"
    #include<limits.h>
    #include<ctype.h>
    #include<string.h>

    // Initialising Symbol table and constant table
    entry **SymbolTable = NULL;
    entry **ConstantTable = NULL;

    int yyerror(char *msg);
    int checkScope(char *value);
    int typeCheck(char*,char*,char*);
    int checkFunc(char*);
    char* curr_data_type;
    int yylex(void);
    int is_loop = 0;
    int curr_nest_level = 1;
    int return_exists = 0;
    int is_param = 0;
    extern int yylineno;
    extern char* yytext;


    int is_declaration = 0;
    int is_function = 0;
    char* func_type;
    char *param_list[10];
    char *arg_list[10];

    int p_idx = 0;
    int a_idx = 0;
    int p=0;
%}


// Data types of tokens
%union{
        char *str;
        entry *tbEntry;
        double dval;
```

```
}

/* Operators */
%token ADD SUBTRACT MULTIPLY DIVIDE ASSIGN ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN
/* Relational Operators */
%token GREATER_THAN LESSER_THAN LESS_EQ GREATER_EQ NOT_EQ EQUAL
/* Keywords */
%token VOID IF ELSE FOR DO WHILE BREAK CONTINUE RETURN
/* Data types */
%token INT SHORT LONG CHAR FLOAT
/* Logical Operators */
%token LG_OR LG_AND NOT
/* Assignment Operators */
%token DECREMENT INCREMENT
/* Constants */
%token <tbEntry> HEX_CONSTANT DEC_CONSTANT INT_CONSTANT STRING
/* Identifier */
%token <tbEntry> IDENTIFIER


%type <tbEntry> identifier varDecId
%type <str> mutable call factor expression simpleExpression andExpression sumExpression unaryExpression
unaryRelExpression term immutable relExpression const_type


// Start Symbol of the grammar
%start program

/* Precedence of Operators */
%left ','
%right ASSIGN
%left LG_OR
%left LG_AND
%left EQUAL NOT_EQ
%left LESSER_THAN GREATER_THAN LESS_EQ GREATER_EQ
%left ADD SUBTRACT
%left MULTIPLY DIVIDE MOD
%right NOT


%nonassoc IFX
%nonassoc ELSE
%%


    program : declarationList;

    declarationList : declarationList declaration | declaration;

    declaration : varDeclaration | funDeclaration ;
```

```
    varDeclaration : typeSpecifier varDeclList ';' {is_declaration = 0;} ;

    varDeclList : varDeclList ',' varDeclInitialize | varDeclInitialize;

    varDeclInitialize : varDecId | varDecId assign_symbol simpleExpression
{  typeCheck($1->data_type,$3,"="); is_declaration=1;} ;
    varDecId : identifier {$$=$1;} | identifier '[' INT_CONSTANT  ']'
   {
     if($3->value < 1){
       yyerror("Arrays can't have dimension lesser than 1");
     }
    $$=$1;
    $1->is_array = 1;
    $1->array_dim = (int)$3->value;
   }
   ;
    typeSpecifier : INT {curr_data_type = strdup("INT");  is_declaration = 1; }
                  | VOID {curr_data_type = strdup("VOID");  is_declaration = 1; }
                  | CHAR {curr_data_type = strdup("CHAR");  is_declaration = 1;}
                  | FLOAT {curr_data_type = strdup("FLOAT");  is_declaration = 1;}
                  ;

    assign_symbol : ASSIGN {is_declaration = 0;}

    funDeclaration : typeSpecifier
                     identifier             {
                                                func_type = curr_data_type;
                                                is_declaration = 0;
                                            }
                     '(' params ')'         {
                                                fill_parameter_list($2,param_list,p_idx);
                                                p_idx = 0;
                                                is_function = 1;
                                                int flag = set_is_function(SymbolTable,$2->lexeme);
                                                if(flag == 0){return -1;}
                                                p=1;
                                            }
                     compoundStmt           {
                                              is_function = 0;
                                              if(!return_exists && strcmp(func_type,"VOID") != 0)
                                              {
                                                yyerror("This Function must have a return type");
                                              }
                                          };

     // Rules for parameter list
    params : {is_param = 1; }paramList | ;
    paramList : paramList ',' paramTypeList | paramTypeList;
    paramTypeList : typeSpecifier
```

```
                paramId           {
                                      param_list[p_idx] = (char*)malloc(sizeof(curr_data_type));
                                      strcpy(param_list[p_idx++],curr_data_type);
                                      is_declaration = 0;
                                  }
                ;
paramId : identifier | identifier '[' ']';
// Types os statements in C
statement : expressionStmt  | compoundStmt  | selectionStmt | iterationStmt | jumpStmt | returnStmt
          | breakStmt | varDeclaration ;

// compound statements produces a list of statements with its local declarations
compoundStmt : {curr_nest_level++;}'{' statementList '}'
               {curr_nest_level++;   insertNest(curr_nest_level - 1,yylineno);};
statementList : statementList statement
              |  ;
// Expressions
expressionStmt : expression ';' | ';' ;
selectionStmt : IF '(' simpleExpression ')' statement %prec IFX
              | IF '(' simpleExpression ')' statement ELSE statement
              ;

iterationStmt : WHILE '(' simpleExpression ')' {is_loop = 1;} statement {is_loop = 0;}
              | DO {is_loop = 1;}statement WHILE '(' expression ')' ';' {is_loop = 0;}
              | FOR '(' optExpression ';' optExpression ';' optExpression ')'{is_loop = 1;}
              statement {is_loop = 0;};
// Optional expressions in case of for
optExpression : expression | ;

jumpStmt : CONTINUE ';' {
                          if(!is_loop) {
                             yyerror("Illegal use of continue");
                          }
                        }
           ;

returnStmt : RETURN ';'  {
                           if(is_function) {
                             if(strcmp(func_type,"VOID")!=0)
                                yyerror("return type (VOID) does not match function type");
                           }
                         }



           | RETURN expression {
                                 return_exists = 1;
                                 if(strcmp(curr_data_type,$2)!=0)
                                   yyerror("return type does not match function type");
                               } ;
```

```
breakStmt : BREAK ';' {if(!is_loop) {yyerror("Illegal use of break");}};

expression : mutable ASSIGN expression {typeCheck($1,$3,"=");$$ = $1;}
           | mutable ADD_ASSIGN expression {typeCheck($1,$3,"+=");$$ = $1;}
           | mutable SUB_ASSIGN expression {typeCheck($1,$3,"-=");$$ = $1;}
           | mutable MUL_ASSIGN expression {typeCheck($1,$3,"*=");$$ = $1;}
           | mutable DIV_ASSIGN expression {typeCheck($1,$3,"/=");$$ = $1;}
           | mutable INCREMENT { $$ = $1;}
           | mutable DECREMENT { $$ = $1;}
           | simpleExpression { $$ = $1;}
           ;

simpleExpression : simpleExpression LG_OR andExpression { $$ = $1;}
                 | andExpression { $$ = $1;};

andExpression : andExpression LG_AND unaryRelExpression { $$ = $1;}
              | unaryRelExpression { $$ = $1;} ;

unaryRelExpression : NOT unaryRelExpression { $$ = $2;}
                   | relExpression { $$ = $1;} ;

relExpression : sumExpression GREATER_THAN sumExpression {typeCheck($1,$3,">");$$ = $1;}
              | sumExpression LESSER_THAN sumExpression {typeCheck($1,$3,"<");$$ = $1;}
              | sumExpression LESS_EQ sumExpression {typeCheck($1,$3,"<=");$$ = $1;}
              | sumExpression GREATER_EQ sumExpression {typeCheck($1,$3,">=");$$ = $1;}
              | sumExpression NOT_EQ sumExpression {typeCheck($1,$3,"!=");$$ = $1;}
              | sumExpression EQUAL sumExpression {typeCheck($1,$3,"==");$$ = $1;}
              | sumExpression { $$ = $1;}
              ;
sumExpression : sumExpression ADD term {typeCheck($1,$3,"+");$$ = $1;}
              | sumExpression SUBTRACT term {typeCheck($1,$3,"-");$$ = $1;}
              | term { $$ = $1;}
              ;



term : term MULTIPLY unaryExpression {typeCheck($1,$3,"*");$$ = $1;}
     | term DIVIDE unaryExpression {typeCheck($1,$3,"/");$$ = $1;}
     | unaryExpression { $$ = $1;}
     ;

unaryExpression : ADD unaryExpression { $$ = $2;}
                | SUBTRACT unaryExpression { $$ = $2;}
                | factor { $$ = $1;} ;


factor : immutable {$$ = $1;} | mutable {$$ = $1;};
mutable : identifier {
                        if(checkScope(yylval.str) == 0){ return -1;};
                        $$ = $1->data_type;
```

```
                    }
        | identifier '[' INT_CONSTANT ']' {
                                            if($3->value < 0 || $3->value >= $1->array_dim)
                                            {yyerror("Exceeds Array Dimensions\n"); }
                                            $$ = $1->data_type;
                                          }

immutable : '(' expression ')' { $$ = $2;}
          | call {$$=$1;}
          | const_type {$$=$1;} ;

call : identifier '(' args ')' {
                                  if(checkFunc($1->lexeme) == 0)
                                    {return -1;};
                                  $$ = $1->data_type;
                                  check_parameter_list($1,arg_list,a_idx);
                                  a_idx = 0;
                                }
 args : argList | ;

 argList : argList ',' arg
         | arg ;

 arg : expression     {
                         arg_list[a_idx] = (char *)malloc(sizeof($1));
                         strcpy(arg_list[a_idx++],$1);
                      }
      ;

 const_type : DEC_CONSTANT { $$ = $1->data_type;}
            | INT_CONSTANT { $$ = $1->data_type;}
            | HEX_CONSTANT { $$ = $1->data_type;}
            | STRING       { $$ = $1->data_type;}
            ;

 identifier : IDENTIFIER {
                           if(is_declaration){
                           $$ = $1;
                           is_function = 0;
                           }
                           else
                           {
                           $1 = Search(SymbolTable,yytext);
                           $$ = $1;

                           if($1 == NULL)
                           {
                             yyerror("Variable Not Declared");
                             return -1;
                           }
```

```
                                    }
                            }
                    ;
%%


int checkFunc(char* lexeme)
{
    entry *res = searchFunc(SymbolTable,lexeme);
    if(res != NULL)
    {
        res = InsertSearch(SymbolTable,lexeme,curr_nest_level);
        if(res != NULL)
        {
            yyerror("Defined as variable in this scope, calling not allowed");
            return 0;
        }
        else
        {
            return 1;
        }

    }
    else
    {
        yyerror("No such declaration\n");
        return 0;
    }
}

int typeCheck(char *a,char *b,char *c){

       if(strcmp(a,b)!=0){
              yyerror("Type Mismatch");
              exit(0);
       }

       else
              return 1;
}


void disp()
{
    printf("\n\tSymbol table");
    Display(SymbolTable);
    printf("\n\tConstant table");
    DisplayConstant(ConstantTable);
}
```

```
int checkScope(char *val)
{
    char *extract = (char *)malloc(sizeof(char)*32);
    int i;
    // Don't touch this CRUCIAL AS FUCK
    for(i = 0; i < strlen(val) ;i=i+1)
    {
        //printf("%d\n",i);
        if((isalnum(*(val + i)) != 0) || (*(val + i)) == '_')
        {
            *(extract + i) = *(val + i);
        }
        else
        {
            *(extract + i) = '\0';
            break;
        }
    }

    entry *res = Search(SymbolTable,extract);
    // First check if variable exists then check for nesting level
    if (res == NULL)
    {
        yyerror("Variable Not Declared\n");
        return 0;
    }
    else
    {
        int level = res->nesting_level;
        int endLine = -1;
        if(Nester[level] == NULL)
            endLine = yylineno + 100;
        else
            endLine = Nester[level]->line_end;

        if(level <= curr_nest_level && yylineno <= endLine)
        {

            return 1;
        }
        else
        {

            yyerror("Variable Out Of Scope\n");
            return 0;
        }
    }

}
```

```
#include "lex.yy.c"
int main(int argc , char *argv[]){

    SymbolTable = CreateTable();
    ConstantTable = CreateTable();
    nested_homekeeping();
    int i;
    // Open a file for parsing
    yyin = fopen(argv[1], "r");

    if(!yyparse())
    {
        printf("\nParsing complete.\n");
        disp();
    }
    else
    {
            printf("\nParsing failed.\n");
    }

    fclose(yyin);
    return 0;
}

int yyerror(char *msg)
{
  // Function to display error messages with line no and token
    printf("Line no: %d Error message: %s Token: %s\n", yylineno, msg, yytext);
    return 0;
}
```

The above code is the yacc script semantic checking of the code. Semantic rules related to each production are written opposite to it in a block of code comprised between flower braces.

The below file is a tables.h which deals with functions related to entries within the tables.

```
/*
  This File implements all functions required to implement Symbol Table and
  Constant Table.
  The functions here are used to Add entries to a hash table.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 100

extern int yyerror(char *msg);
extern int curr_nest_level;
extern int is_function;
extern int yylineno;

struct table_entry{

    int line_number;
    char *lexeme;
    double value;
    char* data_type;
    int is_function;
    int is_array;
    int nesting_level;
    int array_dim;
    char* parameter_list[10];
    int num_params;
    struct table_entry *next;
};

struct nested{
     int nesting_level;
    int line_end;
};

struct nested **Nester = NULL;

void nested_homekeeping()
{
  Nester = malloc(sizeof(struct nested*)*SIZE);

  int i;
  for(i = 0;i<SIZE;i++)
  {
```

```
    Nester[i] = NULL;
  }
}

void insertNest(int nesting_level,int line_end)
{
  struct nested *temp = NULL;
  temp = malloc(sizeof(struct nested));
  temp->nesting_level =  nesting_level;
  temp->line_end = line_end;

  Nester[nesting_level] = temp;
}


typedef struct table_entry entry;

void Display(entry** TablePointer);
int hash(char *lexeme)
{
  int hash = 0,i=0;

  for(i=0; i < strlen(lexeme); i++)
    hash += lexeme[i];

  return hash % SIZE;
}

entry** CreateTable()
{
  entry **TablePointer = NULL;

  TablePointer = malloc(sizeof(entry*)*SIZE);

  if(TablePointer == NULL)
    return NULL;

  int i;
  for(i=0;i<SIZE;i++)
    TablePointer[i] = NULL;

  return TablePointer;
}

entry* searchFunc(entry** TablePointer,char *lexeme)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];
```

```
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0 && head->is_function == 1)
    {
      return head;
    }
    else
      head = head->next;
  }
  if(head == NULL)
    return NULL;
  return head;
}

entry* Search(entry** TablePointer, char *lexeme)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0)
    {
      return head;
    }
    else
      head = head->next;
  }
  if(head == NULL)
    return NULL;
  return head;
}

entry* InsertSearch(entry** TablePointer, char *lexeme,int currScope)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  head = TablePointer[temp];
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0 && currScope == head->nesting_level)
    {
      return head;
     }
    else
       head = head->next;
```

```
  }
  if(head == NULL)
    return NULL;
  return head;
}

int funcSearch(entry** TablePointer,char *lexeme,int currLine)
{
  int temp = hash(lexeme);
  entry *head = NULL;
  entry **array = NULL;
  int raiseFlag = 0;

  array = malloc(sizeof(entry*)*SIZE);

  if(array == NULL)
    return 0;

  int i;
  for(i=0;i<SIZE;i++)
    array[i] = NULL;
  i = 0;
  head = TablePointer[temp];
  //Display(TablePointer);
  while(head != NULL)
  {
    if(strcmp(head->lexeme,lexeme) == 0 )
    {
      array[i] = head;
      head=head->next;
    }
    else
      head = head->next;
  }
  for(i=0;i<SIZE;i++)
  {

    entry* tempPoint = array[i];
    if(tempPoint == NULL)
      continue;
    else if(tempPoint->is_function == 1)
    {
      raiseFlag = 1;
      break;
    }
   }
  if(raiseFlag == 1)
  {
    head = TablePointer[temp];
```

```
    entry* prev = NULL;

    while(head != NULL)
    {
      printf("1\n");
      if(strcmp(head->lexeme,lexeme) == 0 && head->line_number == currLine)
      {
        if(prev != NULL)
        {
          prev->next = head->next;
          free(head);
        }
        else
        {
          TablePointer[temp] = NULL;
          free(head);
        }

        return 0;
      }
      else
      {
        prev = head;
        head = head->next;
      }

    }
    return 0;
  }
  else
  {
    return 1;
  }


}

int set_is_function(entry** TablePointer, char *lexeme)
{
 if(funcSearch(TablePointer,lexeme,yylineno))
  {
    entry* Entry = Search(TablePointer,lexeme);
   if (Entry == NULL)
     return 0;
   else
     Entry->is_function = 1;
    return 1;
  }
  else
  {
```

```
    yyerror("Duplicate Functions Not allowed\n");
    return 0;
  }

}

entry* InsertEntry(entry** TablePointer, char *lexeme,double value,char* DataType,int
line_number ,int nesting_level)
{
  int temp = hash(lexeme);
  if(InsertSearch(TablePointer,lexeme,curr_nest_level) != NULL)
  {
    yyerror("Duplicate Variable Declaration not allowed\n");
    return TablePointer[temp];
  }

  else
  {

    entry *head = NULL;

    head = TablePointer[temp];

    entry *tempPoint = NULL;
    tempPoint = malloc(sizeof(entry));
    tempPoint->lexeme = strdup(lexeme);
    tempPoint->value = value;
    tempPoint->data_type = strdup(DataType);
    tempPoint->line_number = line_number;
    tempPoint->nesting_level = nesting_level;
    tempPoint->next = NULL;

    if (head == NULL)
    {
      TablePointer[temp] = tempPoint;
    }
    else
    {
      tempPoint->next = TablePointer[temp];
      TablePointer[temp] = tempPoint;
    }

  }
  return TablePointer[temp];

}
```

```
void fill_parameter_list(entry* tableEntry, char **list, int n)
{

   int i;
   for(i=0; i<n; i++)
   {
    tableEntry->parameter_list[i] = (char *)malloc(sizeof(char));
    if(strcmp(list[i],"VOID") == 0)
    {
      yyerror("Parameters of type void not allowed\n");
      return;
    }
     strcpy(tableEntry->parameter_list[i],list[i]);
   }
   tableEntry->num_params = n;
}

int check_parameter_list(entry* tableEntry, char** list, int m)
{

 if(m != tableEntry->num_params)
 {
        yyerror("Number of parameters and arguments do not match");
    exit(0);
 }

 int i;
 for(i=0; i<m; i++)
 {
        if( strcmp(list[i],tableEntry->parameter_list[i]) !=0 ){
        yyerror("Parameter and argument types do not match");
    exit(0);
  }

 }

 return 1;
}


void Display(entry** TablePointer)
{
  int i =0;
  entry *temp = NULL;

  printf("\n\n");
  printf("-----------------------------------------\n");
  printf("\n\t(lexeme, \t    value, Data type, Line Number, isArray, ArrayDimensions,
  isFunction, Nesting Level, num_params\n");
```

```
  for(i=0;i<SIZE;i++)
  {
    temp = TablePointer[i];
    while(temp != NULL)
    {
      printf("\t(%6s, %-5f, %9s, %11d, %7d, %15d, %10d, %13d, %10d)\n"
,temp->lexeme,temp->value,temp->data_type,temp->line_number,temp->is_array,temp->array_dim,te
mp->is_function,temp->nesting_level, temp->num_params);
      int j;
      if(temp->num_params > 0)
      {
        printf("\tParameter List for %s\n",temp->lexeme);
        printf("\t( ");
        for(j=0; j < temp->num_params; j++)
        {
            printf("%s",temp->parameter_list[j]);
            printf("\t");
        }
        printf(" )\n");
      }

      temp = temp->next;
    }
}
  printf("----------------------------------------\n");
}

void DisplayConstant(entry** TablePointer)
{
  int i =0;
  entry *temp = NULL;
  printf("\n\n");
  printf("----------------------------------------\n");
  printf("\n\t(lexeme,   value, Data type, Line Number\n");
  for(i=0;i<SIZE;i++)
  {
    temp = TablePointer[i];
    while(temp != NULL)
    {
      printf("\t(%5s, %5f, %9s, %11d)\n"
     ,temp->lexeme,temp->value,temp->data_type,temp->line_number);
      temp = temp->next;
    }

  }

  printf("----------------------------------------\n");
}
```

The above code is the C code for tables.h which handles all functions relating to entries in the table. The functions used in the file are described in detail below -

1. **InsertEntry** is a function that inserts new variables and functions into the table. It checks for duplicate variables by carrying out a search and reports errors in case it finds a duplicate declaration.

2. Search Functions:

Tables.h employs two utility search functions based on different types of uses.

I. **Search** is a search function which searches for entries in Symbol Table based on the lexeme, it doesn't take any other criteria into consideration. This function is used to look for the most recent entry for the current lexeme, and is in employed in cases where only a simple lookup is required.

II. **InsertSearch** is a search function which searches for entries based on two criteria lexeme and current scope of the variable, this function is primarily used during declaration to check if any other variable with the same name exists in the current scope.

3. **funcSearch** is a function employed during declaration of functions, when a new function is declared we have to ensure that no other function with the same name exists in the program, and this function takes care of that, it searches the table for any other identifiers which might also be a function and then throws an error if the current declaration is a duplicate.

4. **searchFunc** is another utility function employed during function calls, it searches for any functions with the same name as the current function and enables appropriate actions in the caller function based on syntactical rules.

5. **set_is_function** is used to declare variables as functions if the declaration follows function declaration rules, it takes care of appropriate sanitization and sets is_function as 1 if the variable is a function.

6. **fill_parameter_list** is used to fill all the parameters for a declared function, this is done so that verification can be done at a later stage when the function is called. fill_parameter_list uses a linked list implementation to keep track of the declared parameters and then stores the linked list in the symbol table.

7. **check_parameter_list** is used to match the types of arguments and parameters. First the function matches the number of two and then the type.

Symbol table implementation

The above table contains all entries for variables and functions. The fields include name of the identifier, line number , boolean fields is_array , is_function and is_array (which are self explanatory) , value , data type , nesting level , array dimension , number of params and parameter list. The implementation uses a hash table with open chaining. The constant table also uses the same structure.

Test Cases

Without Errors

Serial Number	Test Case	Expected Output	Status

1.	`int func(int b){`
	`int a;`
	`return b;`
	`}`
	
	`void main(){`
	`int a;`
	`}`
		`Parsing complete.`
		
		`Symbol table`
		
		`----------`
		
		`(lexeme,value,Data type,Line Number,isArray,`
		`ArrayDimensions, isFunction, Nesting Level,`
		`num_params )`
		`( main, 2147483647.000000, VOID,`
		`6, 0, 0, 1,`
		`3, 0)`
		`( func, 2147483647.000000, INT,`
		`1, 0, 0, 1,`
		`1, 1)`
		`Parameter List for func`
		`( INT )`
		`( a, 2147483647.000000, INT,`
		`7, 0, 0, 0,`
		`4, 0)`
		`( a, 2147483647.000000, INT,`
		`2, 0, 0, 0,`
		`2, 0)`
		`( b, 2147483647.000000, INT,`
		`1, 0, 0, 0,`
		`2, 0)`
		`----------`
			PASS

2.	`#include <stdio.h>`
	
	`int abc(int gcd)`
	`{`
	`int a,b,c[10];`
	`b = a + c[1];`
	`return 1;`
	`}`
	`int main()`
	`{`
	`int i,j,k,n=12;`
	`for(i = 0; i < n;i = i + 1)`
	`j=j*2;`
	
	`for(k=0;k<5;k++){`
	`if(k==3){`
	`break;`
		`Parsing complete.`
		
		`Symbol table`
		
		`----------`
		
		`(lexeme,value,Data type,Line Number, isArray,`
		`ArrayDimensions, isFunction, Nesting Level,`
		`num_params )`
		`( gcd, 2147483647.000000, INT,`
		`3, 0, 0, 0,`
		`2, 0)`
		`( i, 2147483647.000000, INT,`
		`11, 0, 0, 0,`
		`4, 0)`
		`( j, 2147483647.000000, INT,`
		`11, 0, 0, 0,`
		`4, 0)`
		`( k, 2147483647.000000, INT,`
		`11, 0, 0, 0,`
		`4, 0)`
		`( n, 2147483647.000000, INT,`
		`11, 0, 0, 0,`
			PASS

	`}`
	`}`
	`int res = abc(i);`
	`res = abc(i);`
	`return res;`
	`}`
		4, 0)
		(main, 2147483647.000000, INT,
		9, 0, 0, 1,
		3, 0)
		(res, 2147483647.000000, INT,
		20, 0, 0, 0,
		8, 0)
		(abc, 2147483647.000000, INT,
		3, 0, 0, 1,
		1, 1)
		Parameter List for abc
		(INT)
		(a, 2147483647.000000, INT,
		5, 0, 0, 0,
		2, 0)
		(b, 2147483647.000000, INT,
		5, 0, 0, 0,
		2, 0)
		(c, 2147483647.000000, INT,
		5, 1, 10, 0,
		2, 0)
		--- 	
		
		Constant table
		
		--- 	
		
		(lexeme, value, Data type, Line Number
		(0, 0.000000, INT, 15)
		(0, 0.000000, INT, 12)
		(1, 1.000000, INT, 12)
		(1, 1.000000, INT, 7)
		(1, 1.000000, INT, 6)
		(2, 2.000000, INT, 13)
		(3, 3.000000, INT, 16)
		(5, 5.000000, INT, 15)
		(10, 10.000000, INT, 5)
		(12, 12.000000, INT, 11)
		--- 		

With Errors

Serial Number	Test Case	Expected Output	Status

1.	`void main(){`
	`int a;`
	`a = f(2,3);`
	`}`
		Line no: 5 Error message: Variable Not Declared Token: f
		
		Parsing failed.
			PASS

2.	`int a(int b, int c){`
	`return b;`
	`}`
	`int a(int d){`
	`return d;}`
		Line no: 7 Error message: Duplicate Functions Not allowed
		Token:)
		
		Parsing failed.
			PASS

3.	`int a(int b, int c){`
	`  return b;`
	`}`
	
	`void main(){`
	`  int c = a(3.5,2);`
	`}`
		Line no: 8 Error message: Parameter and argument types do not match Token:)
		
		Parsing failed.
			PASS

4.	`int main(){`
	`  return 3.5;`
	`}`
		Line no: 5 Error message: return type does not match function type Token: }
		
		Parsing failed.
			PASS

5.	`void main(){`
	`  continue;`
	`}`
		Line no: 4 Error message: Illegal use of continue Token: ;
		
		Parsing failed.
			PASS

6.	`int main()`
	`{`
	`  int a;`
	`  a = c + 1;`
	`  {`
	`  int b;`
	`  }`
	`  a = b;`
	`  int a;`
	`  int b[10];`
	`  return 1;`
	`}`
		Line no: 8 Error message: Variable Not Declared Token: c
		
		Parsing failed.
			PASS

Functionality and Implementation

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

1. Variable Declaration
   - Duplicate Declaration : The program inserts the variable name into the symbol table and then when a user declares the variable again in the same scope with same of different data type , the script does not let the user do so. When the InsertEntry function is called in the lexer_parser.y it

checks if the entry for the variable with same name and scope already exists. If it does the program throws an error stating *“Duplicate declaration of variable”.*

- Array size less than 1 : Array size of less than 1 is not permitted in C language. To handle this error , we make use of fields is_array and array_dim. During declaration if the subscript of the array is 0 we throw an error stating *“Arrays can't have dimension lesser than 1".* And if the elements of the array are being used in expression we check during parsing expressions if the subscript is less than 0 or greater than the dimensions using the condition *INT_CONSTANT* < *array_dim.* If the condition is not satisfied the program throws an exception *“Exceeds array dimensions”.*
- Variables with same name, different scopes :

2. Function Declarations

- Duplicate Declaration : The program as specified earlier inserts all identifiers in the lex file as it identifies them and calls InsertEntry. When the parser matches the declaration statement of function using the rule stated below it sets the variable is_function to 1 and does a check on existing declared function using set_is_function. In case it finds a function with the same name already declared it throws an exception *"Duplicate functions not allowed”.*

```
funDeclaration : typeSpecifier
                 identifier          {
                                     func_type = curr_data_type;
                                     is_declaration = 0;
                                     }
               '(' params ')'        {
                                     fill_parameter_list($2,param_list,p_idx);
                                     p_idx = 0;
                                     is_function = 1;
                                     int flag = set_is_function(SymbolTable,$2->lexeme);
                                     if(flag == 0){return -1;}
                                     p=1;
                                     }
               compoundStmt          {
                                     is_function = 0;
                                     if(!return_exists && strcmp(func_type,"VOID") != 0)
                                     {
                                     yyerror("This Function must have a return type");
                                     }
                                     };
```

- Parameters of type void : The function declaration rule as stated above calls a function fill_parameter_list which enters the list of parameters in the symbol table. In this function using the condition strcmp(list[i],”VOID”)!=0 the program makes sure that none of the parameters are void. The error is *"Parameters of type void not allowed”.*

- No functions defined : If a user makes a call to a function that has not yet been declared the program throws an exception stating *“No such declaration”.* Everytime the parser parses an identifier for a call it calls a function checkFunc() which looks for functions with the given name. In case it finds a function it checks for declarations of variables with same name. Since this would cause a conflict the program throws an error saying *"Defined as variable in this scope, calling not allowed" .*
- Return type mismatch : The program accounts for return type mismatches for the following cases-
  1. Return type expression and return type of function don’t match - This is handled by the condition *strcmp(func_type,expr_type)!=0.*
  2. Return type of function is not void and does not have a return statement - This is handled by setting the variable return_exists to 1 when a return statement is encountered. As it can be seen in the above rule after the matching of compoundStatement the if condition makes sure every function with a non-void return type and a matching return statement.

3. Call expressions :

- Use of undeclared function : This is handled using a checkFunc() which on reducing using the rule *call -> identifier ‘(‘ args ‘)* looks up in the symbol table a function with the same name as identifier. If it finds such a function parsing is continued else error *“No functions declared”* is thrown.
- Number of parameters and arguments do not match : The semantic analyser on reaching the rule *call -> identifier ‘(‘ args ‘)* calls the following semantic rules.

```
call : identifier '(' args ')'  {
                                    if(checkFunc($1->lexeme) == 0)
                                    {return -1;};
                                    $$ = $1->data_type;
                                    check_parameter_list($1,arg_list,a_idx);
                                    a_idx = 0;
                                }
args : argList | ;

argList : argList ',' arg | arg ;

arg : expression        {
                                    arg_list[a_idx] = (char *)malloc(sizeof($1));
                                    strcpy(arg_list[a_idx++],$1);
                        };
```

Since the symbol table already contains parameter types for a function saved , the program has to match the passed arguments in a call to those entries. The semantic rules save the arguments to an char * array arg_list and the number of these arguments is saved in the variable a_idx. When the closing ‘)’ is matched the function check_parameter_list is called which performs a check on number of parameters and a_idx. If the numbers don’t match the program throws an error *"Number of parameters and arguments do not match".*

- Type of arguments and parameters don't match : The semantic analyser on initialising arg_list performs a check on the number of arguments and parameters. If these two match the function check_arg_list moves on to matching corresponding entries of parameter_list and arg_list using the library strcmp function. If types don't match the exception "Types of arguments and parameters don't match" is thrown.

4. Select / while statements :

- Expression is of type int : To handle this exception we use the non-terminal simpleExpression which return only boolean value i.e 0 or 1.

5. Expressions :

- Identifier undeclared in current scope : Whenever identifiers occur in expressions a check for their existence in current scope is checked. This is done when the semantic analyser reaches the rule

  mutable -> identifier {checkScope(yylval.str); $$ = $1->data_type;}

  The checkScope function takes the value of the identifier and tries to find a variable with the same name in the symbol table. It matches only the variable with the same name that is also present in the same scope. If no such variable is present it throws an exception *"Variable out of scope".*
- Expression on lhs not a single variable : This condition is accounted for by the context - free grammar described in parser.y. It does not let an expression of the form a op b form on the lhs of an assignment statement.
- Type checking for expressions : The program does not allow operations on operators of different data types. Every production rule for expression parsing has a semantic rule that calls the function typeCheck which takes as arguments the type of operands on left and right of operator. The function does a strcmp on these arguments. The function returns 1 or throws an error saying *"Type mismatch".*

Results and Future Work

Results

We were able to successfully parse the tokens recognized by the flex and perform semantic analysis script for C. The output displays the set of semantic errors that we have covered along with structures that get accepted into the program.

Input 1: Sample program with no errors(covering most implemented cases)

```c
// This is a test file for the case when we have no errors
#include <stdio.h>

int abc(int gcd)
{
    int a,b,c[10];
    b = a + c[1];
    return 1;
}

int main()
{
    int i,j,n=12;
    for(i = 0; i < n;i = i + 1)
        j=j*2;
    int res = abc(i);
    res = abc(i);
    return res;
}
```

Output 1 :

```
Parsing complete.

        Symbol table

----------------------------------------

        (lexeme,            value, Data type, Line Number, isArray, ArrayDimensions, isFunction, Nesting Level, num_params
        (   gcd, 2147483647.000000,       INT,           5,       0,               0,          0,             2,          0)
        (     i, 2147483647.000000,       INT,          14,       0,               0,          0,             4,          0)
        (     j, 2147483647.000000,       INT,          14,       0,               0,          0,             4,          0)
        (     k, 2147483647.000000,       INT,          14,       0,               0,          0,             4,          0)
        (     n, 2147483647.000000,       INT,          14,       0,               0,          0,             4,          0)
        (  main, 2147483647.000000,       INT,          12,       0,               0,          1,             3,          0)
        (   res, 2147483647.000000,       INT,          23,       0,               0,          0,             8,          0)
        (   abc, 2147483647.000000,       INT,           5,       0,               0,          1,             1,          1)
        Parameter List for abc
        ( INT    )
        (     a, 2147483647.000000,       INT,           7,       0,               0,          0,             2,          0)
        (     b, 2147483647.000000,       INT,           7,       0,               0,          0,             2,          0)
        (     c, 2147483647.000000,       INT,           7,       1,              10,          0,             2,          0)
----------------------------------------

        Constant table

----------------------------------------

        (lexeme,    value, Data type, Line Number
        (    0, 0.000000,       INT,          18)
        (    0, 0.000000,       INT,          15)
        (    1, 1.000000,       INT,          15)
        (    1, 1.000000,       INT,           9)
        (    1, 1.000000,       INT,           8)
        (    2, 2.000000,       INT,          16)
        (    3, 3.000000,       INT,          19)
        (    5, 5.000000,       INT,          18)
        (   10, 10.000000,       INT,           7)
        (   12, 12.000000,       INT,          14)
----------------------------------------
```

The above screenshot displays symbol table and constant table after successful parsing of the input code. The symbol table has all fields as mentioned in design. The analyser successfully parses array declarations also.

Input 2: Test Case based on variable scopes

```c
#include <stdio.h>
int main()
{
    int a;
    // Undeclared Identifier
    a = c + 1;
    {
        int b;
    }
    // Variable out of scope
    a = b;
    // Duplicate variable declaration
    int a;
    // Duplicate variable new scope
    int b[10];
    return 1;
}
```

Output 2:

```
Line no: 8 Error message: Variable Not Declared Token: c

Parsing failed.
```

The above screenshot shows failed parsing because variable c is being used before it is declared.

Input 3: Test case for type matching errors

```c
#include <stdio.h>
// Test file to for matching types of variables in expression
void main()
{
    int a;
    float b;
    a = a + b;
    char s ;
    s = a;
}
```

Output 3:

```
Ayushs-MacBook-Air-2:Phase-3 ayush$ ./a.out testcases/test-typecheck.c
Line no: 9 Error message: Type Mismatch Token: ;
```

The above screenshot shows the error when the program tries to assign a char type to an int type.

Input 4: Duplicate Functions

```c
#include<stdio.h>

int a(int b, int c){
    return b;
}

int a(int d){
    return d;
}
```

Output 4:

```
Line no: 7 Error message: Duplicate Functions Not allowed
 Token: )

Parsing failed.
```

The above screenshot shows the error when the program tries to declare an existing function.

Input 5: Undeclared function

```c
#include<stdio.h>

void main(){
    int a;
    a = f(2,3);
}
```

Output 5:

```
Line no: 5 Error message: Variable Not Declared Token: f

Parsing failed.
```

The above screenshot shows the error when the program tries to access an undeclared function.

Input 6: Function call arguments mismatch

```c
#include<stdio.h>

int a(int b, int c){
    return b;
}

int a(int d){
    return d;
}
```

Output 6 :

```
Line no: 8 Error message: Parameter and argument types do not match Token: )
```

The above screenshot shows the error when the program calls the function with different argument types than it's parameters mentioned in the function's definition.

Input 7: Function return type mismatch

```c
#include<stdio.h>

int main(){
    return 3.5
}
```

Output 7:

```
Line no: 5 Error message: return type does not match function type Token: }
```

Above screenshot shows error when the program tries to return a float when function is of integer type

Input 8: Illegal use of loop control structures

```c
#include<stdio.h>

void main(){
    continue;
}
```

Output 8:

```
Line no: 4 Error message: Illegal use of continue Token: ;
```

The above screenshot shows the error when the program tries to use loop control structures outside of loops.

Future work

The yacc script we have used for the semantic analyser implements a subset of the C language, further work can be done to include more features, we had to implement a small subset because of time constraints.

More work can be done on lex script and yacc script to work with type casting. Further work under this project includes the implementation of Intermediate Code Generator.

References

1. https://www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/Ccfg.html
2. http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf
3. http://dinosaur.compilertools.net/yacc/