Compiler Design

Project Report - 3

Semantic analyser

By:

15CO248 - Jeshventh Raja T K

15CO229 - Zoeb Mithaiwala

CONTENTS

CONTENTS

Introduction

- Analysis phase
- Synthesis phase

Semantic analyser

Code - LEX

Code - PARSER

Explanation of implementation

Output and screenshots

- Test case 1
- Test case 2
- Test case 3
- Test case 4
- Test case 5

Introduction

What is a compiler?

A compiler is a special program that processes statements written in a high level programming language and turns them into machine language.

The input given to the compiler is a source code written in a high level language. The compiler outputs an equivalent and correct target code in machine level language. The compiler checks and outputs errors, if present. The compiler can check for syntax and semantic errors only.

There are two phases in compiling:

1) Analysis phase (Front end)
2) Synthesis phase (Back end)

Analysis phase

This phase consists of four steps:

1) Lexical analysis - Read and convert characters into words. Done by lexical/linear analyser or scanner
2) Syntax analysis - Convert the words into sentences. Done by syntax analyser or parser
3) Semantic analysis - Understand the meaning of sentences.
4) Intermediate code generation - Convert the understanding into intermediate code

Synthesis phase

This phase consists of two steps:

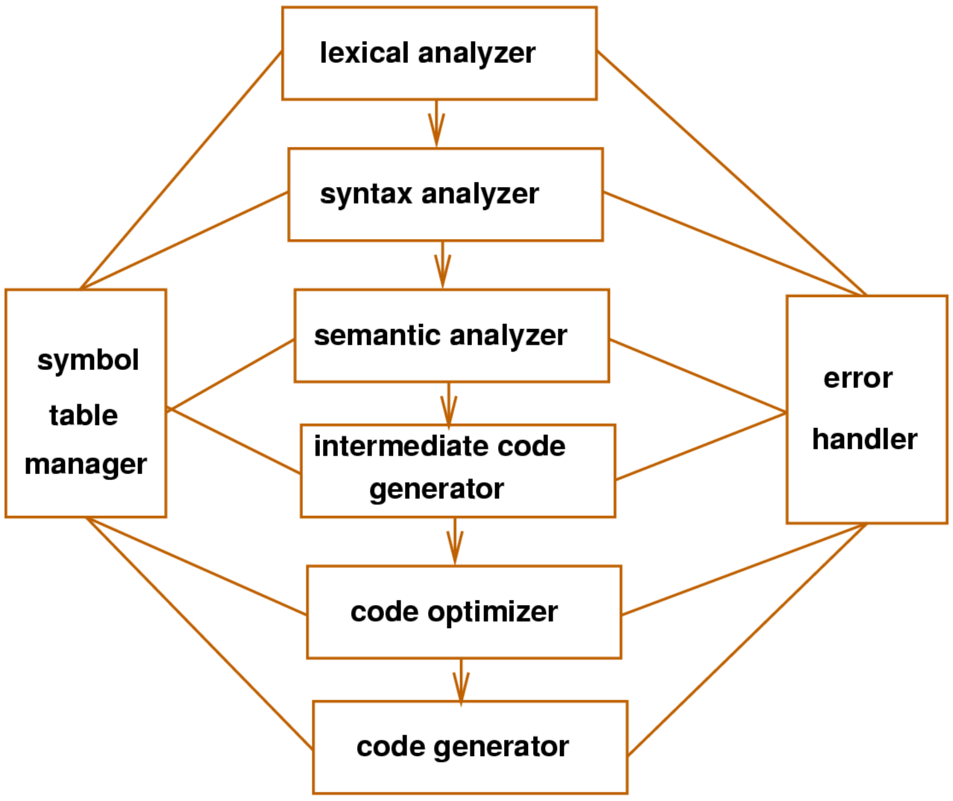
1) Code optimisation - Remove any unwanted or redundant code
2) Code generation - Convert to machine level language

There are various types of compilers:

1) Retargetable compiler - Converts H/L language to M/L language that is understood by a different machine
2) Single pass compiler - It scans an instruction, converts it to machine level code and then goes to the next instruction
3) Multipass compiler - It processes the source code or abstract syntax tree of a program several times, generating an output file after each pass

Additional functionalities of the compiler can be:

1) Error handling
2) Fast compilation
3) Efficient machine level code
4) Fast executing machine level output



Semantic analyser

Semantic analysis is the third phase in a compiler. 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 functions of a semantic analyser are:

1) Scope resolution
2) Check for declaration of variables
3) Type checking
4) Bounds checking

Attribute grammar

Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.

semantic errors

1) Semantic errors occur when the semantic rules are not satisfied

   Examples of errors

   1) Type mismatch
   2) Undeclared variable
   3) Reserved identifier misuse.
   4) Multiple declaration of variable in a scope.
   5) Accessing an out of scope variable.
   6) Actual and formal parameter mismatch.
   7) Out of bounds in arrays

Symbol table after semantic analysis

The symbol table consists of eight columns after the semantic analysis phase.

1) Index
2) Lexeme
3) Token type
4) Attribute
5) Scope
6) Procedure Definition Flag (for functions)
7) Nesting level
8) Parameter list (for functions)

It is made using a hash table. A hash is calculated on the lexeme and chaining is used to deal with collisions.

The first three columns of the symbol table is filled by the parser. The tokens are inserted into the table when the sequence of tokens are reduced to basic tokens. The remaining columns are filled by the semantic rules.

Code - LEX

```
%{
	#include <stdio.h>
	#include <stdlib.h>
	#include <string.h>
	#include "y.tab.h"
	int lineNo=1;
%}

keyword
auto|break|case|char|const|continue|default|do|double|else|enum|extern|floa
t|for|goto|if|int|long|register|return|short|signed|sizeof|static|struct|sw
itch|typedef|union|unsigned|void|volatile|while
singleLineComment \/\/.*
multilineComment "/*"([^*]|\*+[^*/])*\*+"/"
multilineCommentError "/*"([^*]|\*+[^*/])*
str \"[^\"]*\"
strError \"[^\"]*
letter [a-zA-Z]
digit [0-9]
nonIdentifier ({digit})+(_|{letter})(_|{letter}|{digit})*
identifier (_|{letter})(_|{letter}|{digit})*
dataType int|float|double|short\ int|long\ int
argument {dataType}[\ ]*{identifier}
integer ({digit})+
float ({digit})*\.({digit})+
assignmentOperator "="
arithmeticOperator "+"|"-"|"*"|"/"|"%"
operator
"+="|"-="|"*="|"/="|"%="|">>="|"<<="|"&="|"^="|"|="|"++"|"--"|"=="|"!="|">"
|"<"|">="|"<="|"||"|"&&"|"!"|"&"|"|"|"^"|"~"|"<<"|">>"|"?"
headerFile "#include"[\ ]*"<"[\ ]*[^>]*[\ ]*">"
specialCharacters \;|\{|\}|\(|\)|\[|\]|,
whitespaces [\ ]|[\t]
%%
{multilineCommentError} {printf("ERROR: Multiline comment is not
closed\n");}
{strError} {printf("ERROR: String is not closed\n");}
{singleLineComment} {}
{multilineComment} {}
```

```
{headerFile} {printf("Header file found\n"); return HEADERFILE;}
{str} {
            char *te = (char *)malloc(strlen(yytext)-1);
            int l = strlen(yytext);
            strncpy(te, yytext + 1, l-2);
            printf("%s- String constant\n", te);  yylval.sym=te; return
STR;}
auto {printf("Keyword auto found\n");  yylval.sym="auto"; return AUTO;}
break {printf("Keyword break found\n");  yylval.sym="break"; return BREAK;}
case {printf("Keyword case found\n");  yylval.sym="case"; return CASE;}
char {printf("Keyword char found\n");  yylval.sym="keyword"; return CHAR;}
const {printf("Keyword const found\n");  yylval.sym="const"; return CONST;}
continue {printf("Keyword continue found\n");  yylval.sym="continue";
yylval.sym=""; return CONTINUE;}
default {printf("Keyword default found\n");  yylval.sym="default"; return
DEFAULT;}
do {printf("Keyword do found\n");  yylval.sym="do"; return DO;}
double {printf("Keyword double found\n");  yylval.sym="double"; return
DOUBLE;}
else {printf("Keyword else found\n");  yylval.sym="else"; return ELSE;}
enum {printf("Keyword enum found\n");  yylval.sym="enum"; return ENUM;}
extern {printf("Keyword extern found\n");  yylval.sym="extern"; return
EXTERN;}
float {printf("Keyword float found\n");  yylval.sym="float"; return FLOAT;}
for {printf("Keyword for found\n");  yylval.sym="for"; return FOR;}
goto {printf("Keyword goto found\n");  yylval.sym="goto"; return GOTO;}
if {printf("Keyword if found\n");  yylval.sym="if"; return IF;}
int {printf("Keyword int found\n"); yylval.sym="int"; return INT;}
long {printf("Keyword long found\n");  yylval.sym="long"; return LONG;}
register {printf("Keyword register found\n");  yylval.sym="register";
return REGISTER;}
return {printf("Keyword return found\n");  yylval.sym="return"; return
RETURN;}
short {printf("Keyword short found\n");  yylval.sym="short"; return SHORT;}
signed {printf("Keyword signed found\n");  yylval.sym="signed"; return
SIGNED;}
sizeof {printf("Keyword sizeof found\n");  yylval.sym="sizeof"; return
SIZEOF;}
static {printf("Keyword static found\n");  yylval.sym="static"; return
STATIC;}
struct {printf("Keyword struct found\n");  yylval.sym="struct"; return
STRUCT;}
```

```
switch {printf("Keyword switch found\n");  yylval.sym="switch"; return
SWITCH;}
typedef {printf("Keyword typedef found\n");  yylval.sym="typedef"; return
TYPEDEF;}
union {printf("Keyword union found\n");  yylval.sym="union"; return UNION;}
unsigned {printf("Keyword unsigned found\n");  yylval.sym="unsigned";
return UNSIGNED;}
void {printf("Keyword void found\n");  yylval.sym="void"; return VOID;}
volatile {printf("Keyword volatile found\n");  yylval.sym=yytext; return
VOLATILE;}
while {printf("Keyword while found\n");  yylval.sym="while"; return WHILE;}
{identifier} {printf("%s is a Identifier\n", yytext);  yylval.sym=yytext;
return ID;}
{integer} {printf("%s is an Integer constant\n", yytext);
yylval.sym=yytext; return CONSTANT;}
{float} {printf("%s is a floating point constant\n", yytext);
yylval.sym=yytext; return CONSTANT;}
{nonIdentifier} {printf("ERROR: %s is an ill formed token\n", yytext);}

"+="  {printf("%s is addition assignment operator\n", yytext);  return
ADD_ASSIGN;}
"-="  {printf("%s is subtraction assignment operator\n", yytext);  return
SUB_ASSIGN;}
"*="  {printf("%s is multiplication assignment operator\n", yytext);
return MUL_ASSIGN;}
"/="  {printf("%s is division assignment operator\n", yytext);  return
DIV_ASSIGN;}
"%="  {printf("%s is mod assignment operator\n", yytext);  return
MOD_ASSIGN;}
">>=" {printf("%s is left shift assignment operator\n", yytext);  return
LEFT_ASSIGN;}
"<<=" {printf("%s is right shift assignment operator\n", yytext);  return
RIGHT_ASSIGN;}
"&="  {printf("%s is bitwise And assignment operator\n", yytext);  return
AND_ASSIGN;}
"^="  {printf("%s is bitwise Xor assignment operator\n", yytext);  return
XOR_ASSIGN;}
"|="  {printf("%s is bitwise Or assignment operator\n", yytext);  return
OR_ASSIGN;}
"++"  {printf("%s is increment operator\n",yytext);  return INC;}
"--"  {printf("%s is decrement operator\n", yytext);  return DEC;}
"=="  {printf("%s is equal comparator\n", yytext);  return EQ;}
```

```
"!="  {printf("%s is not equal comparator\n", yytext);  return 'NE';}
">"   {printf("%s is greater than comparator\n", yytext);  return '>';}
"<"   {printf("%s is less than comparator\n", yytext);  return '<';}
">="  {printf("%s is greater than equal\n", yytext);  return GE;}
"<="  {printf("%s is less than equal\n", yytext);  return LE;}
"||"  {printf("%s is logical or operator\n", yytext);  return OR;}
"&&"  {printf("%s is logical and operator\n", yytext);  return AND;}
"!"   {printf("%s is not operator\n", yytext);  return '!';}
"&"     {printf("%s is and operator\n", yytext);  return '&';}
"|"   {printf("%s is or operator\n", yytext);  return '|';}
"^"   {printf("%s is xor operator\n", yytext);  return '^';}
"~"   {printf("%s is bitwise not\n", yytext);  return '~';}
"<<"  {printf("%s is left shift operator\n", yytext);  return LEFT;}
">>"  {printf("%s is right shift operator\n", yytext);  return RIGHT;}
"?"   {printf("%s is conditional operator\n", yytext);  return '?';}
{assignmentOperator} {printf("%s is an assignment operator\n", yytext);
return '=';}
"-"   {printf("%s is arithmetic operator\n", yytext);  return '-';}
"+"   {printf("%s is arithmetic operator\n", yytext);  return '+';}
"*"   {printf("%s is arithmetic operator\n", yytext);  return '*';}
"/"   {printf("%s is arithmetic operator\n", yytext);  return '/';}
"%"   {printf("%s is arithmetic operator\n", yytext);  return '%';}
";" {printf("%s is special character\n",yytext); return(';');}
"{"   {printf("%s is special character\n", yytext);  return '{';}
"}"   {printf("%s is special character\n", yytext);  return '}';}
"("   {printf("%s is special character\n", yytext); return '(';}
")"   {printf("%s is special character\n", yytext); return ')';}
"["   {printf("%s is special character\n", yytext);  return '[';}
"]"   {printf("%s is special character\n", yytext);  return ']';}
","   {printf("%s is special character\n", yytext);  return ',';}
{whitespaces} {}
. {printf("Error: %s is an Illegal characters\n",yytext);}
"\n" {lineNo++;}
%%
```

Code - PARSER

```
%token ID CONSTANT STR HEADERFILE

%token INC DEC LE GE EQ NE
%token AND OR LEFT RIGHT MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN

%token AUTO BREAK CASE CHAR CONST CONTINUE DEFAULT DO DOUBLE ELSE ENUM
EXTERN FLOAT FOR GOTO IF INT LONG

%token REGISTER RETURN SHORT SIGNED SIZEOF STATIC STRUCT SWITCH TYPEDEF
UNION UNSIGNED VOID VOLATILE WHILE

%union{
    int integer;
    float floating_point;
    char *sym;
    char *val;
}

%start startSymbol

%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <limits.h>

    int size=20;
    char *alc=0;
    int globalIndex=0;
    int globalScope=1;
    int newScope=1;
    char *alc1=0;
    char *alc2=0;
    char *alc3=0;
    int setDatatype=1;
    int flag=0;
    int flag2=0;
    int procFlag=-1;
    int parameterNumber=0;
    int currentScope=1;
    int rhs=0;
    int whileStart=0;
    #define ANSI_COLOR_RED     "\x1b[31m"
    #define ANSI_COLOR_RESET   "\x1b[0m"

    int yyerror(char *msg);
    int yylex();

    struct Stack
    {
       int top;
       unsigned capacity;
       int* array;
    };

    struct Stack* createStack(unsigned capacity)
    {
      struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
      stack->capacity = capacity;
      stack->top = -1;
      stack->array = (int*) malloc(stack->capacity * sizeof(int));
      return stack;
    }

    int isFull(struct Stack* stack)
    {   return stack->top == stack->capacity - 1; }

    int isEmpty(struct Stack* stack)
    {   return stack->top == -1;  }

    void push(struct Stack* stack, int item)
    {
      if (isFull(stack))
            return;
      stack->array[++stack->top] = item;
    }

    int pop(struct Stack* stack)
```

```
{
  if (isEmpty(stack))
        return INT_MIN;
  return stack->array[stack->top--];
}

struct symbolTable{
   int index;
   char *symbol;
   char *attribute;
   char *data;
   struct symbolTable *next;
   int scope;
   int procedureDefinitionFlag;
   struct parameterChain *params;
   int nesting;
};

struct parameterChain{
   char *name;
   char *dataType;
   struct parameterChain *next;
};

struct symbolTable *hash[2][20];
struct Stack* stack;
struct Stack* stack1;
int i=0;

void init()
{
     stack = createStack(100);
     alc2 = (char*)malloc(sizeof(char)*100);
     alc3 = (char*)malloc(sizeof(char)*100);
     stack1 = createStack(100);
   for(;i<size;i++)
   {
         hash[0][i] = NULL;
         hash[1][i] = NULL;
   }
}
```

```
    int hashLocation(char *sym)
    {
       int sum=0;
       size_t length = strlen(sym);
       int k=0;
       for(;k<length;k++)
       {
             sum = sum + (int)sym[k];
       }
       return sum%size;
    }

    int searchHash(int type, char *sym, int x)
    {
       struct symbolTable *temp = hash[type][x];
       while(temp!=NULL)
       {
             if (strcmp(temp->symbol, sym)==0 && globalScope ==
temp->scope)
             {
                   return 0;
             }
             temp=temp->next;
       }
       return 1;
    }

    int searchMain()
    {
       struct symbolTable *temp = hash[0][421%size];
       while(temp!=NULL)
       {
             if (strcmp(temp->symbol, "main")==0 && temp->scope==0)
             {
                   return 1;
             }
             temp = temp->next;
       }
       return 0;
    }

    int getDatatype(char *sym)
    {
       int x = hashLocation(sym);
       int i=stack->top;
       for(;i>=0;i--)
       {
             struct symbolTable *temp = hash[0][x];
             while(temp!=NULL)
             {
             if(strcmp(temp->symbol, sym)==0 && temp->scope ==
stack->array[i])
             {
                   if (strcmp(temp->data, "float")==0)
                   return 2;
                   else if (strcmp(temp->data, "int")==0)
                   return 1;
                   else if(strcmp(temp->data, "void")==0)
                   return 0;
             }
             temp = temp->next;
             }
       }
       return -1;
    }

    void addToTable(int type, char *sym, char *attr, char *dat)
    {
       int x = hashLocation(sym);
       if (searchHash(type, sym, x)==0)
             return;
       struct symbolTable *newSymbol = (struct symbolTable
*)malloc(sizeof(struct symbolTable));
       char *te = (char *)malloc(strlen(sym)+1);
       strcpy(te, sym);
       newSymbol->symbol = te;
       newSymbol->attribute = attr;
       newSymbol->data = dat;
       newSymbol->next = NULL;
       newSymbol->index = globalIndex + 1;
       newSymbol->nesting = currentScope - 1;
       newSymbol->scope = globalScope - 1;
       newSymbol->params = NULL;
       newSymbol->procedureDefinitionFlag = procFlag;
```

```
        globalIndex++;
        struct symbolTable *temp = hash[type][x];
        hash[type][x] = newSymbol;
        hash[type][x]->next = temp;
    }

    void addParams(char *dataType, char *sym)
    {
        int x = hashLocation(alc2);
        struct symbolTable *temp = hash[0][x];
        while(strcmp(temp->symbol, alc2)!=0 )
        {
                temp=temp->next;
                if(temp==NULL)
                {
                        printf("\nTemp = null when searching for function name
in symbol table\n");
                        return;
                }
        }
        struct parameterChain *newParam = (struct
parameterChain*)malloc(sizeof(struct parameterChain));
        char *te = (char *)malloc(strlen(sym)+1);
        strcpy(te, sym);
        newParam->name = te;
        newParam->dataType = dataType;
        newParam->next=NULL;
        struct parameterChain *temp1 = temp->params;
        if(temp1==NULL)
        {
                temp->params=newParam;
                return;
        }
        while(temp1->next!=NULL)
        {
                temp1=temp1->next;
        }
        temp1->next=newParam;
    }

    int searchHashScope(char *sym, int x)
    {
        struct symbolTable *temp = hash[0][x];
        while(temp!=NULL)
        {
              if (strcmp(temp->symbol, sym)==0 && globalScope -1 ==
temp->scope)
              { return 0; }
              temp=temp->next;
        }
        return 1;
     }

     // This returns 0 if an identifier is already declared in the same
scope
     int checkDeclaration(char *sym)
     {
        int x = hashLocation(sym);
        if(searchHashScope(sym, x))
        return 1;
        else
        return 0;
     }

     char* getAttribute(char *sym)
     {
        char abc[20]="";
        char *ans = abc;
        int x = hashLocation(sym);
        struct symbolTable *temp = hash[0][x];
        while(temp!=NULL)
        {
              if (strcmp(temp->symbol, sym)==0 && 0 == temp->scope) //Need
to check if comparing scope with 0 is fine
              {
                    struct parameterChain *temp1 = temp->params;
                    while(temp1!=NULL)
                    {
                          //printf("\nParams are %s
%s\n",temp1->name,temp1->dataType);
                          temp1=temp1->next;
                    }
                    return temp->attribute;
              }
```

```
            temp=temp->next;
        }
        return ans;
    }

    int searchHashScope1(char *sym, int x)
    {
        struct symbolTable *temp = hash[0][x];
        while(temp!=NULL)
        {
            int i=0;
            if (strcmp(temp->symbol, sym)==0 && 0 == temp->scope)
            { return 0; }
            for (;i<=stack->top;i++)
            {
                if (strcmp(temp->symbol, sym)==0 && stack->array[i] ==
temp->scope)
                { return 0; }
            }
            temp=temp->next;
        }
        return 1;
    }

    int checkDeclaration1(char *sym)
    {
        int x = hashLocation(sym);
        if (searchHashScope1(sym, x))
        return 1;
        else
        return 0;
    }

    void checkParameterType()
    {
        int x = hashLocation(alc2);
        struct symbolTable *temp = hash[0][x];
        if(temp==NULL)
        {
            return;
        }
        while(strcmp(temp->symbol, alc2)!=0 || temp->scope!=0)
        {
                temp=temp->next;
                if(temp==NULL)
                {
                        printf("\nTemp = null when searching for function name
in symbol table\n");
                        return;
                }
        }
        struct parameterChain *temp1;
        temp1 = temp->params;
        int i;
        for(i=0;i<parameterNumber-1;i++)
        {
                temp1=temp1->next;
                if(temp1==NULL)
                {
                        if(flag==0)
                        printf(ANSI_COLOR_RED "ERROR: Arguments passes is more
than expected\n" ANSI_COLOR_RESET);
                        return;
                }

        }

        if(flag)
        {
                if(temp1->next!=NULL)
                {
                        printf(ANSI_COLOR_RED "ERROR: Arguments passes is less
than expected\n" ANSI_COLOR_RESET);
                }
                return;
        }

        x = hashLocation(alc3);
        i=stack->top;
        struct symbolTable *temp2;
        for(;i>=0;i--)
        {
                temp2 = hash[0][x];
                while(temp2!=NULL)
                {
                if(strcmp(temp2->symbol, alc3)==0 && temp2->scope ==
stack->array[i])
                {
                        i=-1;
                        break;
                }
                temp2 = temp2->next;
                }
        }

        if(strcmp(temp1->dataType,temp2->data)!=0)
        {
                printf(ANSI_COLOR_RED "ERROR: Argument is not of the correct
type\n" ANSI_COLOR_RESET);
        }
    }

    void display()
    {
        int k=0;
        printf("\n\nSYMBOL TABLE:\n");

printf("-------------------------------------------------------------------------
-------------------------------------------------------------\n");
        printf("%*s\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*s\n", 10,
"INDEX", 10, "SYMBOL", 10, "ATTRIBUTE", 10, "DATATYPE", 10, "SCOPE", 10,
"ProcDefFlag", 10, "NESTING");

printf("-------------------------------------------------------------------------
-------------------------------------------------------------\n");
        for(;k<size;k++)
        {
                struct symbolTable *temp = hash[0][k];
                while(temp!=NULL)
                {

printf("%*d\t|\t%*s\t|\t%*s\t|\t%*s\t|\t%*d\t|\t%*d\t|\t%*d\n",10,
temp->index, 10, temp->symbol, 10, temp->attribute, 10, temp->data, 10,
temp->scope, 10, temp->procedureDefinitionFlag,10,temp->nesting);
                        temp = temp->next;
                }
```

```
       }

printf("---------------------------------------------------------------------------
--------------------------------------------------------------\n");
       k=0;
       printf("\n\nCONSTANT TABLE:\n");

printf("---------------------------------------------------------------------------
\n");
       printf("%*s\t|\t%*s\t|\t%*s\n", 10, "INDEX", 10, "SYMBOL", 10, 
"ATTRIBUTE");

printf("---------------------------------------------------------------------------
\n");
       for(;k<size;k++)
       {
             struct symbolTable *temp = hash[1][k];
             while(temp!=NULL)
             {
                   printf("%*d\t|\t%*s\t|\t%*s\n",10, temp->index, 10, 
temp->symbol, 10, temp->attribute);
                   temp = temp->next;
             }
       }

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

%%

/*The supported datatypes*/
dataType : SHORT {alc="short";}
        | INT {alc="int";}
        | LONG {alc="long";}
        | FLOAT {alc="float";}
        | DOUBLE {alc="double";}
        ;

    /*List of all statements*/
statement : declarationStatement
```

```
          | ifAndElseMatched
          | ifAndElseUnmatched
          | whileLoopStatement
          | expressionStatement
          | functionCall
          | jumpStatement
          | compoundStatement
          | ';'
          ;

statement1 : declarationStatement
          | whileLoopStatement
          | expressionStatement
          | functionCall
          | jumpStatement
          | compoundStatement
          ;

statements : statements statement
             | statement
             ;


    /*defining each type of statements*/

    /*Declaration statements: Include declaration of identifiers, with or
without initialisation*/
declarationStatement : declarationList ';'
                          | functionDeclaration
                          ;
 /* For function definition */
declaration : dataType ID { int len = strlen(yylval.sym);
                                        char *buffer=(char *)malloc(len);
                                        int i;
                                        for (i=0;i<len-1;i++)
                                        {
                                              buffer[i] = yylval.sym[i];
                                              alc2[i]=buffer[i];
                                        }
                                        if(checkDeclaration(buffer))
                                              {
                                                    procFlag = 1;
addToTable(0,buffer,"function", alc);
                                                    procFlag = -1;
                                              }
                                              else
                                              {
                                                    printf(ANSI_COLOR_RED
"ERROR: Function is already declared\n" ANSI_COLOR_RESET);
                                              }
                                        alc1 = alc;
                                        }
         | void ID {
                                        int len = strlen(yylval.sym);
                                        int i;
                                        for (i=0;i<len;i++)
                                        {
                                              alc2[i]=yylval.sym[i];
                                        }
                                  if(checkDeclaration(alc2))
                                  {
                                        procFlag = 1;

addToTable(0,yylval.sym,"function", alc);
                                        procFlag = -1;
                                  }
                                  else
                                  {
                                        printf(ANSI_COLOR_RED "ERROR:
Function is already declared\n" ANSI_COLOR_RESET);
                                  }
                            alc1 = alc;
                            }
         ;

 /* For argument list */
declaration1 : dataType ID {if(checkDeclaration(yylval.sym))
                                              {

addToTable(0,yylval.sym,"identifier", alc);

addParams(alc,yylval.sym);
                                              }
```

```
                                                else
                                                {
                                                      printf(ANSI_COLOR_RED
"ERROR: Variable is already declared\n" ANSI_COLOR_RESET);
                                                }
                                    };

 /* For function declaration */
declaration2 : dataType ID { int len = strlen(yylval.sym);
                                          char *buffer=(char *)malloc(len);
                                          int i;
                                          for (i=0;i<len-1;i++)
                                          {
                                                buffer[i] = yylval.sym[i];
                                                alc2[i]=buffer[i];
                                          }
                                          if(checkDeclaration(buffer))
                                                {
                                                      procFlag = 0;

addToTable(0,buffer,"function", alc);
                                                      printf("\nadding

function to table\n");

                                                      procFlag = -1;
                                                }
                                                else
                                                {
                                                      printf(ANSI_COLOR_RED
"ERROR: Function is already declared\n" ANSI_COLOR_RESET);
                                                }
                                          }
             | void ID {
                                    int len = strlen(yylval.sym);
                                    int i;
                                    for (i=0;i<len;i++)
                                    {
                                          alc2[i]=yylval.sym[i];
                                    }
                                    if(checkDeclaration(yylval.sym))
                                    {
                                          procFlag = 0;
addToTable(0,yylval.sym,"function", alc);
                                          procFlag = -1;
                                          alc2=yylval.sym;
                                    }
                                    else
                                    {
                                          printf(ANSI_COLOR_RED "ERROR:
Function is already declared\n" ANSI_COLOR_RESET);
                                    }
                              }
             ;

declarationAndAssignment : declaration1 '=' consta
                              | declaration1 '=' stri
                              ;

completeDeclaration : declarationAndAssignment
                         | declaration1
                         ;

argumentList : argumentList ',' completeDeclaration
                   | completeDeclaration
                   ;

declarationList : dataType identi1 ',' identifierList
                   | dataType identi1
                   | dataType identi1 '=' consta
                   | dataType identi1 '=' identi
                   ;

identifierList : identifierList ',' identi1
                   | identifierList ',' identi1 '=' consta
                   | identifierList ',' identi1 '=' identi
                   | identi1
                   | identi1 '=' consta
                   | identi1 '=' identi
                   ;

/*declarationStatementError : declaration { printf("Semicolon missing after
statement %s\n",$1);}
                                | declarationAndAssignment {
printf("Semicolon missing after statement %s\n",$1);}
```

```
                                ;*/



    /*If and else statements.*/
    /*The dangling else problem is taken care by having two types of if and
else statements: Matched and unmatched statements*/
/*ifAndElseStatement : ifAndElseMatched
                        | ifAndElseUnmatched
                        ;*/
ifAndElseMatched : IF '(' expression ')' ifAndElseMatched1 ELSE
ifAndElseMatched1
                    ;
ifAndElseMatched1 : IF '(' expression ')' ifAndElseMatched1 ELSE
ifAndElseMatched1
                     | statement1
                     ;
ifAndElseUnmatched : IF '(' expression ')' ifAndElseMatched1 ELSE
ifAndElseUnmatched
                        | IF    '(' expression ')' statement1
                        ;
/*ifAndElseStatementError : IF '(' expression ifAndElseMatched ELSE
ifAndElseMatched
                               | IF expression ')' ifAndElseMatched ELSE
ifAndElseMatched
                               | IF expression ')' ifAndElseMatched ELSE
ifAndElseUnmatched
                               | IF '(' expression ifAndElseMatched ELSE
ifAndElseUnmatched
                               | IF expression ')' statement
                               | IF '(' expression statement
                               ;*/



    /*While loop*/
whileLoopStatement : WHILE whileParanthesisStart expression
whileParanthesisEnd statement
                               ;
whileParanthesisStart : '(' {whileStart=1;};
whileParanthesisEnd : ')' {whileStart=0;};
/*whileLoopStatementError : WHILE expression ')' statement
```

```
{addToTable(0,"while","keyword");}
                               | WHILE '(' expression statement
{addToTable(0,"while","keyword");}
                               ;*/


    /*Expressions are statements with operators. The expressions are
defined taking care of the precedence of operators*/
expressionStatement : expression ';'
                         ;
expression : assignment_expression
            | expression ',' assignment_expression
            ;
primary_expression
    : identi {int len = strlen(yylval.sym);
                   char *buffer=(char *)malloc(len);
                   int i;
                   for (i=0;i<len;i++)
                   {
                   if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                         buffer[i] = yylval.sym[i];
                   }
                   if(whileStart==1 && getDatatype(buffer)!=1)
printf(ANSI_COLOR_RED "\nERROR: Condition of while has to be int\n"
ANSI_COLOR_RESET);}
    | consta {if(rhs==0) printf(ANSI_COLOR_RED "\nERROR: Lvalue required to
be identifier\n" ANSI_COLOR_RESET);}
    | '(' expression ')'
    ;
postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' identi
    | postfix_expression INC
    | postfix_expression DEC
    ;

argument_expression_list
```

```
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC unary_expression
    | DEC unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' dataType ')'
    | SIZEOF '(' VOID ')'
    ;
unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;
cast_expression
    : unary_expression
    | '(' dataType ')' cast_expression
    ;
multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;
additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;
shift_expression
    : additive_expression
    | shift_expression LEFT additive_expression
    | shift_expression RIGHT additive_expression
    ;
relational_expression
```

```
    : shift_expression
    | relational_expression gt shift_expression
    | relational_expression lt shift_expression
    | relational_expression LE shift_expression
    | relational_expression GE shift_expression
    ;

gt : '>' {rhs=1;};
lt : '<' {rhs=1;};

equality_expression
    : relational_expression
    | equality_expression EQ relational_expression
    | equality_expression NE relational_expression
    ;
and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;
exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;
inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;
logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND inclusive_or_expression
    ;
logical_or_expression
    : logical_and_expression
    | logical_or_expression OR logical_and_expression
    ;
conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;
assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;
assignment_operator
    : '=' {rhs=1;}
    | MUL_ASSIGN {rhs=1;}
    | DIV_ASSIGN {rhs=1;}
    | MOD_ASSIGN {rhs=1;}
    | ADD_ASSIGN {rhs=1;}
    | SUB_ASSIGN {rhs=1;}
    | LEFT_ASSIGN {rhs=1;}
    | RIGHT_ASSIGN {rhs=1;}
    | AND_ASSIGN {rhs=1;}
    | XOR_ASSIGN {rhs=1;}
    | OR_ASSIGN {rhs=1;}
    ;


    /*Jump statements include continue,break and return statements*/
jumpStatement : CONTINUE ';'
              | BREAK ';'
              | RETURN ';' {
              if (alc1!=NULL && strcmp(alc1,"void")!=0)
printf(ANSI_COLOR_RED "\nERROR: Function type is %s return void found\n"
ANSI_COLOR_RESET, alc1);}
              | RETURN ID ';' {
                  int len = strlen(yylval.sym);
                  char *buffer=(char *)malloc(len);
                  int i;
                  for (i=0;i<len;i++)
                  {
                  if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                        buffer[i] = yylval.sym[i];
                  }
                  setDatatype = getDatatype(buffer);
                  if(setDatatype==1 && strcmp(alc1, "int"))
                        printf(ANSI_COLOR_RED "\nERROR: Function with
return type int returning void\n" ANSI_COLOR_RESET);
                  else if (setDatatype==0 && strcmp(alc1, "void"))
                        printf(ANSI_COLOR_RED "\nERROR: Function with
return type void returning int\n" ANSI_COLOR_RESET);
                  }
```

```
                    | RETURN consta ';' {
                    if (strcmp(alc1, "void")==0)
                          printf(ANSI_COLOR_RED "\nERROR: Function with
return type void returning value\n" ANSI_COLOR_RESET);
                    }
                | RETURN expression ';'
                ;
/*            jumpStatementError : CONTINUE
                     | BREAK
                     | RETURN
                     ;*/


    /*Compound statements are the statements enclosed within the curly
braces*/
compoundStatement : startCompound statements endCompound
                     | startCompound endCompound
                     ;

startCompound : '{' {push(stack, newScope++); globalScope = newScope;
                         push(stack1, currentScope++);
                         }
                ;
endCompound : '}' {pop(stack); globalScope =  stack->array[stack->top]+1;
                         if (stack->top == -1) globalScope = 1;
                         pop(stack1); currentScope =
stack1->array[stack1->top]+1;
                         if (stack1->top == -1) currentScope = 1;
                         }
              ;

/*compoundStatementError : '{'
                              | '}'
                              | '{' statements
                              | statements '}'
                              ;*/



startSymbol
    : external_declaration
    | startSymbol external_declaration
    ;

external_declaration
    : functionDefinition
    | declarationStatement
    | HEADERFILE
    ;

functionCall : identi3 '(' ')' ';'

             | identi3 '(' parameters ')' ';' {if(flag2==0){ flag=1;
checkParameterType(); flag=0; parameterNumber=0;}}
             ;

parameters : parameters ',' identi {parameterNumber++;
checkParameterType();}
                   | parameters ',' identi '=' stri {parameterNumber++;
checkParameterType();}
                   | parameters ',' identi '=' consta {parameterNumber++;
checkParameterType();}
                   | parameters ',' identi '=' identi {parameterNumber++;
checkParameterType();}
                   | parameters ',' consta {parameterNumber++;
checkParameterType();}
                   | parameters ',' stri {parameterNumber++;
checkParameterType();}
                   | identi { parameterNumber++; checkParameterType();}
                   | identi '=' stri {parameterNumber++;
checkParameterType();}
                   | identi '=' consta {parameterNumber++;
checkParameterType();}
                   | identi '=' identi {parameterNumber++;
checkParameterType();}
                   | consta {parameterNumber++; checkParameterType();}
                   | stri {parameterNumber++; checkParameterType();}
                   ;

functionDefinition : declaration '(' ')' compoundStatement
                        | declaration startParenthesis argumentList ')' '{'
statements endCompound {setDatatype = 0;}
                        | declaration startParenthesis argumentList ')' '{'
endCompound
```

```
                         ;

functionDeclaration : declaration2 '(' ')' ';'
                         | declaration2 startParenthesis argumentList ')'
';' {pop(stack);
                         globalScope = stack->array[stack->top]+1;
                         if (stack->top==-1)
                         {
                               globalScope = 1;
                         }
                         pop(stack1); currentScope =
stack1->array[stack1->top]+1;
                         if (stack1->top == -1) currentScope = 1;
                         }
                         ;

startParenthesis : '(' {push(stack,newScope++); globalScope = newScope;
                               push(stack1,currentScope++);}
                    ;

identi3 : ID {
                   int len = strlen(yylval.sym);
                   char *buffer=(char *)malloc(len);
                   int i;
                   for (i=0;i<len-1;i++)
                   {
                         if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                         {
                               buffer[i] = yylval.sym[i];
                               alc2[i]=buffer[i];
                         }
                   }
                   alc2[i]='\0';
                   if(checkDeclaration1(buffer)==0)
                   {
                         if(strcmp(getAttribute(buffer),"function")!=0)
                         {
                               printf(ANSI_COLOR_RED "\nERROR: %s is not a
function\n" ANSI_COLOR_RESET, buffer);
                         }
```

```
                        else
                        {
                        }
                }
                else
                {
                        printf(ANSI_COLOR_RED "\nERROR: Function used is
not declared\n" ANSI_COLOR_RESET);
                        flag2=1;
                }
        };

identi : ID {
                int len = strlen(yylval.sym);
                char *buffer=(char *)malloc(len);
                int i;
                for (i=0;i<len;i++)
                {
                if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                {
                        buffer[i] = yylval.sym[i];
                        alc3[i]=buffer[i];
                }
        }
        if(checkDeclaration1(buffer)==0)
                {
                }
        else
        {
                printf(ANSI_COLOR_RED "\nERROR: Variable used is not
declared\n" ANSI_COLOR_RESET);
        }
    };

identi1 : ID {
                int len = strlen(yylval.sym);
                char *buffer=(char *)malloc(len);
                int i;
                for (i=0;i<len;i++)
                {
                   if ((yylval.sym[i]>='a' && yylval.sym[i]<='z') ||
(yylval.sym[i]>='A' && yylval.sym[i]<='Z') || (yylval.sym[i]>='0' &&
yylval.sym[i]<='9') || (yylval.sym[i]=='_'))
                         buffer[i] = yylval.sym[i];
                   }

                   if(checkDeclaration(buffer))
                   {
                         addToTable(0,buffer,"identifier", alc);
                   }
                   else
                   {
                         printf(ANSI_COLOR_RED "ERROR: Variable is already
declared\n" ANSI_COLOR_RESET);
                   }
             };



consta : CONSTANT {addToTable(1,yylval.sym,"constant", "");};
stri : STR {addToTable(1,yylval.sym,"string", "");};
void : VOID {alc="void";};
%%

#include<stdio.h>
extern int lineNo;
int main()
{
    init();
    yyparse();
    if (searchMain()==0)
       printf(ANSI_COLOR_RED "ERROR: main function not present\n"
ANSI_COLOR_RESET);
    display();
}
int yywrap()
{
    return 1;
}
int yyerror(char *msg)
{
    printf("Error: %s in line %d\n",msg, lineNo);
```

```
}
```

Explanation of implementation

The lex code identifies tokens and returns them to the parser. The lex rules are written in the form of regular expressions.

The parser code takes the tokens from lexical analyser and matched them with the rules. The parser rules are written in the form of context free grammar.

The semantic code checks if the sentences formed follow the semantic rules. The semantic rules are written along with the syntax rules. They make use of global variables and symbol table columns like attribute, token type.

The semantic code contain the following main sections:

1) Scope resolution - Two variables: globalScope and newScope are used to identify the scope of any identifier which is encountered. The globalScope keeps track of current scope while the newScope keeps track of new scope number to be added. All the open scopes are stored in an array.
2) Attribute - The identifier also has a variable indicating if it is a function or not. This is found out using the grammar rules.
3) Checking for declaration of variable - The scope array which stores all the open scopes is used to check if the used variable is defined in the present or open scope.
4) Redeclaration of variable - The hash table is checked to see if the variable is already declared in the current scope.
5) Procedure definition flag - A function which is defined has a flag 1 while a function which is only declared has a flag 0. Non-function identifiers have a value of -1.
6) Parameter list - When grammar for functions is encountered, the parameters(name and datatype) following are stored under the function name
7) Arguments type checking - The parameter list is used to check if the type and number of arguments of a function are correct.

Output and screenshots

Our program identifies various types of tokens and displays them. Initially, it prints all the tokens as it encounters them. The syntax analyser calls the scanner to fetch tokens one by one and tries to match to the rules specified in the parser. The identifiers and constants are inserted into the symbol and constant table when encountered. In the end, every token is displayed in symbol and constants table. During the syntax phase, semantic rules are also checked. This inserts a few more columns in the table. We have made various test cases depicting various functionalities of our compiler.

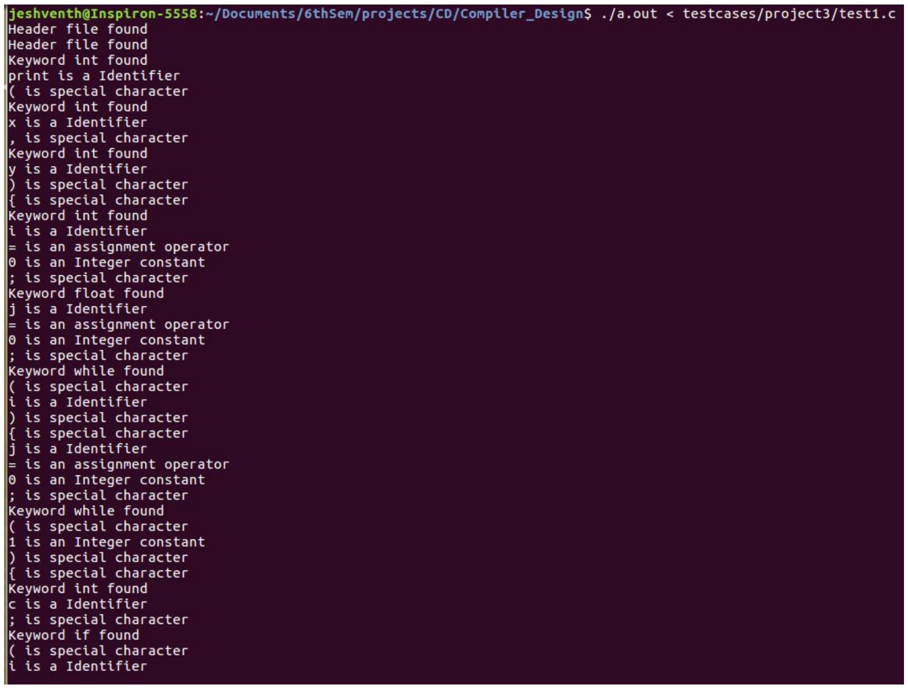
Test case 1

This test case doesn't have any lexical, syntax or semantics errors. This test case covers:

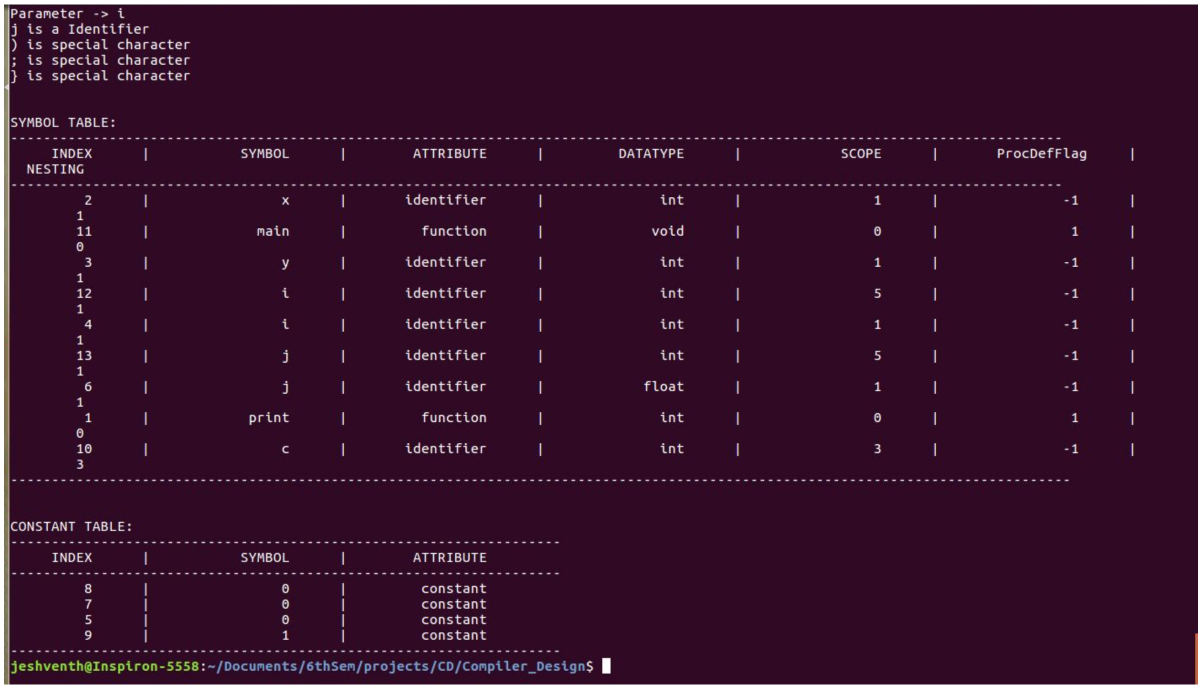
1) Header files
2) Function definition - procedure definition flag, function parameters
3) While loop - nested while loops
4) Declaration statement - no redeclaration or undeclared variable usage
5) Compound statement
6) If and else statement

```
#include <stdio.h>
#include <string.h>

int print(int x, int y)
{

    int i=0;
    float j=0;
    while(i)
    {
       j=0;
       while(1)
       {
             int c;
```

```
                if(i==j)
                {

                }

                j++;
        }
        i++;
    }
}

void main()
{
    int i,j;
    print(i,j);
}
```



```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project3/test1.c
Header file found
Header file found
Keyword int found
print is a Identifier
( is special character
Keyword int found
x is a Identifier
, is special character
Keyword int found
y is a Identifier
) is special character
{ is special character
Keyword int found
i is a Identifier
= is an assignment operator
0 is an Integer constant
; is special character
Keyword float found
j is a Identifier
= is an assignment operator
0 is an Integer constant
; is special character
Keyword while found
( is special character
i is a Identifier
) is special character
{ is special character
j is a Identifier
= is an assignment operator
0 is an Integer constant
; is special character
Keyword while found
( is special character
1 is an Integer constant
) is special character
{ is special character
Keyword int found
c is a Identifier
; is special character
Keyword if found
( is special character
i is a Identifier
```

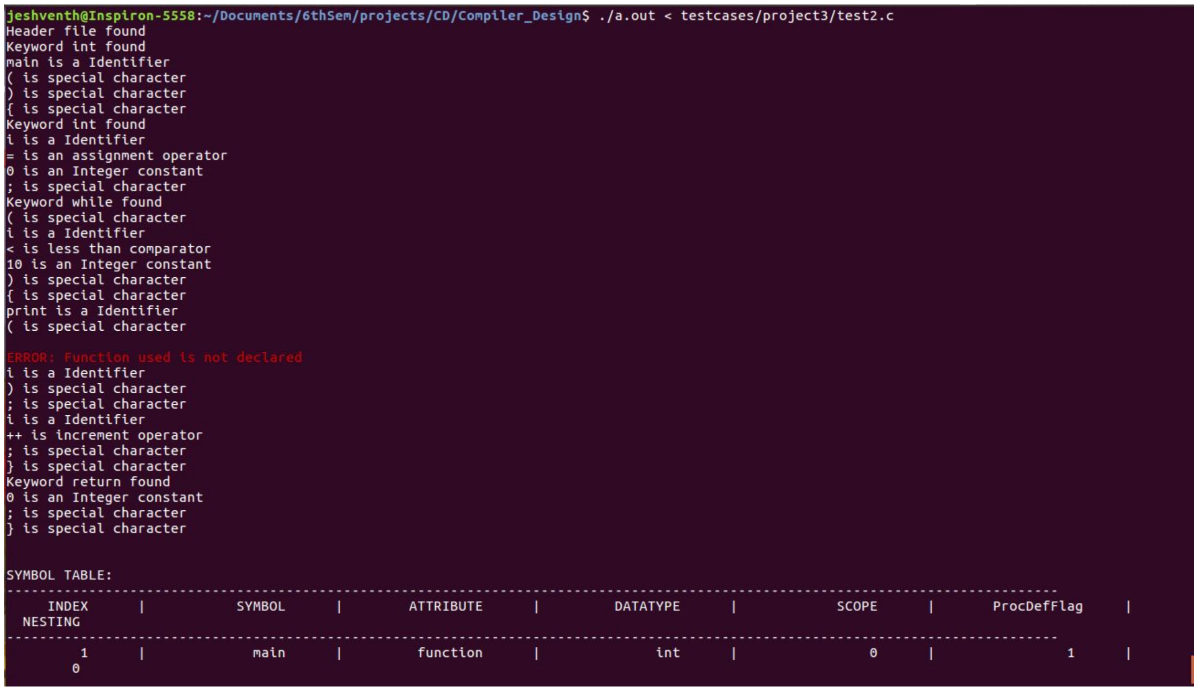


```
Parameter -> i
j is a Identifier
) is special character
; is special character
} is special character

SYMBOL TABLE:
------------------------------------------------------------------------------------------------------------------------------
     INDEX      |          SYMBOL       |      ATTRIBUTE       |        DATATYPE      |           SCOPE       |     ProcDefFlag      |
  NESTING
------------------------------------------------------------------------------------------------------------------------------
         2      |               x       |      identifier      |             int      |               1       |             -1       |
        1
        11      |            main       |       function       |            void      |               0       |              1       |
        0
         3      |               y       |      identifier      |             int      |               1       |             -1       |
        1
        12      |               i       |      identifier      |             int      |               5       |             -1       |
        1
         4      |               i       |      identifier      |             int      |               1       |             -1       |
        1
        13      |               j       |      identifier      |             int      |               5       |             -1       |
        1
         6      |               j       |      identifier      |           float      |               1       |             -1       |
        1
         1      |           print       |       function       |             int      |               0       |              1       |
        0
        10      |               c       |      identifier      |             int      |               3       |             -1       |
        3
-------------------------------------------------------------------------------------------------------------------------------


CONSTANT TABLE:
------------------------------------------------------------------
     INDEX      |          SYMBOL       |      ATTRIBUTE
------------------------------------------------------------------
         8      |               0       |       constant
         7      |               0       |       constant
         5      |               0       |       constant
         9      |               1       |       constant
------------------------------------------------------------------
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ 
```

Test case 2

The test case has one semantic error. The function print which is used in main is not defined.

```
#include <stdio.h>

int main()
{
    int i=0;
    while(i<10)
    {
       print(i);
       i++;
    }
    return 0;
}
```

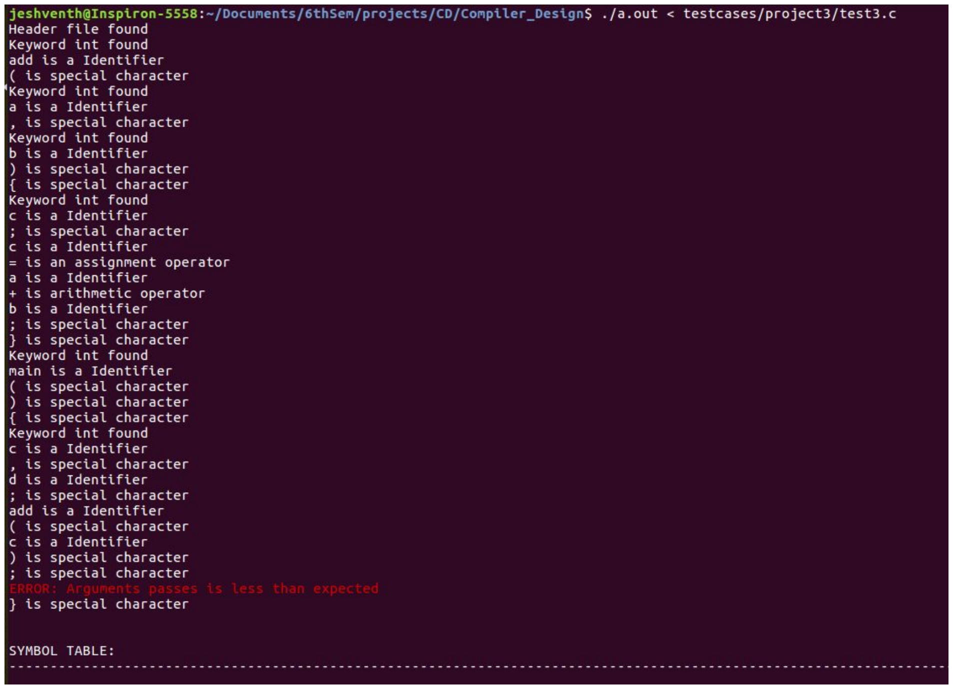


```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project3/test2.c
Header file found
Keyword int found
main is a Identifier
( is special character
) is special character
{ is special character
Keyword int found
i is a Identifier
= is an assignment operator
0 is an Integer constant
; is special character
Keyword while found
( is special character
i is a Identifier
< is less than comparator
10 is an Integer constant
) is special character
{ is special character
print is a Identifier
( is special character

ERROR: Function used is not declared
i is a Identifier
) is special character
; is special character
i is a Identifier
++ is increment operator
; is special character
} is special character
Keyword return found
0 is an Integer constant
; is special character
} is special character


SYMBOL TABLE:
------------------------------------------------------------------------------------------------------------------------
     INDEX      |          SYMBOL        |       ATTRIBUTE        |        DATATYPE        |          SCOPE         |       ProcDefFlag      |
  NESTING
------------------------------------------------------------------------------------------------------------------------
         1      |            main        |        function        |             int        |              0         |               1        |
        0
```

Test case 3

This test case two semantic errors. The first error is that the function add doesn't have a return statement although it is of type int. The second error is that function call has one less number of argument.

```
#include <stdio.h>

int add(int a, int b)
{
    int c;
    c=a+b;
}

int main()
{
    int c,d;
    add(c);
}
```

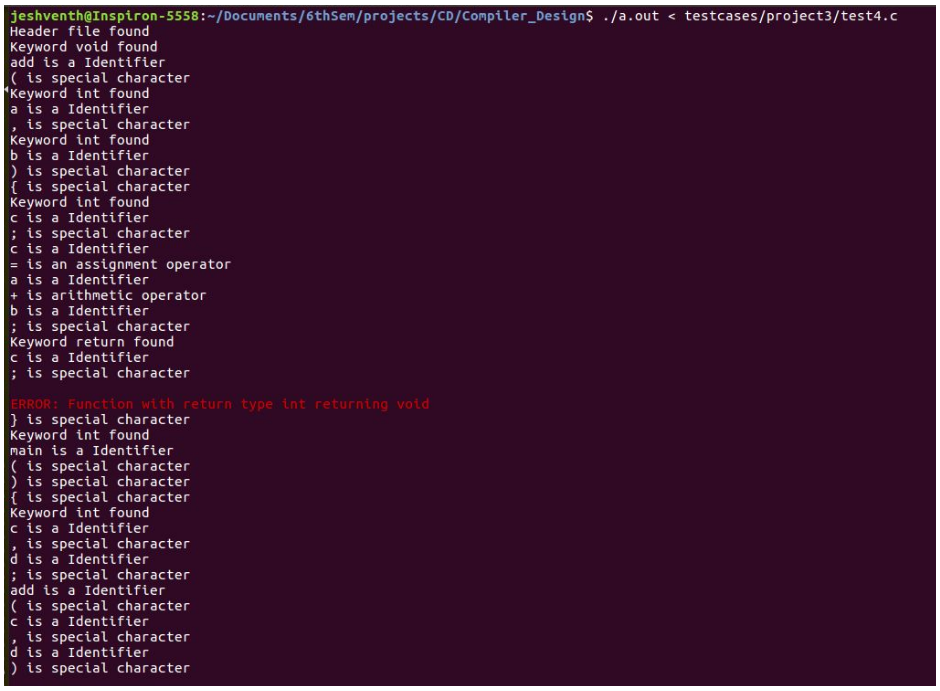


```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project3/test3.c
Header file found
Keyword int found
add is a Identifier
( is special character
Keyword int found
a is a Identifier
, is special character
Keyword int found
b is a Identifier
) is special character
{ is special character
Keyword int found
c is a Identifier
; is special character
c is a Identifier
= is an assignment operator
a is a Identifier
+ is arithmetic operator
b is a Identifier
; is special character
} is special character
Keyword int found
main is a Identifier
( is special character
) is special character
{ is special character
Keyword int found
c is a Identifier
, is special character
d is a Identifier
; is special character
add is a Identifier
( is special character
c is a Identifier
) is special character
; is special character
ERROR: Arguments passes is less than expected
} is special character


SYMBOL TABLE:
-----------------------------------------------------------------------------------------------------------------
```

Test case 4

This test case has a return statement although the function is of type void. Hence, this shows a semantic error.

```
#include <stdio.h>

void add(int a, int b)
{
    int c;
    c=a+b;
    return c;
}

int main()
{
    int c,d;
    add(c,d);
}
```

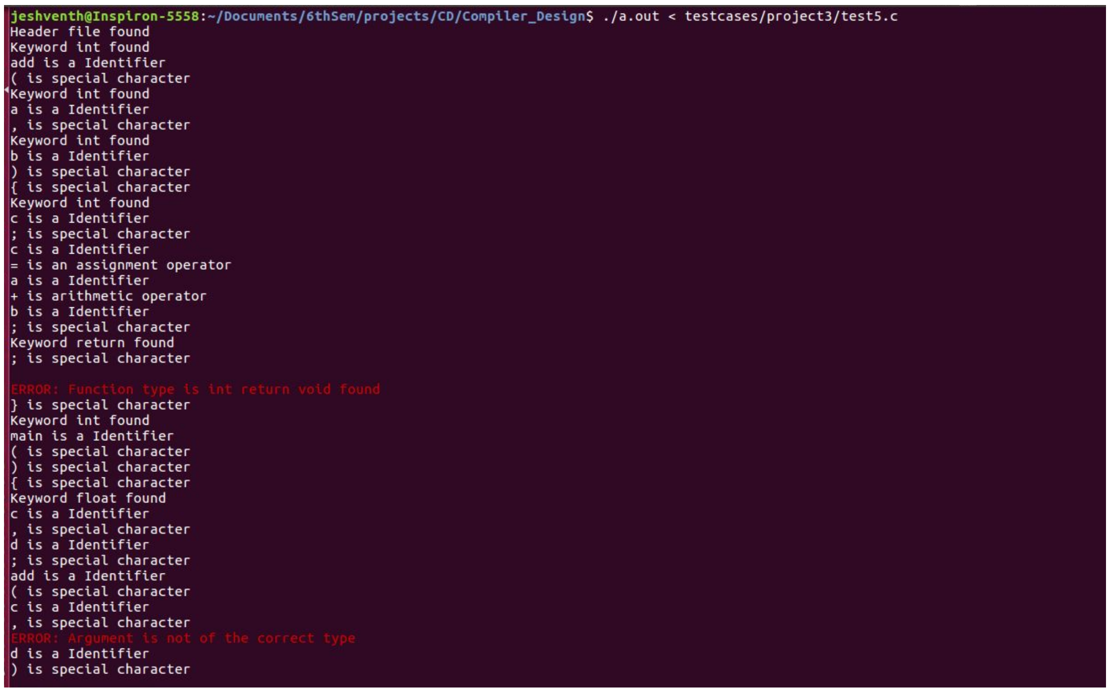


```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project3/test4.c
Header file found
Keyword void found
add is a Identifier
( is special character
Keyword int found
a is a Identifier
, is special character
Keyword int found
b is a Identifier
) is special character
{ is special character
Keyword int found
c is a Identifier
; is special character
c is a Identifier
= is an assignment operator
a is a Identifier
+ is arithmetic operator
b is a Identifier
; is special character
Keyword return found
c is a Identifier
; is special character

ERROR: Function with return type int returning void
} is special character
Keyword int found
main is a Identifier
( is special character
) is special character
{ is special character
Keyword int found
c is a Identifier
, is special character
d is a Identifier
; is special character
add is a Identifier
( is special character
c is a Identifier
, is special character
d is a Identifier
) is special character
```

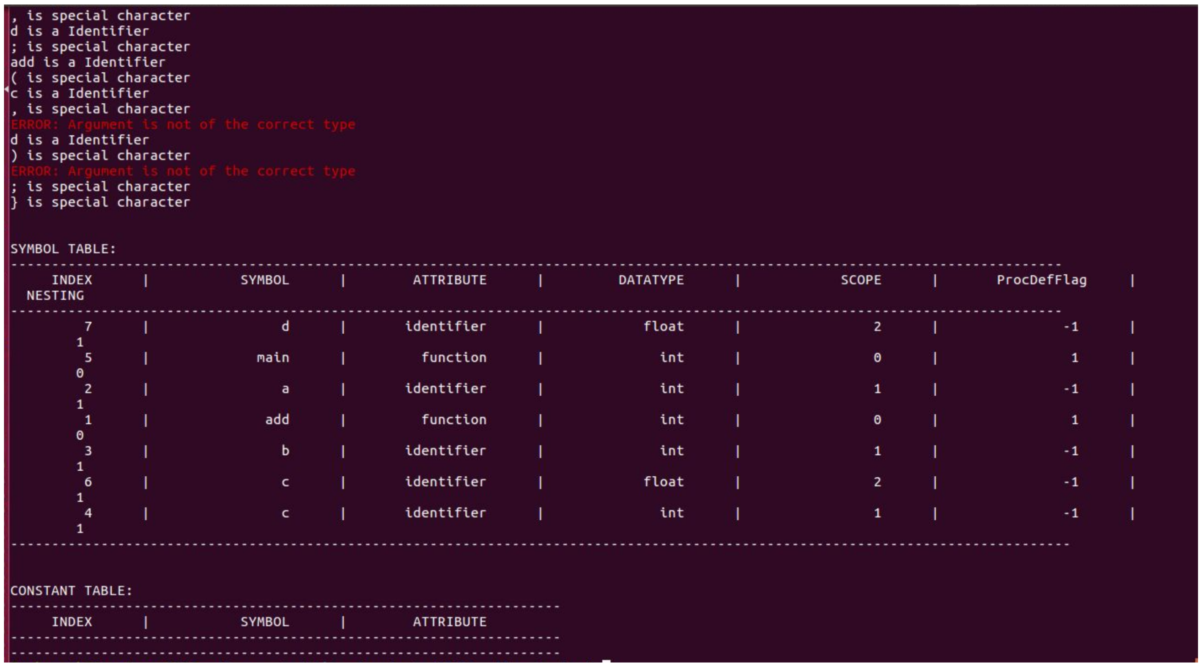
Test case 5

This test case has two semantic errors. The function add expects two int arguments but two float arguments are passed. And also a int type function has a void return.

```
#include <stdio.h>

int add( int a, int b)
{
    int c;
    c=a+b;
    return ;
}

int main()
{
    float c,d;
    add(c,d);
}
```



```
jeshventh@Inspiron-5558:~/Documents/6thSem/projects/CD/Compiler_Design$ ./a.out < testcases/project3/test5.c
Header file found
Keyword int found
add is a Identifier
( is special character
Keyword int found
a is a Identifier
, is special character
Keyword int found
b is a Identifier
) is special character
{ is special character
Keyword int found
c is a Identifier
; is special character
c is a Identifier
= is an assignment operator
a is a Identifier
+ is arithmetic operator
b is a Identifier
; is special character
Keyword return found
; is special character

ERROR: Function type is int return void found
} is special character
Keyword int found
main is a Identifier
( is special character
) is special character
{ is special character
Keyword float found
c is a Identifier
, is special character
d is a Identifier
; is special character
add is a Identifier
( is special character
c is a Identifier
, is special character
ERROR: Argument is not of the correct type
d is a Identifier
) is special character
```



```
, is special character
d is a Identifier
; is special character
add is a Identifier
( is special character
c is a Identifier
, is special character
ERROR: Argument is not of the correct type
d is a Identifier
) is special character
ERROR: Argument is not of the correct type
; is special character
} is special character


SYMBOL TABLE:
-------------------------------------------------------------------------------------------------------------------------------
     INDEX      |            SYMBOL       |          ATTRIBUTE      |          DATATYPE       |              SCOPE      |        ProcDefFlag      |
  NESTING
-------------------------------------------------------------------------------------------------------------------------------
         7      |                 d       |         identifier      |             float       |                  2      |                -1      |
        1
         5      |              main       |           function      |               int       |                  0      |                 1      |
        0
         2      |                 a       |         identifier      |               int       |                  1      |                -1      |
        1
         1      |               add       |           function      |               int       |                  0      |                 1      |
        0
         3      |                 b       |         identifier      |               int       |                  1      |                -1      |
        1
         6      |                 c       |         identifier      |             float       |                  2      |                -1      |
        1
         4      |                 c       |         identifier      |               int       |                  1      |                -1      |
        1
--------------------------------------------------------------------------------------------------------------------------------


CONSTANT TABLE:
-------------------------------------------------------------------
     INDEX      |            SYMBOL       |          ATTRIBUTE
-------------------------------------------------------------------
-------------------------------------------------------------------
```