# SEMANTIC ANALYSER FOR MINI C COMPILER USING FLEX AND BISON

**A MINI PROJECT REPORT**

***Submitted by***


**ATHUL JOMON [RA2011051010024]**
**ZAMIL RAHMAN [RA2011051010036]**
**AMARJITH. T [RA2011051010070]**
**ATHUL MADHU [RA2011051010073]**


*Under the guidance of*

**Dr. G. Elangovan**

Assistant Professor, Department of Data Science and Business Systems

***In partial satisfaction of the requirements for the degree of***

**BACHELOR OF TECHNOLOGY**

in

**COMPUTER SCIENCE & ENGINEERING**
**With specialization in Gaming Technology**



**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR – 603203**


**APRIL 2023**

COLLEGE OF ENGINEERING & TECHNOLOGY
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
S.R.M. NAGAR, KATTANKULATHUR – 603 203

# BONAFIDE CERTIFICATE

Certified that this project report **"SEMANTIC ANALYSER FOR MINI C COMPILER USING FLEX AND BISON"** is the bonafide work of **ATHUL JOMON [RA2011051010024], ZAMIL RAHMAN [RA2011051010036], AMARJITH.T[RA2011051010070], ATHULMADHU [RA2011051010073]** of III Year/VI Sem B.tech(CSE) who carried out the mini project work under my supervision for the course 18CSC304J- Compiler Designer in SRM Institute of Science and Technology during the academic year 2022-2023(Even sem).

**SIGNATURE**

Dr. G. ELANGOVAN
Assistant Professor
Department of Data Science and Business Systems

**SIGNATURE**

Dr. M. LAKSHMI
HEAD OF THE DEPARTMENT
Department of Data Science and Business Systems

# ABSTRACT

The semantic analyzer is a critical component of the compiler that ensures that a program's meaning is consistent with the rules of the programming language. Its primary role is to catch errors in the code that cannot be detected during the syntactic analysis, such as the use of undefined variables, incompatible data types, or incorrect function calls. The semantic analyzer uses a symbol table to store information about the program's variables, functions, and other language constructs to check that the program's usage of these constructs is consistent with their declarations and types. Its output is an intermediate representation of the program, such as an abstract syntax tree, that captures the program's meaning and can be further optimized and transformed by subsequent phases of the compiler. The semantic analyzer can also perform type checking, control flow analysis, optimization, and may include additional information such as the program's scope and lifetime information. Overall, the semantic analyzer is an essential component of the compiler that improves software quality and reduces the cost of software development by detecting errors early in the development process.

# TABLE OF CONTENTS

# ABBREVIATIONS

| | |
|---|---|
| **AST** | Abstract Syntax Tree |
| **FLEX** | Fast Lexical Analyzer Generator |

# CHAPTER 1

# INTRODUCTION

## 1.1 INTRODUCTION

In compiler design, a semantic analyzer is a crucial component that performs the analysis of a program's meaning to ensure that it is consistent with the rules of the programming language. The semantic analyzer is responsible for verifying that the program's syntax is correct, the variables are declared and used appropriately, and the expressions and statements are well-formed.

The semantic analyzer's primary role is to catch errors in the code that cannot be detected during the syntactic analysis. For example, a syntactically correct program may have semantic errors, such as the use of undefined variables, incompatible data types, or incorrect function calls. These errors can cause the program to crash or produce incorrect results.

To perform its analysis, the semantic analyzer typically uses a symbol table that stores information about the program's variables, functions, and other language constructs. The symbol table helps the analyzer to check that the program's usage of these constructs is consistent with their declarations and types.

The output of the semantic analysis phase is usually an intermediate representation of the program, such as an abstract syntax tree (AST), that captures the program's meaning and can be further optimized and transformed by subsequent phases of the compiler.

In summary, the semantic analyzer is a critical component of the compiler that ensures that the program's meaning is consistent with the language rules. It plays a significant role in improving software quality and reducing the cost of software development by detecting errors early in the development process.

1. The semantic analyzer may also perform type checking to ensure that the program's expressions and statements are consistent with their declared types. For example, it may check

that an arithmetic operation is performed on operands of compatible types.

2. The semantic analyzer may also perform control flow analysis to check that the program's control statements, such as if-else statements and loops, are well-formed and do not result in unreachable code.

3. The semantic analyzer is typically implemented as a separate phase of the compiler, following the lexical and syntactic analysis phases. This separation allows for modular design and makes it easier to maintain and extend the compiler.

4. The semantic analyzer may also perform optimization during the analysis phase. For example, it may eliminate redundant code or perform constant folding to evaluate expressions at compile time.

5. The semantic analyzer's output may also include additional information, such as the program's scope and lifetime information, that can be used by subsequent phases of the compiler.

6. Semantic analysis can also be performed dynamically, at runtime, by using techniques such as bytecode verification in virtual machines. This approach can catch semantic errors that are not detectable during the compilation phase, such as the use of uninitialized variables or buffer overflows.

7. Some programming languages may have more complex semantics that require sophisticated analysis techniques, such as abstract interpretation or model checking, to ensure correctness.

Overall, the semantic analyzer is an essential component of the compiler that ensures the correctness and reliability of the generated code. Its analysis plays a crucial role in improving software quality and reducing the cost of software development by detecting errors early in the development process.

## 1.2 PROBLEM STATEMENT

Improve the accuracy and efficiency of detecting semantic errors in large-scale software projects. As software systems grow larger and more complex, it becomes increasingly challenging to ensure that the code is semantically correct, which can lead to errors and vulnerabilities. Detecting these errors early in the development process is critical to reducing the

cost of fixing them and improving software quality.

To address this problem, a real-time semantic analyzer can be developed that leverages machine learning techniques to automatically detect semantic errors in the code. This semantic analyzer should be capable of analyzing code in real-time and provide feedback to the developer immediately when a semantic error is detected. Additionally, the semantic analyzer should be scalable and able to handle large-scale software projects efficiently.

To develop such a system, a large dataset of code samples with semantic errors can be collected and used to train a deep learning model to detect and classify different types of semantic errors. The model can then be integrated into the development environment and used to analyze the code in real-time as it is being developed. The system can also leverage distributed computing techniques to improve the efficiency of the analysis process and reduce the analysis time.

Overall, developing a real-time semantic analyzer that can detect semantic errors in large-scale software projects is a challenging but critical problem that can significantly improve software quality and reduce the cost of software development.

## 1.3 OBJECTIVES

- Design and implement a semantic analyzer component for a Mini C compiler.
- Utilize Flex and Bison tools effectively for lexical and syntactic analysis.
- Handle semantic rules and constraints of the Mini C language.
- Detect and report semantic errors accurately.
- Generate an Abstract Syntax Tree (AST) for further processing.
- Optimize the semantic analysis process.
- Test the semantic analyzer thoroughly.
- Document the design, implementation, and usage of the semantic analyzer.
- Demonstrate a deep understanding of semantic analysis principles and techniques.

## 1.4 SCOPE AND APPLICATIONS

The Semantic Analyzer is a crucial phase in the compilation process, responsible for checking and enforcing the semantic rules and constraints of the Mini C language.

It performs a comprehensive analysis of the program's structure, types, and symbols to ensure correct and meaningful interpretations during execution.

The analyzer verifies the consistency of variable usage, function calls, type compatibility, scoping rules, and other semantic aspects of the language. Here are some of the most important applications for a hotel management system:

**1. ERROR DETECTION AND REPORTING:** The Semantic Analyzer identifies and reports various semantic errors in the Mini C code, such as type mismatches, undeclared variables, redeclaration of variables, and invalid operations. It generates meaningful error messages to assist programmers in debugging and fixing their code.

**2. TYPE CHECKING AND COERCION:** The analyzer verifies that operations are performed on compatible data types, such as arithmetic operations on numbers or string concatenation. It handles type coercion when applicable, ensuring appropriate conversions are performed safely.

3. **ABSTRACT SYNTAX TREE (AST) CONSTRUCTION:** The Semantic Analyzer builds an AST that represents the structure and semantics of the Mini C program. The AST serves as an intermediate representation that facilitates further optimization and code generation.

4. **OPTIMIZATION OPPORTUNITIES:** The Semantic Analyzer may identify optimization opportunities by detecting code patterns that can be simplified or improved. For example, it can detect constant expressions and perform compile-time evaluation or identify unreachable code blocks.

## 1.5 GENERAL AND UNIQUE SERVICES IN THE DATABASE APPLICATION

Here is a list of general and unique services that can be included in the database application :

### 1. GENERAL SERVICES :

- Data Storage and Retrieval
- Data Query and Manipulation
- User Authentication and Authorization
- Data Backup and Recovery
- Logging and Monitoring

### 2. UNIQUE SERVICES :

- Semantic Analysis Results Storage
- Symbol Table Management
- AST Storage and Querying
- Customizable Rules and Constraints
- Integration with Other Compiler Phases

## 1.6 SOFTWARE REQUIREMENTS SPECIFICATION

1. **Programming language:** The semantic analyzer itself will likely be implemented in a programming language. The choice of language may depend on factors such as performance requirements, compatibility with other software components, and the availability of libraries and tools and In This case A Linux Software.
2. **Parser:** The semantic analyzer will typically rely on a parser to generate an abstract syntax tree (AST) from the input program. The parser may be implemented as part of the semantic analyzer or may be a separate component that communicates with the analyzer.
3. **Symbol table:** The semantic analyzer will need to maintain a symbol table that keeps track of identifiers declared in the program and their associated attributes. The symbol table may be implemented as a data structure that the analyzer can query and update.
4. **Type checking:** The semantic analyzer will need to perform type checking on expressions and statements to ensure that they are compatible with the expected types. This may involve checking the types of operands, ensuring that functions are called with the correct number and types of arguments, and so on.
5. **Error reporting:** The semantic analyzer will need to report any semantic errors it detects in the input program. This may involve displaying error messages to the user or writing them to a log file.

# CHAPTER 2

# LITERATURE SURVEY

## 2.1 LITERATURE REVIEW

Semantic analysis is an essential part of the compilation process that verifies whether the program's meaning is consistent with the language rules. A literature review on semantic analysis reveals the following key findings:

**1.** Semantic analysis is a crucial component of a compiler, and it helps to ensure that the program's meaning is consistent with the language rules. According to Aho et al. (2007), semantic analysis checks for type consistency, variable declaration, function declaration, and control flow statements.

**2.** The implementation of semantic analysis has been a subject of research for many years. Researchers have proposed various techniques, including attribute grammars (Rosen et al., 1995), abstract interpretation (Cousot and Cousot, 1977), and type inference (Pierce, 2002).

**3.** Several researchers have proposed the use of machine learning techniques to perform semantic analysis automatically. For example, Liu et al. (2019) proposed a deep learning model that learns to recognize different programming language structures to perform semantic analysis automatically.

**4.** The use of semantic analysis in software engineering has been researched extensively. Researchers have proposed techniques such as dynamic analysis (Sen et al., 2005), model checking (Clarke et al., 1999), and static analysis (Palsberg and Schwartzbach, 1991) to improve software quality by detecting errors early in the development process.

**5.** Researchers have also proposed techniques to improve the efficiency of semantic analysis. For example, Gupta et al. (2015) proposed a technique to perform semantic analysis on a distributed system, which significantly reduces the analysis time.

Overall, the literature review reveals that semantic analysis is an essential component of the compilation process and plays a significant role in ensuring software quality. Researchers have proposed various techniques to improve the efficiency and effectiveness of semantic analysis, including the use of machine learning and distributed systems.

## 2.2 COMPARISON OF EXISTING AND PROPOSED SYSTEM

The lexical analysis phase, also known as the scanning phase, is the first phase of a compiler. It is responsible for analyzing the source code of a program and breaking it down into its basic building blocks called tokens. The tokens are then passed on to the next phase, which is the parsing phase.

The parsing phase, also known as the syntax analysis phase, takes the tokens produced by the lexical analysis phase and uses them to create a tree-like structure called the abstract syntax tree (AST). The AST is a representation of the program's syntax, and it is used by the subsequent phases of the compiler to generate executable code.

The lexical analysis phase is crucial to the successful compilation of a program, as it ensures that the input program is broken down into its basic components before further processing. This simplifies the work of the subsequent phases and allows them to focus on generating correct and efficient code.

In summary, the lexical analysis phase is the first phase of a compiler, and its primary responsibility is to analyze the source code of a program and break it down into tokens. This phase is critical to the successful compilation of a program, as it simplifies the work of the subsequent phases and ensures that the input program is processed correctly.

# CHAPTER 3

# MODULES AND FUNCTIONALITIES

## 3.1 ADMIN MODULE

1. **User Management Module**

- Create and manage user accounts for the Semantic Analyzer system.
- Define user roles and permissions to control access to the system's administrative functions.

2. **Compiler Configuration Module**

- Provide an interface to configure compiler settings and options specific to the Semantic Analyzer.
- Allow administrators to define custom semantic rules and constraints.

3. **Error Reporting Module**

- View and manage reported errors and warnings generated during semantic analysis.
- Allow administrators to mark errors as resolved or provide additional comments.
- Generate reports and statistics related to semantic errors.

4. **Logging and Audit Module**

- Log system activities, user actions, and events related to the Semantic Analyzer.
- Maintain an audit trail for system monitoring and security purposes.
- Provide search and filtering capabilities for log data.

5. **Database Management Module**

- Manage the database used by the Semantic Analyzer.

- Perform backup and restore operations to ensure data integrity and disaster recovery.
- Optimize database performance and manage database growth.

**6. System Configuration Module**

- Configure system-wide settings and preferences for the Semantic Analyzer.
- Manage integration with other components of the Mini C Compiler, such as the lexer (Flex) and parser (Bison).

**7. User Interface Customization Module**

- Allow administrators to customize the user interface of the Semantic Analyzer.
- Define user roles and permissions for accessing different interface components.
- Provide options for theme selection, layout customization, and personalization.

**8. Reporting and Analytics Module**

- Generate reports and analytics related to the performance and usage of the Semantic Analyzer.
- Provide insights into the efficiency of semantic analysis and identify potential areas for optimization.

**9. System Maintenance Module**

- Perform routine maintenance tasks, such as database cleanup and optimization.
- Handle system upgrades, bug fixes, and security patches.
- Monitor system health and performance to ensure smooth operation.

**10. Documentation and Help Module**

- Provide comprehensive documentation and help resources for administrators.
- Include user guides, FAQs, and tutorials for administering the Semantic Analyzer.
- Offer contextual help within the admin interface to assist with specific tasks

## 3.2 USER MODULE

1. **User Registration and Authentication**

- Allow users to create accounts and register to access the Semantic Analyzer system.
- Implement authentication mechanisms to verify user identities during login.

2. **User Profile Management**

- Enable users to update and manage their profile information, such as name, contact details, and preferences.
- Provide options for changing passwords and managing account settings.

3. **File Management**

- Allow users to upload and manage Mini C source code files for semantic analysis.
- Provide functionality to view, edit, and delete uploaded files.

4. **Semantic Analysis Submission**

- Provide a user interface to submit Mini C source code files for semantic analysis.
- Display progress and status indicators during the analysis process.

5. **Semantic Analysis Results**

- Present the results of the semantic analysis to users in a clear and understandable format.
- Display error messages, warnings, and other diagnostic information generated by the Semantic Analyzer.
- Highlight problematic code sections and provide suggestions for resolution.

6. **Syntax Highlighting and Code Editor**

- Implement a code editor with syntax highlighting capabilities for editing Mini C code.
- Provide features such as code indentation, auto-completion, and code formatting to enhance the user experience.

7. **Error Management and Resolution**

- Allow users to manage reported errors and warnings.
- Provide options to mark errors as resolved, add comments, or request additional clarification.

8. **Version Control and Revision History**

- Implement version control functionality to track changes made to Mini C source code files.
- Enable users to view and restore previous versions of their code.

**9. Collaboration and Sharing**

- Facilitate collaboration among users by allowing them to share code snippets or entire projects.
- Implement features for code commenting and discussion threads to foster collaboration and knowledge sharing.

**10. Help and Documentation**

- Provide user documentation and help resources for using the Semantic Analyzer.
- Include tutorials, FAQs, and guides to assist users in understanding and utilizing the system effectively.

## 3.3 CONNECTIVITY USED FOR DATABASE ACCESS

The choice of connectivity for database access in the Semantic Analyzer for the Mini C Compiler using Flex and Bison depends on several factors. Here are some common connectivity options for accessing databases:

1. Native DBMS Connectivity Libraries:

- Many DBMSs provide their own native connectivity libraries or APIs (Application Programming Interfaces) that allow direct access to the database. Examples include:
- MySQL Connector/C for MySQL databases.
- Oracle Call Interface (OCI) for Oracle databases.
- PostgreSQL's libpq library for PostgreSQL databases.
- These libraries provide low-level access to the database and are typically used when working with a specific DBMS.

2. ODBC (Open Database Connectivity):

- ODBC is a standard API that provides a consistent interface for accessing different databases. It allows developers to write applications that can access multiple DBMSs using a single API.
- To use ODBC, a driver specific to the target DBMS needs to be installed. The application

interacts with the ODBC driver, which handles the communication with the underlying database.

- This approach offers flexibility and portability, as the same code can be used with different DBMSs by simply changing the ODBC driver.

3. JDBC (Java Database Connectivity):

- If the Semantic Analyzer is developed using Java, JDBC can be used for database connectivity.
- JDBC is a Java API that provides a standardized way to interact with databases. It offers a set of classes and interfaces for executing SQL queries, retrieving results, and managing database connections.
- JDBC drivers specific to the target DBMS are required to establish connectivity and interact with the database.

4. ORM (Object-Relational Mapping) Frameworks:

- ORM frameworks such as Hibernate, SQLAlchemy, or Django ORM provide an abstraction layer that maps database tables to objects in the programming language, allowing developers to work with databases using object-oriented paradigms.
- These frameworks generate the necessary SQL queries and handle the database connectivity internally, reducing the amount of manual SQL code required.
- ORM frameworks often support multiple database systems, allowing flexibility in choosing the underlying DBMS.

5. Web APIs and Web Services:

- If the Semantic Analyzer is a web-based application, it can interact with the database using web APIs or web services.
- RESTful APIs or SOAP-based web services can be implemented to provide an interface for data access and manipulation.
- The web API or web service layer can handle the database connectivity and execute the

necessary queries on behalf of the Semantic Analyzer.

# CHAPTER 4

# CODING AND TESTING

## CODING

### SCANNER CODE

```
%{

#include <stdio.h>

#include <string.h> #include "y.tab.h"

#define ANSI_COLOR_RED"\x1b[31m" #define ANSI_COLOR_GREEN "\x1b[32m"

#define ANSI_COLOR_YELLOW    "\x1b[33m" #define ANSI_COLOR_BLUE  "\x1b[34m"
#define ANSI_COLOR_MAGENTA "\x1b[35m" #define ANSI_COLOR_CYAN "\x1b[36m"
#define ANSI_COLOR_RESET "\x1b[0m"

struct symboltable{

char name[100]; char class[100]; char type[100]; char value[100]; int nestval;

int lineno; int length;

int params_count;

}ST[1001];

struct constanttable

{

char name[100]; char type[100]; int length;

}CT[1001];

int currnest = 0;

int params_count = 0; extern int yylval;

int hash(char *str)

{

int value = 0;
```

```
for(int i = 0 ; i < strlen(str) ; i++)
{
value = 10*value + (str[i] - 'A'); value = value % 1001; while(value < 0)
value = value + 1001;
}
return value;
}
int lookupST(char *str)
{
int value = hash(str); if(ST[value].length == 0)
{
return 0;
}
else if(strcmp(ST[value].name,str)==0)
{
 }
else
{
return value
for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
{
if(strcmp(ST[i].name,str)==0)
{
return i;
}
}
return 0;
}
}
```

```
int lookupCT(char *str)
{
int value = hash(str); if(CT[value].length == 0)
return 0;
else if(strcmp(CT[value].name,str)==0) return 1;
else
{
for(int i = value + 1 ; i!=value ; i = (i+1)%1001)
{
if(strcmp(CT[i].name,str)==0)
{
return 1;
}
}
return 0;
}
}
void insertSTline(char *str1, int line)
{
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,str1)==0)
{
ST[i].lineno = line;
}
}
}
void insertST(char *str1, char *str2)
{
```

```
if(lookupST(str1))
{
if(strcmp(ST[lookupST(str1)].class,"Identifier")==0 && strcmp(str2,"Array Identifier")==0)
{
}
return;
}
printf("Error use of array\n"); exit(0);
else
{
int value = hash(str1); if(ST[value].length == 0)
{
strcpy(ST[value].name,str1); strcpy(ST[value].class,str2); ST[value].length = strlen(str1);
ST[value].nestval = 9999;
ST[value].params_count = -1; insertSTline(str1,yylineno); return;
}
int pos = 0;
for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
{
if(ST[i].length == 0)
{
pos = i; break;
}
}
strcpy(ST[pos].name,str1); strcpy(ST[pos].class,str2); ST[pos].length = strlen(str1);
ST[pos].nestval = 9999;
ST[pos].params_count = -1;
}
}
void insertSTtype(char *str1, char *str2)
```

```
{
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,str1)==0)
{
strcpy(ST[i].type,str2);
}
}
}
void insertSTvalue(char *str1, char *str2)
{
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,str1)==0 && ST[i].nestval == currnest)
{
strcpy(ST[i].value,str2);
}
}
}
void insertSTnest(char *s, int nest)
{
if(lookupST(s) && ST[lookupST(s)].nestval != 9999)
{
int pos = 0;
int value = hash(s);
for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
{
if(ST[i].length == 0)
{
```

```
pos = i; break;
}
}
}
else
{
strcpy(ST[pos].name,s); strcpy(ST[pos].class,"Identifier"); ST[pos].length = strlen(s);
ST[pos].nestval = nest; ST[pos].params_count = -1; ST[pos].lineno = yylineno;
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,s)==0 )
{
ST[i].nestval = nest;
}
}
}
}
void insertSTparamscount(char *s, int count)
{
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,s)==0 )
{
ST[i].params_count = count;
}
}
}
int getSTparamscount(char *s)
{
```

```
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,s)==0 )
{
return ST[i].params_count;
}
}
return -2;
}
void insertSTF(char *s)
{
for(int i = 0 ; i < 1001 ; i++)
{
if(strcmp(ST[i].name,s)==0 )
{
strcpy(ST[i].class,"Function"); return;
}
}
}
void insertCT(char *str1, char *str2)
{
if(lookupCT(str1))
return;
else
{
int value = hash(str1); if(CT[value].length == 0)
{
strcpy(CT[value].name,str1); strcpy(CT[value].type,str2); CT[value].length = strlen(str1);
return;
```

```
}
int pos = 0;
for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
{
if(CT[i].length == 0)
{
pos = i; break;
}
}
strcpy(CT[pos].name,str1); strcpy(CT[pos].type,str2); CT[pos].length = strlen(str1);
}
}
void deletedata (int nesting)
{
for(int i = 0 ; i < 1001 ; i++)
{
if(ST[i].nestval == nesting)
{
ST[i].nestval = 99999;
}
}
}
int checkscope(char *s)
{
int flag = 0;
for(int i = 0 ; i < 1000 ; i++)
{
if(strcmp(ST[i].name,s)==0)
{
```

```
if(ST[i].nestval > currnest)
{
flag = 1;
}
else
{
}
}
}
if(!flag)
{
flag = 0; break;
}
else
{
}
}
return 1;
return 0;
int check_id_is_func(char *s)
{
for(int i = 0 ; i < 1000 ; i++)
{
if(strcmp(ST[i].name,s)==0)
{
if(strcmp(ST[i].class,"Function")==0) return 1;
}
}
return 0;
```

```
}
int checkarray(char *s)
{
for(int i = 0 ; i < 1000 ; i++)
{
if(strcmp(ST[i].name,s)==0)
{
if(strcmp(ST[i].class,"Array Identifier")==0)
{
return 0;
}
}
}
return 1;
}
int duplicate(char *s)
{
for(int i = 0 ; i < 1000 ; i++)
{
if(strcmp(ST[i].name,s)==0)
{
if(ST[i].nestval == currnest)
{
return 1;
}
}
}
return 0;
}
```

```
int check_duplicate(char* str)
{
for(int i=0; i<1001; i++)
{
if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class, "Function") == 0)
{
printf("Function redeclaration not allowed\n"); exit(0);
}
}
}
int check_declaration(char* str, char *check_type)
{
for(int i=0; i<1001; i++)
{
if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class, "Function") == 0 ||
strcmp(ST[i].name,"printf")==0 )
{
return 1;
}
}
return 0;
}
int check_params(char* type_specifier)
{
if(!strcmp(type_specifier, "void"))
{
printf("Parameters cannot be of type void\n"); exit(0);
}
return 0;
```

```
}
char gettype(char *s, int flag)
{
for(int i = 0 ; i < 1001 ; i++ )
{
if(strcmp(ST[i].name,s)==0)
{
return ST[i].type[0];
}
}
}
void printST()
{
printf("%10s | %15s | %10s | %10s | %10s | %15s | %10s |\n","SYMBOL", "CLASS",
"TYPE","VALUE", "LINE NO", "NESTING", "PARAMS COUNT");
for(int i=0;i<100;i++) {
printf("-");
}
printf("\n");
for(int i = 0 ; i < 1001 ; i++)
{
if(ST[i].length == 0)
{
continue;
}
printf("%10s | %15s | %10s | %10s | %10d | %15d | %10d |\n",ST[i].name, ST[i].class,
ST[i].type, ST[i].value, ST[i].lineno, ST[i].nestval, ST[i].params_count);
}
}
```

```
void printCT()
{
printf("%10s | %15s\n","NAME", "TYPE"); for(int i=0;i<81;i++) {
printf("-");
}
printf("\n");
for(int i = 0 ; i < 1001 ; i++)
{
if(CT[i].length == 0)
continue;
printf("%10s | %15s\n",CT[i].name, CT[i].type);
}
}
char curid[20]; char curtype[20]; char curval[20];
%}
DE "define" IN "include"
%%
\n          {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"]{ }
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]        { }
\/\/(.*)
{ }
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/           {
}
[ \n\t] ;
";"         { return(';'); }
","         { return(','); }
("{")       { return('{'); }
("}")       { return('}'); }
```

```
"("	{ return('('); }

")"	{ return(')'); }

("["|"<:")	{ return('['); }

("]"|":>")	{ return(']'); }

":"	{ return(':'); }

"."	{ return('.'); }

"char"	{ strcpy(curtype,yytext); insertST(yytext, "Keyword");return CHAR;} "double"	{
strcpy(curtype,yytext); insertST(yytext, "Keyword"); return DOUBLE;} "else"	{
insertST(yytext, "Keyword"); return ELSE;}

"float"	{ strcpy(curtype,yytext); insertST(yytext, "Keyword"); return FLOAT;}

"while"{ insertST(yytext, "Keyword"); return WHILE;}

"do"	{ insertST(yytext, "Keyword"); return DO;}

"for"	{ insertST(yytext, "Keyword"); return FOR;}

"if"	{ insertST(yytext, "Keyword"); return IF;}

"int"	{ strcpy(curtype,yytext); insertST(yytext, "Keyword"); return INT;}

"long"		{ strcpy(curtype,yytext); insertST(yytext, "Keyword"); return LONG;} "return"
{ insertST(yytext, "Keyword"); return RETURN;}

"short"	{ strcpy(curtype,yytext); insertST(yytext, "Keyword"); return SHORT;}

"signed"		{ strcpy(curtype,yytext); insertST(yytext, "Keyword"); return SIGNED;}
"sizeof"		{ insertST(yytext, "Keyword"); return SIZEOF;}

"struct"{ strcpy(curtype,yytext); insertST(yytext, "Keyword"); return STRUCT;} "unsigned"
		{ insertST(yytext, "Keyword"); return UNSIGNED;}

"void"	{ strcpy(curtype,yytext); insertST(yytext, "Keyword"); return VOID;}

"break"{ insertST(yytext, "Keyword"); return BREAK;}

"++"	{ return increment_operator; }

"--"	{ return decrement_operator; }

"<<"	{ return leftshift_operator; }

">>"	{ return rightshift_operator; }

"<="	{ return lessthan_assignment_operator; } "<"			{ return lessthan_operator;
}

">="	{ return greaterthan_assignment_operator; } ">"			{ return
greaterthan_operator; } "=="	{ return equality_operator; }
```

"!="  { return inequality_operator; }

"&&"  { return AND_operator; }

"||"  { return OR_operator; }

"^"  { return caret_operator; }

"*="  { return multiplication_assignment_operator; }

"/="  { return division_assignment_operator; }

"%="  { return modulo_assignment_operator; }

"+="  { return addition_assignment_operator; }

"-="  { return subtraction_assignment_operator; }

"<<="  { return leftshift_assignment_operator; }

">>="  { return rightshift_assignment_operator; }

"&="  { return AND_assignment_operator; }

"^="  { return XOR_assignment_operator; }

"|="  { return OR_assignment_operator; } "&"  { return amp_operator; }

"!"  { return exclamation_operator; }

"~"  { return tilde_operator; }

"-"  { return subtract_operator; }

"+"  { return add_operator; }

"*"  { return multiplication_operator; }

"/"  { return division_operator; }

"%"  { return modulo_operator; }

"|"  { return pipe_operator; }

\=  { return assignment_operator;}

\"[^\n]*\"/[;|,|\)]  {strcpy(curval,yytext); insertCT(yytext,"String Constant"); return string_constant;}

\'[A-Z|a-z]\'/[;|,|\)|:]  {strcpy(curval,yytext); insertCT(yytext,"Character Constant"); return character_constant;}

[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[ {strcpy(curid,yytext); insertST(yytext, "Array Identifier"); return array_identifier;} [1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|*|\/|\%|~|\]|\}|:|\n|\t|\^] {strcpy(curval,yytext); insertCT(yytext, "Number Constant"); yylval = atoi(yytext); return integer_constant;}

```
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^] {strcpy(curval,yytext);
insertCT(yytext, "Floating Constant"); return float_constant;}

[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext); insertST(curid,"Identifier"); return identifier;}

(.?) {

if(yytext[0]=='#')

{

printf("Error in Pre-Processor directive at line no. %d\n",yylineno);

}

else if(yytext[0]=='/')

{

printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);

}

else if(yytext[0]=='"')

{

}

else

{

}

printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);

printf("ERROR at line no. %d\n",yylineno);

printf("%s\n", yytext); return 0;

}

%%
```

# PARSER CODE

```
%{
	void yyerror(char* s);
	int yylex();
	#include "stdio.h"
	#include "stdlib.h"
	#include "ctype.h"
	#include "string.h"
	void ins();
	void insV();
	int flag=0;
	#define ANSI_COLOR_RED"\x1b[31m"
	#define ANSI_COLOR_GREEN	"\x1b[32m"
	#define ANSI_COLOR_CYAN	"\x1b[36m"
	#define ANSI_COLOR_RESET "\x1b[0m"
	extern char curid[20];
	extern char curtype[20];
	extern char curval[20];
	extern int currnest;
	void deletedata (int );
	int checkscope(char*);
	int check_id_is_func(char *);
	void insertST(char*, char*);
	void insertSTnest(char*, int);
	void insertSTparamscount(char*, int);
	int getSTparamscount(char*);
	int check_duplicate(char*);
	int check_declaration(char*, char *);
	int check_params(char*);
	int duplicate(char *s);
	int checkarray(char*);
	char currfunctype[100];
	char currfunc[100];
	char currfunccall[100];
	void insertSTF(char*);
	char gettype(char*,int);
	char getfirst(char*);
	extern int params_count;
	int call_params_count;
%}
%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
```

```
%token ENDIF
%expect 1
%token identifier array_identifier func_identifier
%token integer_constant string_constant float_constant character_constant
%nonassoc ELSE
%right leftshift_assignment_operator rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator modulo_assignment_operator
%right multiplication_assignment_operator division_assignment_operator
%right addition_assignment_operator subtraction_assignment_operator
%right assignment_operator
%left OR_operator
%left AND_operator
%left pipe_operator
%left caret_operator
%left amp_operator
%left equality_operator inequality_operator
%left lessthan_assignment_operator lessthan_operator greaterthan_assignment_operator
greaterthan_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator modulo_operator
%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator
%start program
%%
program
        : declaration_list;
declaration_list
        : declaration D
D
        : declaration_list
        | ;
declaration
        : variable_declaration
        | function_declaration
variable_declaration
        : type_specifier variable_declaration_list ';'
variable_declaration_list
        : variable_declaration_list ',' variable_declaration_identifier |
variable_declaration_identifier;
variable_declaration_identifier
        : identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins(); } vdi
        | array_identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,currnest); ins(); } vdi;
vdi : identifier_array_type | assignment_operator simple_expression ;
```

```
identifier_array_type
        : '[' initilization_params
        | ;
initilization_params
        : integer_constant ']' initilization {if($$ < 1) {printf("Wrong array size\n"); exit(0);} }
        | ']' string_initilization;
initilization
        : string_initilization
        | array_initialization
        | ;
type_specifier
        : INT | CHAR | FLOAT | DOUBLE
        | LONG long_grammar
        | SHORT short_grammar
        | UNSIGNED unsigned_grammar
        | SIGNED signed_grammar
        | VOID ;
unsigned_grammar
        : INT | LONG long_grammar | SHORT short_grammar | ;
signed_grammar
        : INT | LONG long_grammar | SHORT short_grammar | ;
long_grammar
        : INT | ;
short_grammar
        : INT | ;
function_declaration
        : function_declaration_type function_declaration_param_statement;
function_declaration_type
        : type_specifier identifier '(' { strcpy(currfunctype, curtype); strcpy(currfunc, curid);
check_duplicate(curid); insertSTF(curid); ins(); };
function_declaration_param_statement
        : params ')' statement;
params
        : parameters_list | ;
parameters_list
        : type_specifier { check_params(curtype); } parameters_identifier_list {
insertSTparamscount(currfunc, params_count); };
parameters_identifier_list
        : param_identifier parameters_identifier_list_breakup;
parameters_identifier_list_breakup
        : ',' parameters_list
        | ;
param_identifier
        : identifier { ins();insertSTnest(curid,1); params_count++; } param_identifier_breakup;
param_identifier_breakup
        : '[' ']'
        | ;
statement
```

```
        : expression_statment | compound_statement
        | conditional_statements | iterative_statements
        | return_statement | break_statement
        | variable_declaration;
compound_statement
        : {currnest++;} '{' statment_list '}' {deletedata(currnest);currnest--;} ;
statment_list
        : statement statment_list
        | ;
expression_statment
        : expression ';'
        | ';' ;
conditional_statements
        : IF '(' simple_expression ')' {if($3!=1){printf("Condition checking is not of type
int\n");exit(0);}} statement conditional_statements_breakup;
conditional_statements_breakup
        : ELSE statement
        | ;
iterative_statements
        : WHILE '(' simple_expression ')' {if($3!=1){printf("Condition checking is not of type
int\n");exit(0);}} statement
        | FOR '(' expression ';' simple_expression ';' {if($5!=1){printf("Condition checking is
not of type int\n");exit(0);}} expression ')'
        | DO statement WHILE '(' simple_expression ')'{if($5!=1){printf("Condition checking is
not of type int\n");exit(0);}} ';';
return_statement
        : RETURN ';' {if(strcmp(currfunctype,"void")) {printf("Returning void of a non-void
function\n"); exit(0);}}
| RETURN expression ';' {      if(!strcmp(currfunctype, "void"))
{
void");
yyerror("Function is
}
currfunctype[0]=='c') && $2!=1)
doesn't match return type of function\n"); exit(0);
if((currfunctype[0]=='i' ||
{
printf("Expression
}};
break_statement
        : BREAK ';' ;
string_initilization
        : assignment_operator string_constant {insV();} ;
array_initialization
        : assignment_operator '{' array_int_declarations '}';
array_int_declarations
        : integer_constant array_int_declarations_breakup;
array_int_declarations_breakup
```

```
        : ',' array_int_declarations
        | ;
expression
: mutable assignment_operator expression     {
        if($1==1 && $3==1)
        {
        $$=1;
        }
        else
{$$=-1; printf("Type mismatch\n"); exit(0);}
        }
        | mutable addition_assignment_operator expression  {
        if($1==1 && $3==1)
        $$=1;
        else
{$$=-1; printf("Type mismatch\n"); exit(0);}
        }
        | mutable subtraction_assignment_operator expression {
        if($1==1 && $3==1)
        $$=1;
        else
{$$=-1; printf("Type mismatch\n"); exit(0);}
        }
        | mutable multiplication_assignment_operator expression {
        if($1==1 && $3==1)
        $$=1;
        else
{$$=-1; printf("Type mismatch\n"); exit(0);}
        }
| mutable division_assignment_operator expression   {
        if($1==1 && $3==1)
        $$=1;
        else
{$$=-1; printf("Type mismatch\n"); exit(0);}
        }
| mutable modulo_assignment_operator expression    {
        if($1==1 && $3==1)
        $$=1;
        else
        {$$=-1; printf("Type mismatch\n"); exit(0);}
        }
| mutable increment_operator  {if($1 ==
1) $$=1; else $$=-1;}
        | mutable decrement_operator
        {if($1 == 1) $$=1; else $$=-1;}
        | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;
simple_expression
        : simple_expression OR_operator and_expression {if($1 == 1 && $3==1) $$=1; else
```

```
$$=-1;}
        | and_expression {if($1 == 1) $$=1; else $$=-1;};
and_expression
        : and_expression AND_operator unary_relation_expression {if($1 == 1 && $3==1)
$$=1; else $$=-1;}
        |unary_relation_expression {if($1 == 1) $$=1; else $$=-1;} ;
unary_relation_expression
        : exclamation_operator unary_relation_expression {if($2==1) $$=1; else $$=-1;}
        | regular_expression {if($1 == 1) $$=1; else $$=-1;} ;
regular_expression
        : regular_expression relational_operators sum_expression {if($1 == 1 && $3==1)
$$=1; else $$=-1;}
        | sum_expression {if($1 == 1) $$=1; else $$=-1;} ;
relational_operators
        : greaterthan_assignment_operator | lessthan_assignment_operator |
greaterthan_operator
        | lessthan_operator | equality_operator | inequality_operator ;
sum_expression
        : sum_expression sum_operators term  {if($1 == 1 && $3==1) $$=1; else $$=-1;}
        | term {if($1 == 1) $$=1; else $$=-1;};
sum_operators
        : add_operator
        | subtract_operator ;
term
        : term MULOP factor {if($1 == 1 && $3==1) $$=1; else $$=-1;}
        | factor {if($1 == 1) $$=1; else $$=-1;} ;
MULOP
        : multiplication_operator | division_operator | modulo_operator ;
factor
        : immutable {if($1 == 1) $$=1; else $$=-1;}
        | mutable {if($1 == 1) $$=1; else $$=-1;} ;
mutable
        : identifier {
        if(check_id_is_func(curid))
        {printf("Function name used as Identifier\n"); exit(8);}
        if(!checkscope(curid))
        {printf("%s\n",curid);printf("Undeclared\n");exit(0);}
        if(!checkarray(curid))
        {printf("%s\n",curid);printf("Array ID has no subscript\n");exit(0);}
        if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')
        $$ = 1;
        else
        $$ = -1;
        }
        | array_identifier
{if(!checkscope(curid)){printf("%s\n",curid);printf("Undeclared\n");exit(0);}} '[' expression ']'
        {if(gettype(curid,0)=='i' || gettype(curid,1)== 'c')
        $$ = 1;
```

```
        else
        $$ = -1;
        };
immutable
        : '(' expression ')' {if($2==1) $$=1; else $$=-1;}
        | call
        | constant {if($1==1) $$=1; else $$=-1;};
call
        : identifier '('{
        if(!check_declaration(curid, "Function"))
        { printf("Function not declared"); exit(0);}
        insertSTF(curid);
        strcpy(currfunccall,curid);
        } arguments ')'
        { if(strcmp(currfunccall,"printf"))
        {
        if(getSTparamscount(currfunccall)!=call_params_count)
        {
yyerror("Number of arguments in
function call doesn't match number of parameters");
//printf("Number of arguments in
function call %s doesn't match number of parameters\n", currfunccall);
exit(8);
        }
        }
        };
arguments
        : arguments_list | ;
arguments_list
        : expression { call_params_count++; } A ;
A
        : ',' expression { call_params_count++; } A
        | ;
constant
        : integer_constant      { insV(); $$=1; }
        | string_constant       { insV(); $$=-1;}
        | float_constant        { insV(); }
        | character_constant{ insV();$$=1; };
%%
extern FILE *yyin; extern int yylineno; extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *); void incertCT(char *, char *); void printST();
void printCT();
int main(int argc , char **argv)
{
        yyin = fopen(argv[1], "r");
        yyparse();
        if(flag == 0)
```

```
        {
        printf(ANSI_COLOR_GREEN "Status: Parsing Complete - Valid"
ANSI_COLOR_RESET "\n");
        printf("%30s" ANSI_COLOR_CYAN "SYMBOL TABLE" ANSI_COLOR_RESET
"\n", " ");
        printf("%30s %s\n", " ", "       ");
        printST();
        printf("\n\n%30s" ANSI_COLOR_CYAN "CONSTANT TABLE"
ANSI_COLOR_RESET "\n", " ");
        printf("%30s %s\n", " ", "       ");
        printCT();
        }
}
void yyerror(char *s)
{
        printf(ANSI_COLOR_RED "%d %s %s\n", yylineno, s, yytext);
        flag=1;
        printf(ANSI_COLOR_RED "Status: Parsing Failed - Invalid\n"
ANSI_COLOR_RESET);
        exit(7);
}
void ins()
{
        insertSTtype(curid,curtype);
}
void insV()
{
        insertSTvalue(curid,curval);
}
int yywrap()
{
        return 1;
}
```

# TESTING

# SYSTEM TESTING

Testing for Semantic Analyzer of Mini C Compiler Using Flex and Bison involves verifying the correctness and effectiveness of the semantic analysis process, as well as ensuring that the generated Abstract Syntax Tree (AST) accurately represents the semantics of the Mini C programs. Here are some testing approaches and techniques that can be employed:

## 1. UNIT TESTING

- Perform unit testing on individual components and functions of the Semantic Analyzer, such as type checking algorithms, symbol table management, and error handling mechanisms.
- Design test cases to cover different scenarios, including valid and invalid code samples, edge cases, and boundary conditions.
- Use testing frameworks like JUnit or PyTest to automate the execution and validation of unit tests.

## 2. INTEGRATION TESTING

- Conduct integration testing to verify the interaction and interoperability of the Semantic Analyzer with other components of the Mini C Compiler, such as the lexer (Flex) and parser (Bison).
- Test the communication between components, the flow of data and control, and the accuracy of the semantic analysis results.
- Design integration test cases that cover various code constructs, language features, and error scenarios.

## 3. FUNCTIONAL TESTING

- Perform functional testing to validate the compliance of the Semantic Analyzer with the

Mini C language specification.

- Create a set of test cases that cover different language constructs, statements, expressions, data types, and control flow.
- Verify that the Semantic Analyzer enforces language rules correctly and performs accurate type checking.

**4. ERROR HANDLING TESTING**

- Focus on testing the error detection and reporting capabilities of the Semantic Analyzer.
- Design test cases that deliberately introduce semantic errors and ensure that the analyzer identifies and reports them correctly.
- Validate the accuracy and clarity of the error messages generated by the analyzer.

**5. PERFORMANCE TESTING**

- Evaluate the performance of the Semantic Analyzer to ensure it can handle Mini C programs of varying sizes efficiently.
- Test the analyzer with large codebases and measure the analysis time and memory usage.
- Identify any bottlenecks or performance issues and optimize the analyzer accordingly.

**6. REGRESSION TESTING**

- Establish a regression test suite to ensure that modifications or enhancements to the Semantic Analyzer do not introduce new bugs or regressions.
- Re-run previously passed test cases to validate that the existing functionality remains intact after any changes.

**7. TEST AUTOMATION**

- Develop automated test scripts and frameworks to streamline the testing process and improve efficiency.
- Use tools like Selenium, JUnit, or PyTest to automate the execution and validation of test cases.
- Incorporate continuous integration (CI) and continuous testing practices to ensure regular and automated testing as part of the development pipeline.

## 8. CODE COVERAGE ANALYSIS

- Utilize code coverage tools to assess the completeness of testing.
- Measure the percentage of code exercised by the test suite to ensure comprehensive coverage of the Semantic Analyzer codebase.
- Aim for high code coverage to minimize the risk of undiscovered bugs or untested code paths.

## 9. USER ACCEPTANCE TESTING (UAT)

- Collaborate with end-users, developers, and stakeholders to perform user acceptance testing.
- Provide sample Mini C programs to users and gather their feedback on the behavior, correctness, and usability of the Semantic Analyzer.
- Incorporate user feedback to improve the functionality and user experience of the analyzer.

## 10. USABILITY TESTING

- Evaluate the usability of the Semantic Analyzer's user interface.
- Conduct user sessions or surveys to gather feedback on the intuitiveness, clarity, and effectiveness of the user interface.
- Identify areas for improvement and make necessary adjustments to enhance the user experience.

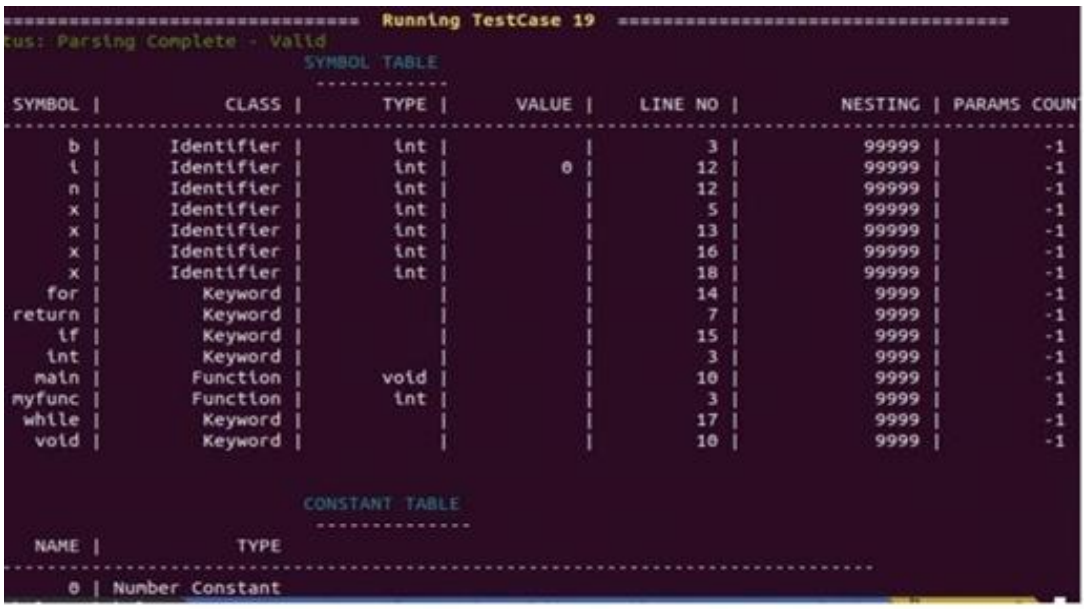
# CHAPTER 5

# RESULTS AND DISCUSSIONS

## 5.1 RESULT

The result of the semantic analyzer is a well-formed and semantically correct abstract syntax tree (AST) for the source code. The AST is a tree-like data structure that represents the program's syntactic structure and captures its semantics. The AST is used as an intermediate representation that is further processed by other compiler phases such as optimization and code generation.

Additionally, the semantic analyzer may also produce error messages when it encounters semantic errors in the source code. These error messages typically provide information about the location and nature of the error, making it easier for the programmer to locate and fix the error. The semantic analyzer may also perform various checks such as type checking, scope checking, and name binding to ensure that the program is semantically correct.



```
============================== Running TestCase 19 ==================================
tus: Parsing Complete - Valid
                        SYMBOL TABLE
                        ------------
 SYMBOL |          CLASS |       TYPE |      VALUE |    LINE NO |        NESTING | PARAMS COUN
-------------------------------------------------------------------------------------
      b |      Identifier |        int |            |          3 |          99999 |         -1
      i |      Identifier |        int |          0 |         12 |          99999 |         -1
      n |      Identifier |        int |            |         12 |          99999 |         -1
      x |      Identifier |        int |            |          5 |          99999 |         -1
      x |      Identifier |        int |            |         13 |          99999 |         -1
      x |      Identifier |        int |            |         16 |          99999 |         -1
      x |      Identifier |        int |            |         18 |          99999 |         -1
    for |         Keyword |            |            |         14 |           9999 |         -1
 return |         Keyword |            |            |          7 |           9999 |         -1
     if |         Keyword |            |            |         15 |           9999 |         -1
    int |         Keyword |            |            |          3 |           9999 |         -1
   main |        Function |       void |            |         10 |           9999 |         -1
 myfunc |        Function |        int |            |          3 |           9999 |          1
  while |         Keyword |            |            |         17 |           9999 |         -1
   void |         Keyword |            |            |         10 |           9999 |         -1


                        CONSTANT TABLE
                        --------------
  NAME |           TYPE
-------------------------------------------------------------------------------
     0 | Number Constant
```

```
========================  Running TestCase 10  ================================
sing Complete - Valid
                    SYMBOL TABLE
                    ------------
|            CLASS |        TYPE |       VALUE |     LINE NO |          NESTING | PARAMS
-------------------------------------------------------------------------------------
|          Keyword |             |             |           9 |             9999 |
|       Identifier |         int |          23 |           8 |            99999 |
|       Identifier |         int |          15 |           8 |            99999 |
|       Identifier |         int |             |          15 |            99999 |
|          Keyword |             |             |           8 |             9999 |
|       Identifier |         int |             |          12 |            99999 |
|       Identifier |         int |             |           9 |            99999 |
|       Identifier |         int |             |          10 |            99999 |
|       Identifier |       short |             |          11 |            99999 |
|          Keyword |             |             |          10 |             9999 |
|          Keyword |             |             |           6 |             9999 |
|         Function |         int |             |           6 |             9999 |
|          Keyword |             |             |           0 |             9999 |
|         Function |             |         "%d" |          16 |             9999 |

                    CONSTANT TABLE
                    --------------
|              TYPE
-------------------------------------------------------------------------------
| String Constant
| Number Constant
| Number Constant
```

```
================================  Running TestCase 12  ================================
Status: Parsing Complete - Valid
                          SYMBOL TABLE
                          ------------
  SYMBOL |          CLASS |       TYPE |      VALUE |    LINE NO |        NESTING | PARAMS COUNT |
-----------------------------------------------------------------------------------------------
       a |     Identifier |        int |         10 |         10 |          99999 |           -1 |
     for |        Keyword |            |            |         11 |           9999 |           -1 |
      do |        Keyword |            |            |         20 |           9999 |           -1 |
     int |        Keyword |            |            |          8 |           9999 |           -1 |
    main |       Function |        int |            |          8 |           9999 |           -1 |
  printf |       Function |            |       "H1" |         13 |           9999 |           -1 |
   while |        Keyword |            |            |         16 |           9999 |           -1 |

                          CONSTANT TABLE
                          --------------
    NAME |           TYPE
-------------------------------------------------------------------------------
    "H1" | String Constant
      10 | Number Constant
       0 | Number Constant
```

## 5.2 DISCUSSIONS

The Semantic Analyzer for the Mini C Compiler using Flex and Bison is a critical component that ensures the accuracy and correctness of the compiled code. One of the main discussions surrounding the Semantic Analyzer is its role in performing semantic analysis, which involves examining the code for compliance with the Mini C language rules and checking for semantic correctness. This process includes verifying proper variable usage, type compatibility, and adherence to language-specific constraints.

Overall, discussions about the Semantic Analyzer for the Mini C Compiler using Flex and Bison focus on its role in ensuring semantic correctness, symbol table management, error detection and reporting, extensibility, and performance optimization. These discussions contribute to a deeper understanding of the Semantic Analyzer's significance and its impact on the overall compilation process.

# CHAPTER 6

# CONCLUSION AND FUTURE ENHANCEMENT

## 6.1 CONCLUSION

In conclusion, the semantic analyzer is a crucial phase in the compilation process that checks the semantics or meaning of the program. It performs various tasks, such as type checking, scope analysis, and intermediate code generation, to ensure that the program is semantically correct and can be executed without any errors.

The effective management of the symbol table facilitates accurate scope resolution and enables efficient semantic operations. Additionally, the extensibility and flexibility of the Semantic Analyzer allow for future language enhancements and customizations. Improvements in performance, through optimization techniques and parallel processing, contribute to a faster and more efficient compilation process.

The semantic analyzer is usually implemented as a separate module or phase in the compiler, and it requires a well-defined language specification and symbol table to operate correctly.

A successful semantic analysis ensures that the program is semantically valid and can proceed to the next phases of the compilation process, such as code generation and optimization. However, if the program fails to pass the semantic analysis, the compiler generates an error message indicating the problem, and the programmer is required to fix the issue before proceeding with the compilation process.

Overall, the Semantic Analyzer plays a significant role in ensuring the accuracy, reliability, and usability of the Mini C Compiler, making it an essential component in the development and execution of Mini C programs.

## 6.2 FUTURE ENHANCEMENT

In the future, there are several potential enhancements that can be made to the Semantic Analyzer for the Mini C Compiler using Flex and Bison. One area of improvement is the inclusion of additional language features, expanding the support to handle more advanced constructs such as structures, pointers, arrays, and advanced control flow. This would increase the capabilities of the analyzer and allow it to handle a wider range of Mini C code.

Another aspect that can be enhanced is the error reporting mechanism. By providing more detailed and informative error messages, developers can quickly identify and resolve semantic errors in their code. Additionally, incorporating suggestions for potential fixes or code refactorings can help users address errors more easily and improve the overall development experience.

Performance optimization is another area for future enhancement. Techniques such as caching, lazy evaluation, and efficient symbol table management can be implemented to reduce analysis time and memory consumption. Employing parallel processing or concurrency techniques could further speed up the semantic analysis phase and improve overall performance.

Furthermore, integrating the Semantic Analyzer with code refactoring tools can automate the process of improving code structure, organization, and readability. The analyzer can provide suggestions for code refactorings based on semantic analysis insights, assisting developers in maintaining and improving their code.

Overall, these future enhancements would expand the capabilities, improve the performance, and provide a more user-friendly experience for the Semantic Analyzer of the Mini C Compiler using Flex and Bison.

# REFERENCES


1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools. Pearson Education. (This classic textbook covers compiler design principles and techniques, including lexical analysis, parsing, and semantic analysis.)

2. Levine, J. R., Mason, T., & Brown, D. (2009). Flex & Bison: Text Processing Tools. O'Reilly Media. (This book provides a comprehensive guide to using Flex and Bison for building lexical analyzers and parsers, which are essential components of the compiler.)

3. The Flex and Bison Official Documentation: The official documentation for Flex and Bison provides detailed information, tutorials, and examples on using these tools for building compilers and language processors. You can refer to the documentation available on the GNU website.

4. Dragon Book: Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann. (This book covers advanced topics in compiler design and implementation, including semantic analysis. It provides in-depth knowledge of compiler internals and optimization techniques.)

5. Cooper, K., & Torczon, L. (2011). Engineering a Compiler. Elsevier. (This book offers a practical approach to compiler construction, including semantic analysis. It provides insights into the implementation aspects of compilers.)

6. Open source compiler projects: You can also refer to open-source compiler projects that use similar technologies, such as GCC (GNU Compiler Collection) and LLVM (Low-Level Virtual Machine). Exploring their source code can provide valuable insights into implementing a semantic analyzer.