

**PROJECT REPORT ON**  
**TINY CONSTRUCTIVE COMPILER**  
**(TCC)**

**By**

**Kushagra Gaur (1013310049)**

**Sonu Kumar Kushwaha (1113310910)**



**Department of Computer Science**  
**Noida Institute of Engineering and Technology**  
**19, Knowledge Park-II, Institutional Area, Phase-II,**  
**Greater Noida, Uttar Pradesh**

**May, 2014**

# **TINY CONSTRUCTIVE COMPILER (TCC)**

**By**

**Kushagra Gaur (1013310049)**

**Sonu Kumar Kushwaha (1113310910)**

**Submitted to the Department of Computer Science**

**in partial fulfilment of the requirements**

**for the degree of**

**Bachelor in Technology**

**in**

**Computer Science and Engineering**



**Noida Institute of Engineering and Technology**

**U.P. Technical University**

**May, 2014**

## ***DECLARATION***

*We hereby declare that this submission is our own work and that, to the best of our knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.*

*Kushagra Gaur*

*1013310049*

*Sonu Kumar Kushwaha*

*1113310910*

*Date:*

# CERTIFICATE

This is to certify that the Project Report entitled “**TINY CONSTRUCTIVE COMPILER (TCC)**” which is submitted by **Kushagra Gaur** and **Sonu Kumar Kushwaha** in partial fulfillment of the requirement for the award of degree B. Tech. in **Department of Computer Science** of U. P. Technical University, is a record of the candidates’ own work carried out by them under my supervision. The matter embodied in this thesis is original and has not been submitted for the award of any other degree.

**Date:**

**Supervisor:**

**Mr. Nishant Kumar Hind**

Assistant Professor,

Department of Computer Science,

NIET

## ***ACKNOWLEDGEMENT***

*It gives us a great sense of pleasure to present the report of the B. Tech Project undertaken during B. Tech. Final Year. We owe special debt of gratitude to **Mr. Nishant Kumar Hind**, Assistant Professor, Department of Computer Science & Engineering, Noida Institute of Engineering and Technology, Greater Noida for his constant support and guidance throughout the course of our work. His sincerity, thoroughness and perseverance have been a constant source of inspiration for us.*

*We also take the opportunity to acknowledge the contribution of **Dr. Prashant Singh**, Head, Department of Computer Science & Engineering, Noida Institute of Engineering and Technology, Greater Noida for his full support and assistance during the development of the project.*

*We also do not like to miss the opportunity to acknowledge the contribution of all faculty members of the department for their kind assistance and cooperation during the development of our project.*

*Last but not the least, we acknowledge our family and friends for their contribution in the completion of the project.*

*Kushagra Gaur*

*1013310049*

*Sonu Kumar Kushwaha*

*1113310910*

*Date:*

## ***ABSTRACT***

*The present day programming languages lack the ability of providing a simple and a user friendly programming approach to the users, especially the novice users who are just entering into the programming world. As a result the users find it hard to learn programming and this ultimately results in the shortage of quality programmers.*

*This project aims at overcoming this hurdle by introducing a new programming language called Tiny which provides an alternative to the concurrent languages, especially for educational purposes, and developing a compiler for it.*

*The project uses a top down approach for solving the problem by dividing the main problem into a set of smaller sub-problems, namely Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Final Code Generation, Error Handling and Symbol Table Management.*

*The developed compiler uses Intel x86 processors as the target processor and Linux & Microsoft Windows NT as the operating platforms. It produces the assembly code using AT&T (GNU) syntax. It is capable of producing the binary code and the executable provided that the GNU Assembler & Linker are installed. It also features a unique ability to provide the users the options to view the output of each of the involving phase which makes the whole compiling process more transparent and easy to grasp.*

*The project has great potential to be a milestone in the direction of introducing programming to the novice users and it is expected that it will prove to be of great importance for the budding programmers.*

# TABLE OF CONTENTS

Page No.

DECLARATION	ii
CERTIFICATE	iii
ACKNOWLEDGEMENT	iv
ABSTRACT	v
LIST OF TABLES	viii
LIST OF FIGURES	ix
<b>CHAPTER 1: INTRODUCTION</b>	
1.1 Structure of Compiler	1
1.2 Phases of Compiler	2
1.3 Problem Statement	3
1.4 Objectives	4
<b>CHAPTER 2: SOFTWARE DEVELOPMENT MODEL</b>	
2.1 Software Development Life Cycle (SDLC)	5
2.2 Choice of Model	7
<b>CHAPTER 3: FEASIBILITY STUDY</b>	
3.1 Introduction	8
3.2 Feasibility Considerations	8
3.3 Steps in Feasibility Study	9
3.4 Prepare and Report Final Project Directive to Management	10
<b>CHAPTER 4: PROJECT PLANNING</b>	
4.1 Project Planning	12
<b>CHAPTER 5: REQUIREMENT ANALYSIS</b>	
5.1 Software Requirement Specification (SRS)	14
<b>CHAPTER 6: DESIGN</b>	
6.1 Project Design Types	16
6.2 Software Design Specification (SDS)	16
<b>CHAPTER 7: CODING</b>	
7.1 Programming Language for Development	21
7.2 Coding Standards	21
7.3 List of Source Code Files	22
7.4 Code Review	22

<b>CHAPTER 8: TESTING</b>	
8.1 Introduction	23
8.2 Static vs. Dynamic Testing	23
8.3 Black Box Testing	23
8.4 White Box Testing	24
8.5 Unit Testing	24
8.6 Integration Testing	24
8.7 System Testing	25
8.8 Test Cases	25
<b>CHAPTER 9: TINY – A PROGRAMMING LANGUAGE</b>	
9.1 What is Tiny?	33
9.2 The Character Set	33
9.3 Keywords	34
9.4 Constants	34
9.5 Variables	35
9.6 Comments	36
9.7 Data Types	36
9.8 Instructions	37
9.9 Escape Sequences	43
9.10 Statements and Expressions	43
9.11 Type Conversion	43
9.12 Arrays	44
9.13 A Few Programs in Tiny	44
<b>CHAPTER 10: IMPLEMENTATION</b>	
10.1 Program (Error Free)	48
10.2 Program (Having Errors)	54
<b>CHAPTER 11: CONCLUSION</b>	56
<b>APPENDIX A</b>	57
<b>APPENDIX B</b>	58
<b>APPENDIX C</b>	59
<b>APPENDIX D</b>	61
<b>APPENDIX E</b>	67
<b>REFERENCES</b>	70



# LIST OF TABLES

<b>Table No.</b>	<b>Table Name</b>	<b>Page No.</b>
7.1	List of Source Code Files	22
9.1	Tiny Character Set	33
9.2	Tiny Constants	34
9.3	Data Types in Tiny	36
9.4	Size Limits in Tiny	37
9.5	Standard Tiny Operators	41
9.6	String Operators in Tiny	42
9.7	Escape Sequences	43

# LIST OF FIGURES

<b>Fig. No.</b>	<b>Fig. Name</b>	<b>Page No.</b>
1.1	Phases of Compiler	2
2.1	Software Development Life Cycle	5
2.2	Iterative Development Model	7
6.1	Data Flow Diagram	19
6.2	Flowchart	20
9.1	Keywords	34
9.2	if-fi keywords	38
9.3	if-fi-else keywords	39
9.4	loop-pool keywords	39
10.1	Sample Program (Error Free)	48
10.2	Displaying Help Menu	49
10.3	Displaying Tokens	49
10.4	Displaying Parse Tree	50
10.5	Check Program Semantics	50
10.6	Display Symbol Table & IR	51
10.7	Generating Assembly Code	52
10.8	Target Assembly Code for Sample Input	53
10.9	Generating Assembly Code with Executable	54
10.10	Running the Executable	54

10.11	Sample Program with Lexical & Syntax Errors	54
10.12	TCC Catches the Lexical & Syntax Errors	55
10.13	Program with Semantic Errors	55
10.14	TCC Catches the Semantic Errors	55
D.1	Phases of Compiler Design	61
D.2	Steps of Parsing	63
D.3	Top Down Parsing	64
D.4	Bottom Up Parsing	64

# CHAPTER 1

## INTRODUCTION

A compiler is a system software that translates a program written in one language (source language) into another language (target language). It usually takes input as a set of files which constitute the source code written in the source language and converts them into their equivalent target code. The source language is usually a high level language which is human readable such as C, C++, Java etc. and the target language is usually a low level language which is more closer to the machine such as the assembly language or the object (binary) code itself.

Another common approach to the resulting compilation effort is to target a virtual machine. That will do just-in-time compilation and byte-code interpretation and blur the traditional categorizations of compilers and interpreters. Such an approach is used by languages such as Java. This brings the advantage of code portability from one machine to another as the Java byte code (which is not a true object code) is independent of the underlying architecture. But the extra overhead of this byte-code interpretation means slower execution speed.

Compiler construction is normally considered as an advanced programming task, mainly due to the quantity of code needed and the difficulties of managing this amount of code.

Any compiler has some essential requirements, which are perhaps more stringent than for most programs:

1. Any valid program must be translated correctly, i.e. no incorrect translation is allowed.
2. Any invalid program must be rejected and not translated.

There will inevitably be some valid programs which can't be translated due to their size or complexity in relation to the hardware available, for example problems due to memory size. The compiler may also have some fixed-size tables which place limits on what can be compiled.

### 1.1 Structure of Compiler

A compiler generally consists of 3 ends, front end, middle end and back end. These are briefly described below:

1. **Front end:** It translates the source code into machine independent intermediate code. It also checks for various errors in the source program and reports them.
2. **Middle end:** It optimizes the intermediate code generated to produce efficient final code. It is optional to use.

3. **Back end:** It translates the intermediate code from the middle end into the target code.

## 1.2 Phases of Compiler

Designing a compiler involves various phases viz. Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Intermediate Code Optimization, Final Code Generation, Symbol Table Management and Error Handling. These phases are shown in Fig. 1.1. Detailed discussion about the phases has been given in Appendix D.

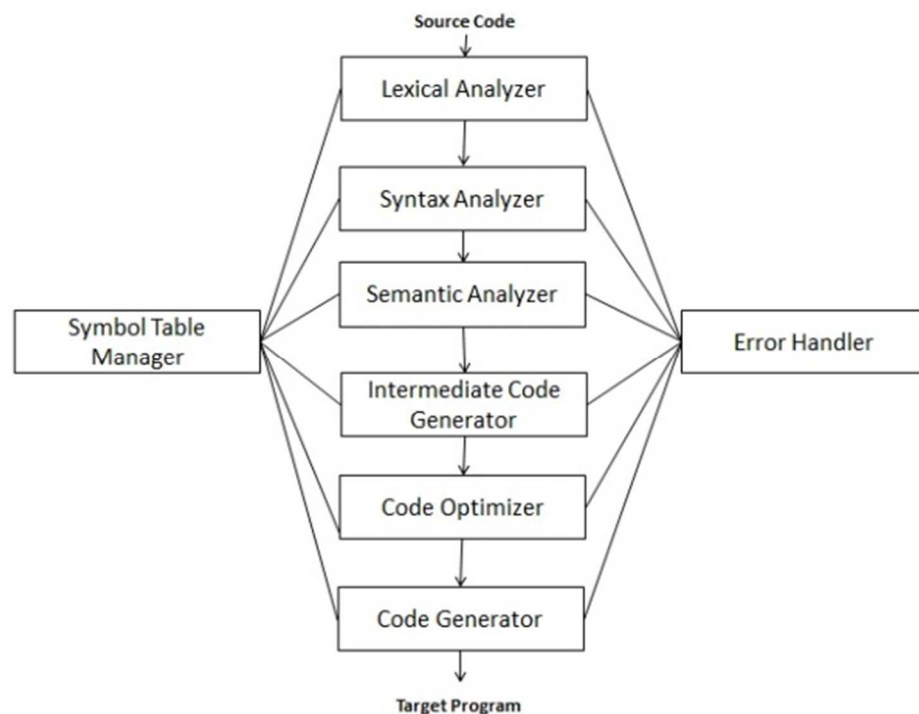


Fig. 1.1 – Phases of Compiler

### 1.2.1 Lexical Analysis

Lexical analysis or scanning is the process where the stream of characters making up the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning. These tokens are then passed to the next phase when required.

### 1.2.2 Syntax Analysis

Syntax analysis collects the tokens which are generated by Lexical analysis and converts these tokens into a parse tree according to the source language grammar. A parse tree is an abstract representation of the source program. Syntax analysis will also check for syntactical

errors which might be present in the program and reports them along with their locations and appropriate messages which help in the program diagnosis to remove the errors.

### **1.2.3 Semantic Analysis**

Semantic Analysis will check for syntactically correct programs for meaningful readings. It involves two major tasks:

1. Processing the declarations and building/updating the symbol table
2. Examining the rest of the program to ensure that the identifiers are used correctly adhering to the type compatibility conventions defined by the language.

Semantic analysis also decorates the abstract parse tree generated by the syntax analysis phase with more information on the data types.

### **1.2.4 Intermediate Code Generation**

Intermediate Code Generation phase will focus on generating the intermediate codes by walking through the parse tree which was decorated by the semantic analysis phase. Intermediate codes are necessary because they are independent of the target machine and are easy to translate into the target code.

Three Address Codes (TAC) are generally used as the intermediate codes.

### **1.2.5 Intermediate Code Optimization**

In this phase, the intermediate code is optimized by using several techniques, like elimination of common sub-expressions, elimination of unreachable code segments, and so on. This is an optional phase and is generally required for producing more efficient target code.

### **1.2.6 Final Code Generation**

In this phase, the intermediate code is translated into machine or assembly code. This phase is characterized by associating memory locations with the variables and choose the appropriate assembly instructions depending on the target processor. The output of this phase is the target code in the form of assembly language of the target hardware.

## **1.3 Problem Statement**

The present day programming languages lack the ability of providing a simple and a user friendly programming approach to the users, especially the novice users who are just entering into the programming world. As a result the users find it hard to learn programming and this ultimately results in the shortage of quality programmers.

This project aims at overcoming this hurdle by introducing a new programming language called Tiny which provides an alternative to the concurrent programming languages, especially for educational purposes, and developing a user-friendly compiler for it.

## 1.4 Objectives

The project must fulfil the below mentioned objectives:

1. Create a new programming language called Tiny with simple and user-friendly construct. The language must provide support for:
  - a. Basic operations like arithmetic operations, logical operations and relational operations.
  - b. Taking input from the keyboard (standard input device) and providing output to the console (standard output device).
  - c. Looping and conditional decision making
  - d. Basic data types like integer, character, real and string.
  - e. One Dimensional (1-D) arrays
2. Develop a user-friendly compiler for the above created language which:
  - a. Performs the lexical analysis.
  - b. Performs the syntax analysis.
  - c. Performs the semantic analysis
  - d. Provides an error handling mechanism.
  - e. Performs the symbol table management.
  - f. Generates the intermediate code.
  - g. Produces the final Intel x86 assembly code using AT&T (GNU) syntax
  - h. Generates the executable code after checking for GNU assembler and linker.
  - i. Provides an option to view the output of each of the above mentioned phases.
  - j. Is cross platform and must run on Linux and Microsoft Windows NT.

## CHAPTER 2

# SOFTWARE DEVELOPMENT MODEL

### 2.1 Software Development Life Cycle (SDLC)

The SDLC is a process used by a systems analyst to develop an information system, training, and user (stakeholder) ownership. Any SDLC should result in a high quality system that meets or exceeds customer expectations, reaches completion within time and cost estimates, works effectively and efficiently in the current and planned Information Technology infrastructure, and is inexpensive to maintain and cost-effective to enhance.

To manage this level of complexity, a number of SDLC models or methodologies have been created, such as Waterfall Model, Prototype Model, Spiral Model, Agile Software Development Model, Rapid Application Development Model and Iterative Model.



Fig. 2.1 – Software Development Life Cycle

The SDLC framework provides a sequence of activities for system designers and developers to follow. It consists of a set of steps or phases in which each phase of the SDLC uses the results of the previous one.

An SDLC adheres to important phases that are essential for developers, such as planning analysis, design and implementation and are explained in the sections below.



### 2.1.1 Preliminary Analysis

The objective of this phase is to conduct a preliminary analysis, propose alternative solutions, describe costs and benefits and submit a preliminary plan with recommendations.

1. **Conduct the preliminary analysis:** in this step, you need to find out the organization's objectives and the nature and scope of the problem under study. Even if a problem refers only to a small segment of the organization itself then you need find out what the objectives of the organization itself are. Then you need to see how the problem being studied fits in with them.
2. **Propose alternative solutions:** In digging into the organization's objectives and specific problems, you may have already covered some solutions. Alternate proposals may come from interviewing employees, clients, suppliers, and/or consultants. You can also study what competitors are doing. With this data, you will have three choices: leave the system as is, improve it, or develop a new system.

### 2.1.2 System Analysis

The goal of system analysis is to determine where the problem is in an attempt to fix the system. This step involves breaking down the system in different pieces to analyse the situation, analysing project goals, breaking down what needs to be created and attempting to engage users so that definite requirements can be defined.

### 2.1.3 Design

In systems design the design functions and operations are described in detail, including screen layouts, business rules, process diagrams and other documentation. The output of this stage will describe the new system as a collection of modules or subsystems.

The design stage takes as its initial input the requirements identified in the approved requirements document. For each requirement, a set of one or more design elements will be produced as a result of interviews, workshops, and/or prototype efforts.

Design elements describe the desired software features in detail, and generally include functional hierarchy diagrams, screen layout diagrams, tables of business rules, business process diagrams, pseudo code, and a complete entity-relationship diagram with a full data dictionary. These design elements are intended to describe the software in sufficient detail that skilled programmers may develop the software with minimal additional input design.

### 2.1.4 Testing

The code is tested at various levels in software testing. Unit, system and user acceptance testing are often performed. This is a grey area as many different opinions exist as to what the stages of testing are and how much, if any iteration occurs. Iteration is not generally part of the waterfall model, but usually some occur at this stage. In testing, all the project modules are first tested one by one and then their combined functionality is checked by integrating them with each other. In the end the project is tested as a whole to check the full functionality. Any errors which might have crept in the testing itself are removed by regression testing.

## 2.2 Choice of Model

The proposed project uses Iterative Model for development. In Iterative model, iterative process starts with a simple implementation of a small set of the software requirements and iteratively enhances the evolving versions until the complete system is implemented and ready to be deployed.

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which is then reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software at the end of each iteration of the model.

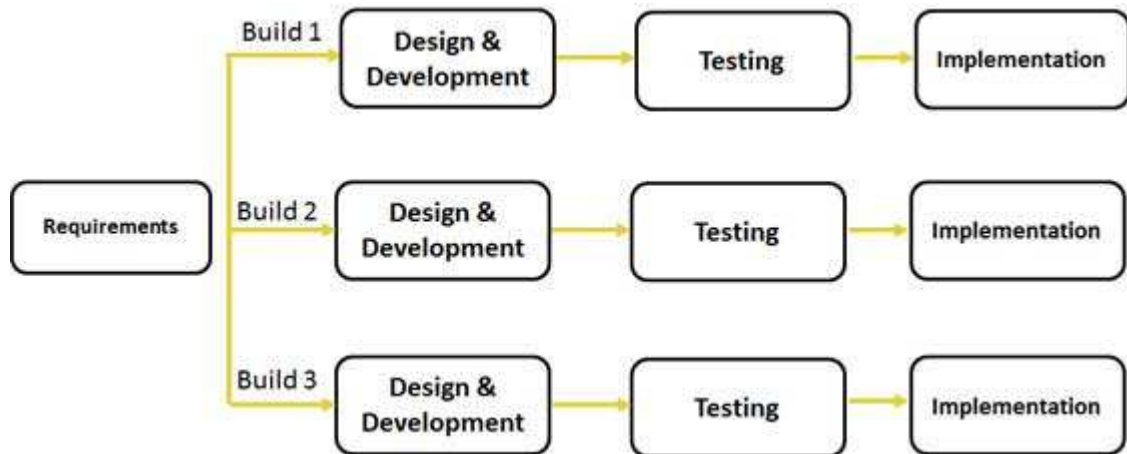


Fig. 2.2 – Iterative Development Model

## **CHAPTER 3**

### **FEASIBILITY STUDY**

#### **3.1 Introduction**

This step determines exactly what the candidate system is to do by defining its expected performance. This study is carried out to select the best system that meets performance requirements. This entails identification, description and evaluation of candidate systems and selection of the best system for the job.

Feasibility study serves as a decision document as it answers three key questions:

1. Is there a new and better way to do the job that will benefit the user?
2. What are the costs and savings of the alternatives?
3. What is recommended?

#### **3.2 Feasibility Considerations**

There are three key considerations that are involved in the feasibility analysis:

1. Economic
2. Technical
3. Behavioural

##### **3.2.1 Economic Feasibility**

Economic analysis is the most frequently used method for evaluating the effectiveness of a candidate system, more commonly known as cost benefit analysis. This procedure determines the benefits and savings that are expected from a candidate system and compare them with costs. If benefits outweigh costs then the decision is made to design and implement the system. Otherwise, further justifications in the proposed system will have to be made if it is to have a chance of being approved. The TCC project is economically feasible because the cost involved in purchasing the hardware and software are well within approachable limits.

##### **3.2.2 Technical Feasibility**

Technical feasibility centres on the existing computer system (hardware, software, etc.) and to what extent it can support the proposed solution. The TCC project uses ANSI C and the GNU Assembly language as the implementation languages. The standard and popular C compiler software GCC is readily available, easy to work with and widely used for developing various applications. GCC is a free software readily available for use in Linux and can be downloaded from the internet to work with Microsoft Windows NT platform.

The project uses any Intel x86 architecture microprocessors along with basic 512 MB DDR2 RAM with minimal 80 GB Hard Disk. This basic hardware is present in almost every existing system. The project can readily work with any hardware having higher specification than the stated ones.

### **3.2.3 Behavioural Feasibility**

This project is based on a subject “Principles of Compiler Design” that is taught to Computer Science graduates in Engineering, thus their reaction is anticipated to be cooperating. Since the development language is ANSI C, it will be much easier to understand. People are inherently resistant to change and computers have been known to facilitate change. An estimate should be made of how strong a reaction the user staff is likely to have toward the development of a computerized system. The proposed solution will help in saving time and fast processing and provide better performance. It is user friendly and timely generate the errors.

## **3.3 Steps in Feasibility Study**

Feasibility analysis involves the following steps:

1. Form a project team and appoint a team leader.
2. Prepare system flowcharts.
3. Enumerate potential candidate systems.
4. Describe and identify characteristics of candidate system.
5. Determine and evaluate performance and cost effectiveness.
6. Weight system performance and cost data.
7. Select the best candidate system.

### **3.3.1 Form a Project Team**

The project development team consists of the following two members:

1. Kushagra Gaur (Project Lead)
2. Sonu Kumar Kushwaha

### **3.3.2 Prepare System Flowcharts**

Information oriented charts and data flow diagrams are prepared. The chart brings up the importance of inputs, outputs and data flow among key points in the existing system. These have been shown in chapter 6 which deals with the project designing.

### **3.3.3 Enumerate Potential Candidate Systems**

This step identifies the candidate systems that are capable of producing the outputs included in the generalized flowcharts. This requires a transformation from logical to physical models. Another aspect of this step is consideration of the hardware that can handle the total system requirements. The project can work with any computer with 256 MB RAM.

### **3.3.4 Describe & Identify Characteristics of Candidate System**

The team begins a preliminary evaluation in an attempt to reduce the project to a manageable number. Technical knowledge and expertise in the hardware/software area are critical for determining what the system can do and can't do. The project has undergone thorough evaluation of the candidate system.

### **3.3.5 Determine and Evaluate Performance & Cost**

Each candidate system's performance is evaluated against the system performance requirements set prior to the feasibility study. The cost encompasses both designing and installing the system. It includes user training, updating the physical facilities and documenting. As the project is beginner level, it encompasses very little cost.

### **3.3.6 Weigh System Performance & Cost Data**

In some cases, the performance and cost data for each system show which the best choice is. This outcome terminates the feasibility study. Many times this situation is not clear so the next step is to weight the importance of each criterion by applying a rating figure. Then the System with the highest score is selected. Since the project has no other competitors, this step can be skipped.

### **3.3.7 Select the Best Candidate System**

The system with the highest total score is judged the best system.

## **3.4 Prepare & Report Final Project Directive to Management**

The culmination of feasibility study is a feasibility report directed to the management (in our case, the faculty); it evaluates and proposes changes on the area(s) in question. The report is a formal, brief, sufficiently detailed to start system designing.

The report contains the following sections:

Cover letter formally presents the report and briefly indicates the nature, general findings, and recommendations to be considered.

Table of contents specifies the location of various parts of the report.

Overview is a narrative explanation of the purpose and scope of the project, the reason for undertaking the feasibility study, and the departments involved by the system. It includes the names of the persons who conducted the study.

Detailed findings outline the methods used in the present system. The system's effectiveness and efficiency as well as operating costs are emphasized.

Economic justification details point-by-point comparisons and preliminary cost estimates for the development and operation of the system. A return on investment analysis is also included.

Recommendations and conclusions suggest the most beneficial and cost effective system. They are written only as recommendations and not as an order. Any conclusions may be included.

Appendices document all memos and data compiled during the investigation. They are placed at the end of the report for reference.

## CHAPTER 4

# PROJECT PLANNING

Planning is a critically important and required step in any successful project for system and software development. The objective of the project planning process is the development of a Baseline Project Plan (BPP) and the Statement of Work (SOW). The BPP becomes the foundation for the remainder of the development system. The SOW produced by the team clearly outlines the objectives and constraints of the project. During the planning of the software products, the scope of the software is described, various risk factors, cost and schedule are analysed and resources required are established.

### 4.1 Project Planning

Once a project is found to be feasible, software project managers undertake project planning. Project planning consists of the following activities:

1. Project scope
2. Project schedule
3. Project team organization
4. Technical description of proposed system
5. Project standards, procedures and proposed techniques tools
6. Quality assurance plan
7. Configuration management plan
8. Documentation plan
9. Data management plan
10. Resource management plan
11. Test plan
12. Training plan
13. Security plan
14. Risk management plan
15. Maintenance plan

#### 4.1.1 Project Scope

It involves estimating some basic attributes of the project like:

Cost: How much will it cost to develop the project?

Cost of the TCC project is nil.

Duration: How long will it take to complete the development?

Project took 4 months of active development.

Effort: How much effort would be required?

The project required extra ordinary efforts for both the designing and the coding.

#### **4.1.2 Project Schedule**

Scheduling the project task is an important project planning activity. It involves deciding which task would be taken up when. In order to schedule the project activities a project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.
7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

#### **4.1.3 Project Team Organisation**

It includes staff organization and staffing plans that means that each person is given his role in project development according to his skills.

The project team members along with their roles are:

1. Kushagra Gaur (Project Lead) – Designing, Coding, Testing, Documentation
2. Sonu Kumar Kushwaha – Designing, Testing, Documentation

#### **4.1.4 Risk Management Plan**

A risk is something that may happen and if it does, will have an adverse impact on the project.

The project did not require any risk management as no real risk was involved.



## CHAPTER 5

### REQUIREMENT ANALYSIS

The software requirement analysis activity is the next step of the software project planning in the software development. This phase is followed by software design. So, basically this phase is the intermediate task to be performed in between the computer system engineering and software design. The key and primary objectives of the requirements in terms of relationships, provide a basis for the design and provide a basis for validation for the software after it is built.

#### 5.1 Software Requirement Specification (SRS)

##### 5.1.1 Introduction

1. **Background:** The project is inspired with a view to understand the working & design of a programming language & its language translator.
2. **Purpose:** To create a basic new programming language called Tiny and develop a compiler for it.
3. **Scope:** The scope of the project covers all the science behind creating a programming language. It also covers the all the phases of compiler design viz. Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Final Code Generation, Symbol Table Management & Error Handling.
4. **Intended audience:** Faculty members & students.

##### 5.1.2 Overall Description

1. **Product perspective:** Independent programming language and its compiler software are developed in the project.
2. **User classes & characteristics:**
  - a) The user must be familiar with programming, both high level and assembly level.
  - b) The user must be aware of using Microsoft Windows NT Command-Prompt (Command line interpreter) software along with Linux Shell software.
3. **Operating Environment:** Microsoft Windows NT, Linux

##### 5.1.3 Functional Requirements

1. To generate and display the tokens in the source program.
2. To generate and display the parse tree of the source program
3. To generate and display the symbol table for the source program
4. To generate and display the Intermediate Representation (IR) of the source program
5. To generate the Intel x86 assembly code (in GNU syntax) of the source program

6. To create the executable code of the source program (requires GNU assembler & linker)
7. To check for lexical correctness of the source program
8. To check for syntax correctness of the source program
9. To check for the semantic correctness of the program
10. To provide full assistance to the user in the form of a help page

#### **5.1.4 Hardware Requirements**

1. Any Intel x86 CPU (preferably P4 or above)
2. Minimum 5 MB free RAM space
3. 100 KB hard disk space

#### **5.1.5 Software Requirements**

1. Microsoft Windows NT (DOS compatible) or Linux
2. GCC compiler

# CHAPTER 6

## DESIGN

The goal of design phase is to transform the requirements into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived. The objective of the design phase is to take the SRS document as input and produce the Software Design Specification (SDS) document. A good software design is seldom achieved by using a single step procedure but requires several iterations. The design activities can be classified into two important parts:

1. Preliminary Design (High Level Design)
2. Detailed Design

### 6.1 Project Design Types

The project design can be classified into:

1. Top Down Design
2. Bottom Up Design

The TCC project has been implemented using top down design. Top down design involves breaking down or decomposition of a problem into various sub problems. These sub problems are then solved individually and in the end are combined to solve the main problem. The details about the project modules has been discussed in the following sections.

### 6.2 Software Design Specification (SDS)

SDS is the output of the design phase. It is passed to the next phase of coding. The SDS for this project has been constructed using the Project Modules, Data Flow Diagram (DFD) and the Flowchart.

#### 6.2.1 Project Modules

The project has been decomposed into the following modules:

1. Main Module
2. Scanner Module
3. Parser Module
4. Semantic Analyser & Intermediate Code Generator Module
5. Final Code Generator Module
6. Error Handler Module
7. Symbol Table Manager Module

## 8. Grammar Module

### 6.2.1.1 Main Module

The main module provides the command line User Interface (UI). It does the following tasks:

1. Takes input from the user.
2. Calls other modules according to the input.
3. Displays the help menu.
4. Reports the errors, if present.

### 6.2.1.2 Scanner Module

The scanner module does the lexical analysis. Its tasks are listed below:

1. Reads the stream of characters from left to right and returns a stream of tokens.
2. Uses Deterministic Finite Automata (DFA) for lexical analysis.
3. Calls error handler if any invalid token is detected.

The DFA used has been given in Appendix A.

### 6.2.1.3 Parser Module

The parser module does the syntax analysis. Its tasks are given below:

1. Takes the input from the Scanner in the form of tokens.
2. Checks the syntax of the statements according to the grammar rules.
3. Uses LALR (Look Ahead LR) parsing technique.
4. Creates an abstract parse tree.
5. Calls error handler the program cannot be parsed.

### 6.2.1.4 Semantic Analyser & Intermediate Code Generator Module

This module is responsible for semantic analysis and intermediate code generation. Its tasks are given below:

1. Takes the input from the Parser in the form of a parse tree.
2. Evaluates the meaning (semantics) of the statements.
3. Uses Rule-Based semantic analysis technique to recursively evaluate the semantics.
4. Calls the Symbol Table Manager to create/update/search the symbol table.
5. Decorates the parse tree.
6. Creates the Intermediate Representation (IR).
7. Uses Three Address Codes (Quadruples) to generate IR.
8. Calls the error handler if semantic errors are detected.

### 6.2.1.5 Final Code Generator Module

This module is responsible for the target code generation phase. It does the following tasks:

1. Generates the final Intel x86 assembly level code which can be assembled by an assembler to generate the executable file.
2. Uses AT&T (GNU) syntax for the assembly code.

This module has been separately coded for the Linux and the Microsoft Windows NT OS because of the separate nature of the GCC assembler for both.

#### **6.2.1.6 Error Handler Module**

This module is responsible for error handling. It does the following tasks:

1. Handles and reports the errors (if found) in the source program.
2. Is able to detect and recover from the errors at multiple lines (using panic mode error recovery).
3. Handles the following errors:
  - a) If source file does not exist.
  - b) If wrong option is entered in the UI.
  - c) If invalid symbol is entered in the source file.
  - d) If program syntax is incorrect.
  - e) If program semantics are incorrect.

#### **6.2.1.7 Symbol Table Manager Module**

This module creates and manages the symbol table for the source program. A symbol table is a special table which stores various details about the identifiers used in the program such as its name, type, value, size, location, offset, etc.

This module does the following tasks:

1. Creates and manages the symbol table.
2. Uses Hash Table for the implementation.
3. Provides the following functionalities:
  - a) Insert the name of the identifier into symbol table.
  - b) Search the identifier in the symbol table.
  - c) Insert the value of the identifier into symbol table.
  - d) Insert the size of the identifier into symbol table.
  - e) Display the symbol table.

#### **6.2.1.8 Grammar Module**

This module stores the whole LALR Parse Table. This table is used by the Parser to verify the syntax of the program. It also stores the details about the grammar in the form of the terminals, non-terminals, start symbol and the grammar rules.

The LALR Parse Table has been given in Appendix B for reference.

#### **6.2.2 Data Flow Diagram (DFD – Level 1)**

A data-flow diagram (DFD) is a graphical representation of the "flow" of data through an information system. DFDs can also be used for visualization of data processing (structured design).

On a DFD, data items flow from an external data source or an internal data store to an internal data store or an external data sink, via an internal process. A DFD provides no information about the timing or ordering of processes, or about whether processes will operate in sequence or in parallel. It is therefore quite different from a flowchart, which shows the flow

of control through an algorithm, allowing a reader to determine what operations will be performed, in what order, and under what circumstances, but not what kinds of data will be input to and output from the system, nor where the data will come from and go to, nor where the data will be stored (all of which are shown on a DFD).

The DFD (Level 1) for this project has been shown in Fig. 6.1.

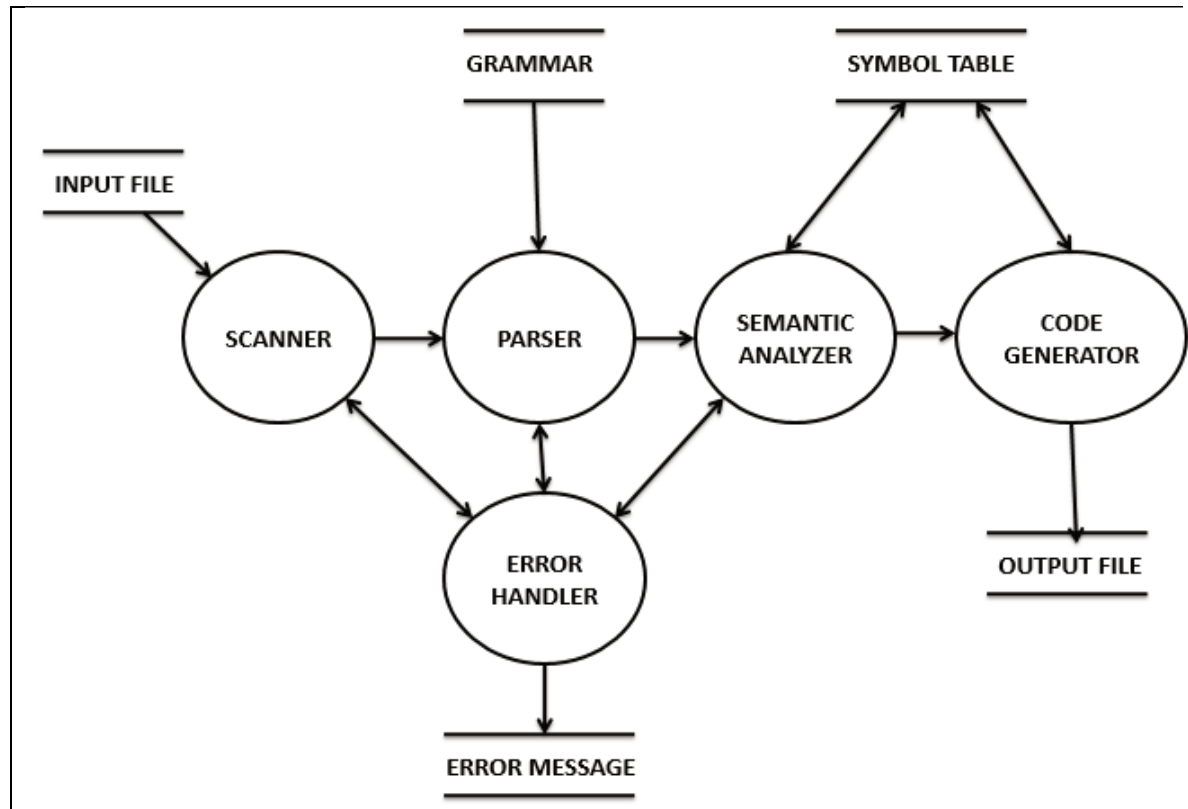


Fig. 6.1 – Data Flow Diagram

### 6.2.3 Flowchart

A flowchart is a common type of diagram that represents an algorithm or process showing the steps as boxes of various kinds, and their order by connecting these with arrows. Flowcharts are used in analysing, designing, documenting or managing a process or program in various fields.

The flowchart used in this project has been shown in Fig. 6.2.

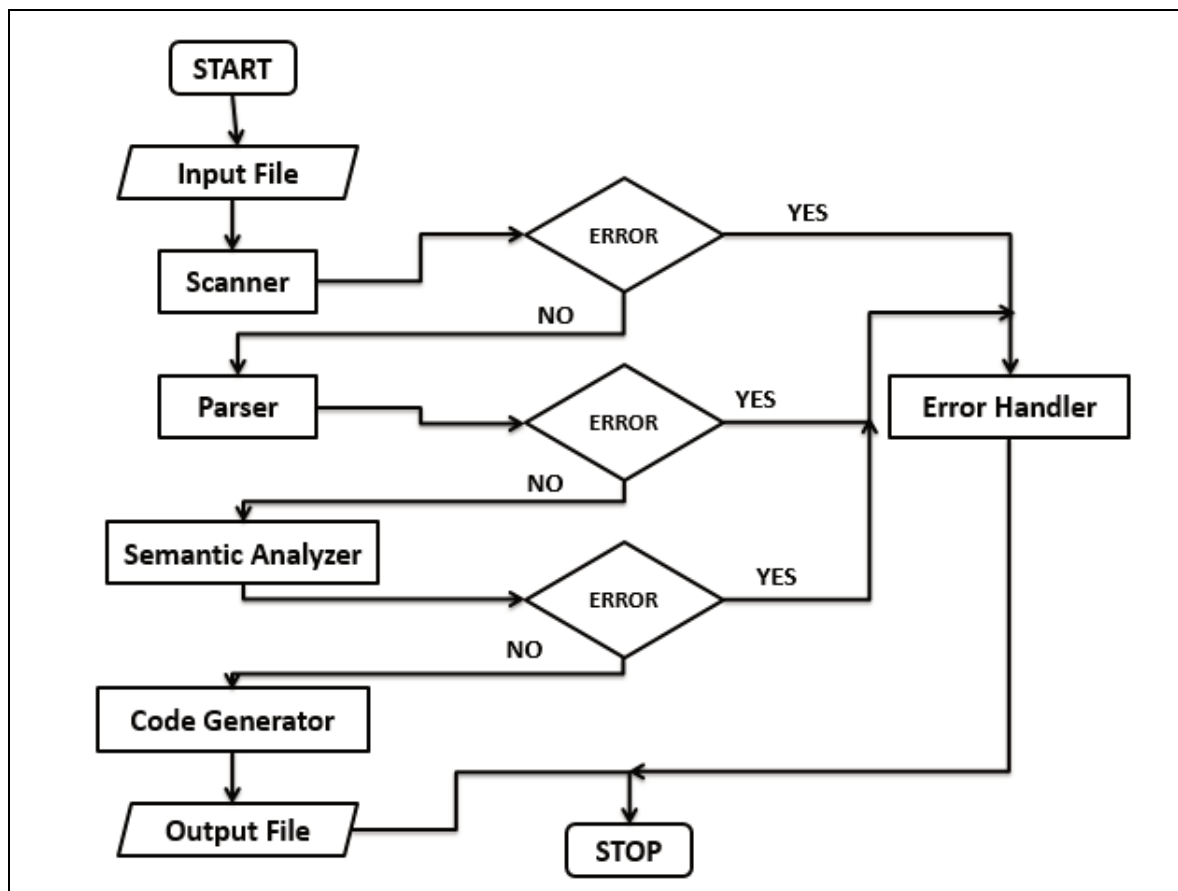


Fig. 6.2 – Flowchart

#### 6.2.4 Grammar

The grammar developed for the Tiny language has been given in Appendix C.

## **CHAPTER 7**

### **CODING**

The input to the coding is the design document. During the coding phase, different modules identified in the design document are coded according to the respective module specifications. Therefore the objective of the coding phase is to transform the design of a system, as given by its SDS, into a high level language code and to unit test this code.

The full project source code can be found in the CD-ROM supplied with this document.

#### **7.1 Programming Language for Development**

Given the nature of the project and the complexities involved in it, ANSI C and the assembly language were chosen for its development. The main focus during the development have been:

1. Efficiency
2. Low Space Complexity
3. Low Time Complexity
4. Code Portability
5. Cross Platform

The whole project has been developed in CodeBlocks IDE using the GCC compiler.

#### **7.2 Coding Standards**

The following coding standards have been maintained:

1. No use of clever or tricky programming.
2. No use of identifier for multiple purposes.
3. Variable must be given a meaningful name.
4. The code should be well documented.

##### **7.2.1 Coding Convention**

The following coding convention has been used throughout the project:

1. Functions - all letters are small
2. Local Variables - first two letters are underscore (\_\_)
3. Global variables - only first letter is underscore (\_)
4. Macros - all letters are capital



### 7.3 List of Source Code Files

The table 7.1 shows the modules along with the source file names which they have been developed in. The full project source code can be found in the supplied CD-ROM.

Table 7.1 – List of Source Code Files

File Name(s)	Implemented Module
tmain.c	Main Module
tlex.c, tlex.h	Scanner
tparse.c, tparse.h	Parser
tsemantic.c, tsemantic.h	Semantic Analyser & Intermediate Code Generator
x86linux.c, x86linux.h	Final Code Generator (Linux)
x86win.c, x86win.h	Final Code Generator (Windows NT)
errhandle.c, errhandle.h	Error Handler
syntab.c, syntab.h	Symbol Table Manager
grammar.c, grammar.h	Grammar Module

### 7.4 Code Review

Code review for the module is carried out after the module is successfully compiled and the syntax errors are eliminated. Normally two types of reviews are carried out:

1. Code Walkthrough
2. Code Inspection

#### 7.4.1 Code Walkthrough

It is an informal code analysis technique. In this technique, after the module has been coded, it is successfully compiled and all the syntax errors are eliminated.

The main objectives of code walkthrough are to discover the algorithmic and logical errors.

#### 7.4.2 Code Inspection

In contrast of code walkthrough, the aim of code inspection is to discover some common types of errors caused due to oversight and improper programming.

Following is the list of classical programming errors checked during code inspection:

1. Use of uninitialized variable.
2. Jumps into loops.
3. Non-terminating loops
4. Array indices out of bound
5. Improper storage allocation and reallocation
6. Mismatches between actual and formal parameter in procedure calls
7. Use of incorrect logical operators or incorrect precedence among operators.

# CHAPTER 8

## TESTING

### 8.1 Introduction

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software: one cannot completely test a program with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a trade-off between budget, time and quality. This is a non trivial pursuit. Testing cannot establish that a product function properly under all conditions. The scope of software testing often includes examination of code, execution of code in various environments and conditions as well as examining the aspects of code i.e. what it is doing, what it is supposed to do and what it needs to correct. In the current scenario of software development, a testing organization may be separate from the development team. Information derived from software testing may be used to correct the process by which software is developed.

### 8.2 Static vs. Dynamic Testing

Static testing is a form of software testing where the software isn't actually used. It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily checking of the code and/or manually reviewing the code or document to find errors. Dynamic testing takes place when the program itself is run. Dynamic testing may begin before the program is 100 percent complete in order to test particular sections of code and are applied to discrete functions or modules. Typical techniques for this are either using stubs/drivers or execution from a debugger environment.

### 8.3 Black Box Testing

Black-box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation. The testers are only aware of what the software is supposed to do, not how it does it. Black-box testing methods include: equivalence

partitioning, boundary value analysis, all-pairs testing, state transition tables, decision table testing, fuzztesting, model-basedtesting, usecase testing, exploratory testing and specification-based testing. Test cases are built around specifications and requirements, i.e., what the application is supposed to do. Test cases are generally derived from external descriptions of the software, including specifications, requirements and design parameters.

## **8.4 White Box Testing**

White box testing is when the tester has access to the internal data structures and algorithms including the code that implements these.

We designed the test cases for carrying out the white box according to the following strategies:

1. Statement coverage
2. Branch coverage
3. Condition coverage

Software is tested on different levels like unit testing, integration testing, system testing and acceptance testing. The product is tested on following levels.

## **8.5 Unit Testing**

Unit testing is a software development process that involves synchronized application of a broad spectrum of defect prevention and detection strategies in order to reduce software development risks, time, and costs. It is performed by the software developer or engineer during the construction phase of the software development lifecycle.

All the project modules have been tested independently for discovering bugs at unit level.

## **8.6 Integration Testing**

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing

All the modules are integrated in bottom-up manner and are tested after integration.

## 8.7 System Testing

As a rule, system testing takes, as its input, all of the "integrated" software components that have passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limited type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

The product is tested after integration of all the modules. Also it is tested against the hardware and software requirements.

## 8.8 Test cases

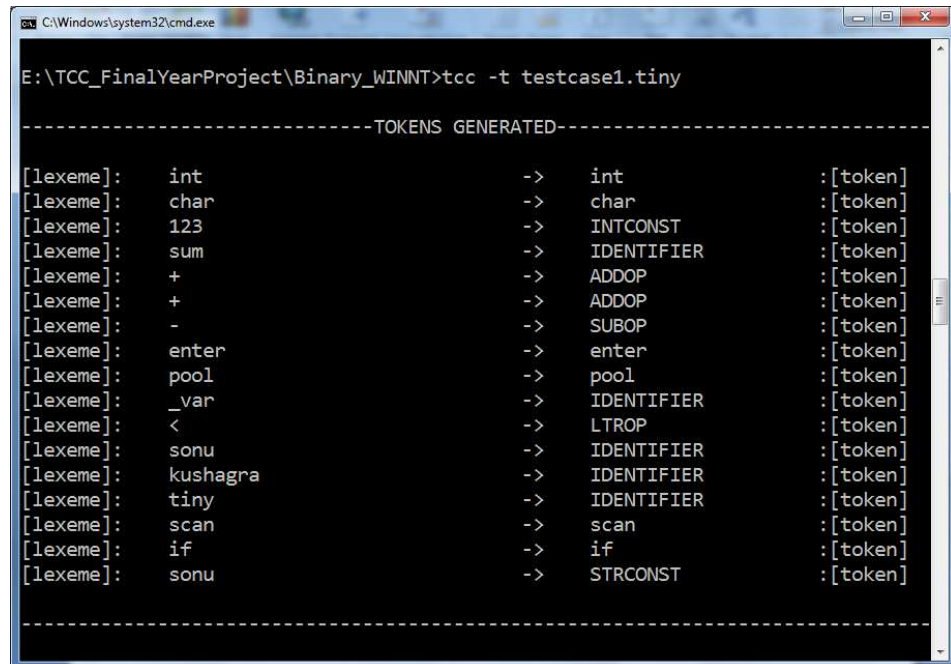
The project has been tested thoroughly using appropriate test cases. The test cases have been chosen in such a way that they cover the entire functionality of the project, both module wise (unit testing) and system wise (integration testing).

### 8.8.1 Test case 1 (Testing the Scanner module independently)

Testcase1.tiny

```
int char 123 sum + + - enter pool _var  
< sonu kushagra tiny scan if "sonu"
```

## Output



```

C:\Windows\system32\cmd.exe
E:\TCC_FinalYearProject\Binary_WINNT>tcc -t testcase1.tiny

-----TOKENS GENERATED-----

[lexeme]:  int          ->  int          :[token]
[lexeme]:  char         ->  char         :[token]
[lexeme]:  123          ->  INTCONST    :[token]
[lexeme]:  sum          ->  IDENTIFIER  :[token]
[lexeme]:  +           ->  ADDOP       :[token]
[lexeme]:  +           ->  ADDOP       :[token]
[lexeme]:  -           ->  SUBOP       :[token]
[lexeme]:  enter       ->  enter       :[token]
[lexeme]:  pool        ->  pool        :[token]
[lexeme]:  _var        ->  IDENTIFIER  :[token]
[lexeme]:  <           ->  LTROP       :[token]
[lexeme]:  sonu        ->  IDENTIFIER  :[token]
[lexeme]:  kushagra    ->  IDENTIFIER  :[token]
[lexeme]:  tiny        ->  IDENTIFIER  :[token]
[lexeme]:  scan        ->  scan        :[token]
[lexeme]:  if          ->  if          :[token]
[lexeme]:  sonu        ->  STRCONST    :[token]

```

## Remarks

All the lexemes are identified properly.

### 8.8.2 Test case 2 (Testing the integration of Scanner and Error Handler)

#### Testcase2.tiny

```

int char 123 sum + + - enter pool 6s _var
< sonu kushagra tiny scan if "sonu"
123var

```

## Output

```

C:\Windows\system32\cmd.exe
E:\TCC_FinalYearProject\Binary_WINNT>tcc -t testcase2.tiny

-----TOKENS GENERATED-----

[lexeme]:    int           ->    int           :[token]
[lexeme]:    char          ->    char          :[token]
[lexeme]:    123           ->    INTCONST      :[token]
[lexeme]:    sum           ->    IDENTIFIER   :[token]
[lexeme]:    +             ->    ADDOP        :[token]
[lexeme]:    +             ->    ADDOP        :[token]
[lexeme]:    -             ->    SUBOP        :[token]
[lexeme]:    enter        ->    enter       :[token]
[lexeme]:    pool         ->    pool        :[token]
testcase2.tiny: lexical error at line 5: illegal symbol '6s' is used
[lexeme]:    _var         ->    IDENTIFIER   :[token]
[lexeme]:    <            ->    LTROP        :[token]
[lexeme]:    sonu         ->    IDENTIFIER   :[token]
[lexeme]:    kushagra     ->    IDENTIFIER   :[token]
[lexeme]:    tiny         ->    IDENTIFIER   :[token]
[lexeme]:    scan         ->    scan        :[token]
[lexeme]:    if           ->    if          :[token]
[lexeme]:    sonu         ->    STRCONST    :[token]
testcase2.tiny: lexical error at line 7: illegal symbol '123var' is used

-----

```

## Remarks

Errors are identified by the error handler. Integration of Scanner and Error Handler is bug free.

### 8.8.3 Test case 3 (Testing the Parser module independently)

Tescase3.tiny

```

enter

    int var[5],i|
    print "Enter the array\n"|
    loop i<5
        scan var[i]| i=i+1|
    pool
exit

```

## Output

```

C:\Windows\system32\cmd.exe
E:\TCC_FinalYearProject\Binary_WINNT>tcc -p testcase3.tiny

-----PARSE TREE-----

rule 1 :      *PROG$      -> enter SUBPROG$ exit
rule 3 :      SUBPROG$    -> INITSTMT$ SUBPROG$
rule 7 :      INITSTMT$   -> IDTYPE$ IDNAME$ |
rule 9 :      IDTYPE$     -> int
rule 14:      IDNAME$     -> ID$ TYPE$ , IDNAME$
rule 16:      ID$         -> var [ EXPRS$ ]
rule 29:      EXPRS$      -> CONST$
rule 21:      CONST$      -> 5
rule 13:      IDNAME$     -> ID$ TYPE$
rule 15:      ID$         -> i
rule 6 :      SUBPROG$    -> STMT$ SUBPROG$
rule 51:      STMT$       -> print OUTVAR$ |
rule 54:      OUTVAR$     -> Enter the array\n
rule 5 :      SUBPROG$    -> LOOPSTMT$ SUBPROG$
rule 50:      LOOPSTMT$   -> loop EXPRP$ SUBPROG$ pool
  
```

## Remarks

The parser module is checking the syntax and generating the parse tree.

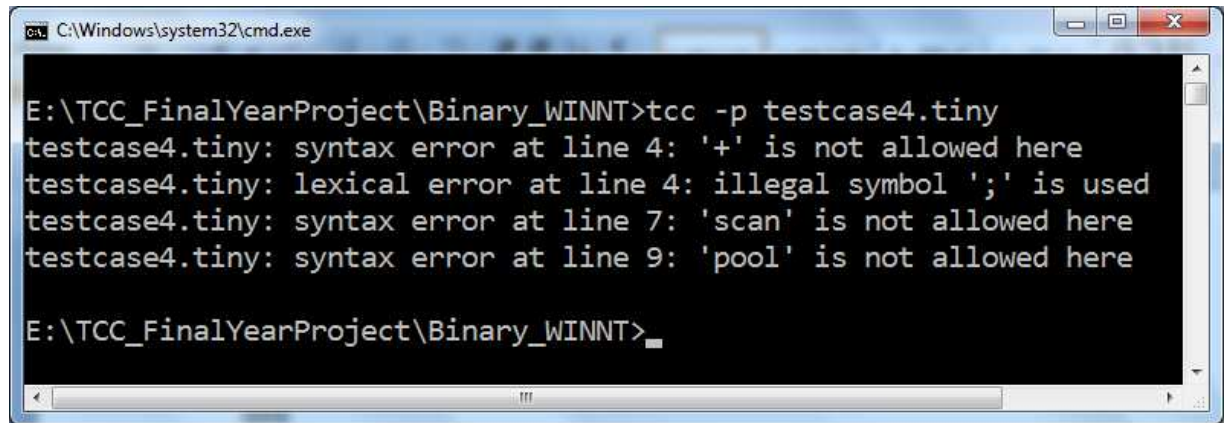
### **8.8.4 Test case 4 (Testing the integration of Parser with Error Handler)**

Testcase4.tiny

```

enter
    + int var[5],i| ;
    print "Enter the array\n"|
    loop i<5 =
        scan var[i]|
        i=i+1|
    pool
exit
  
```

## Output



```

C:\Windows\system32\cmd.exe

E:\TCC_FinalYearProject\Binary_WINNT>tcc -p testcase4.tiny
testcase4.tiny: syntax error at line 4: '+' is not allowed here
testcase4.tiny: lexical error at line 4: illegal symbol ';' is used
testcase4.tiny: syntax error at line 7: 'scan' is not allowed here
testcase4.tiny: syntax error at line 9: 'pool' is not allowed here

E:\TCC_FinalYearProject\Binary_WINNT>

```

## Remarks

Error Handler is detecting syntax errors in parsing.

### **8.8.5 Test Case 5 (Testing the Semantic Analysis module independently)**

Testcase5.tiny

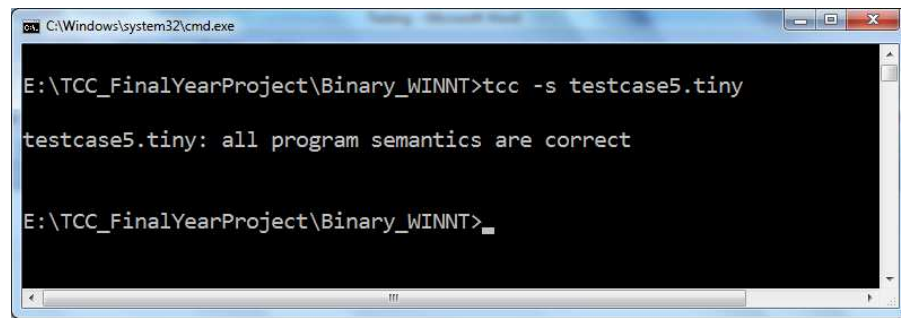
```

enter
    int sum,num,temp|
    print "Enter number\n"|
    scan num|
    if num<=0
        print "Number is invalid\n"|
    else
        temp=num|
        loop temp>0
            sum=sum+temp| temp=temp-1|
        pool
        print "Sum upto ",num," is ",sum|
    fi
exit

```



## Output



```
C:\Windows\system32\cmd.exe
E:\TCC_FinalYearProject\Binary_WINNT>tcc -s testcase5.tiny
testcase5.tiny: all program semantics are correct
E:\TCC_FinalYearProject\Binary_WINNT>
```

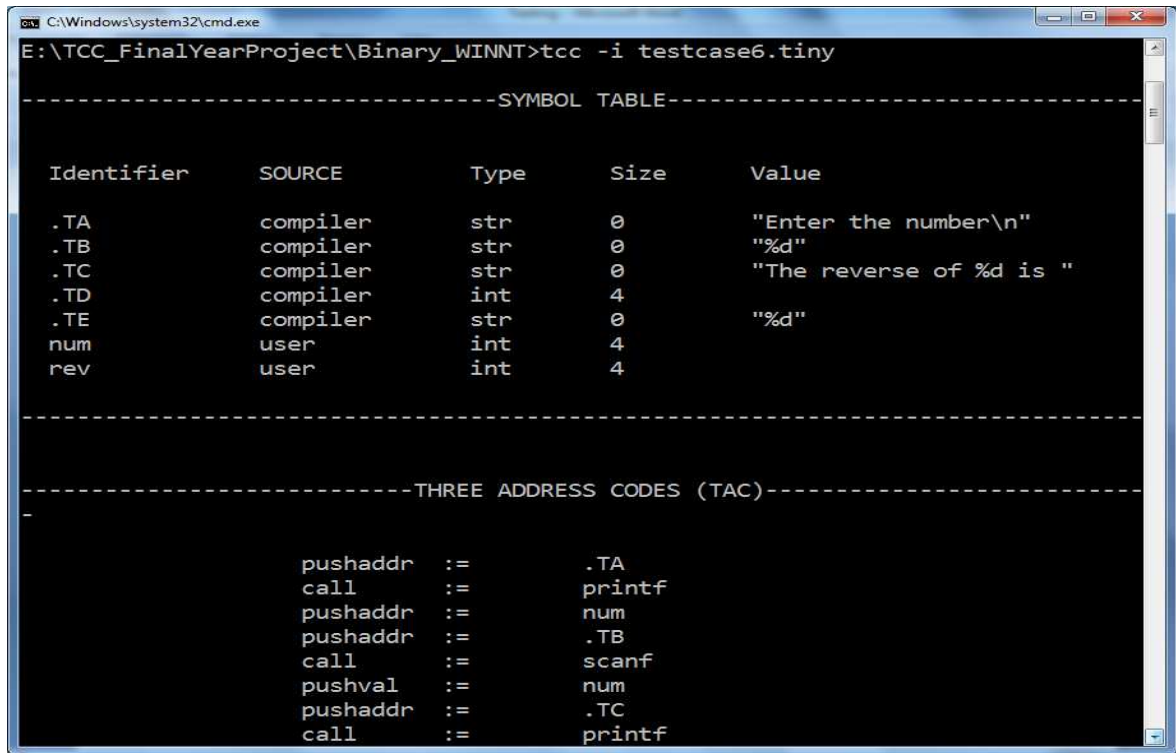
## Remarks

Program semantics have been checked, Semantic Analysis is working properly.

### **8.8.6 Test Case 6 (Testing the Semantic Analyzer & Intermediate code Generator with Symbol Table Manager)**

```
Testcase6.tiny
enter
    int num,rev|
    print "Enter the number\n"|
    scan num|
    print "The reverse of ",num," is "|
    loop num
        rev=(num%10)|
        print rev|
        num=num/10|
    pool
exit
```

## Output



```

C:\Windows\system32\cmd.exe
E:\TCC_FinalYearProject\Binary_WINNT>tcc -i testcase6.tiny

-----SYMBOL TABLE-----

Identifier      SOURCE      Type      Size      Value
-----
.TA              compiler   str        0      "Enter the number\n"
.TB              compiler   str        0      "%d"
.TC              compiler   str        0      "The reverse of %d is "
.TD              compiler   int        4
.TE              compiler   str        0      "%d"
num              user       int        4
rev              user       int        4

-----THREE ADDRESS CODES (TAC)-----
-
                                pushaddr  :=      .TA
                                call       :=      printf
                                pushaddr  :=      num
                                pushaddr  :=      .TB
                                call       :=      scanf
                                pushval   :=      num
                                pushaddr  :=      .TC
                                call       :=      printf

```

## Remarks

Semantics is checked properly, Symbol Table is generated properly and Intermediate code is generated.

### **8.8.7 Test Case 7 (Testing Assembly Code Generator with integration to other modules)**

Testcase7.tiny

```

enter

str:20 a,b,c|
print "Enter the string 1\n"|
scan a|
print "Enter the string 2\n"|
scan b|
c=a+b|
print "The concatenated string is ",c,"\\n"| exit

```

## Output

An assembly file named “testcase7.s” is generated.

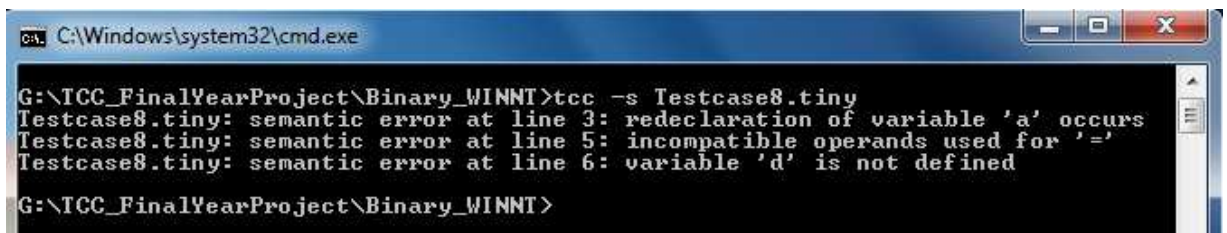
### Remarks

The Assembly Code Generator is working properly.

### **8.8.8 Test case 8 (Testing the integration of Semantic Analyzer & Intermediate Code generator with Error Handler)**

```
Testcase8.tiny  
  
enter  
  
    char a|  
    int a,b|  
    str c|  
    c=b+2|  
    print d|  
  
exit
```

### Output



```
C:\Windows\system32\cmd.exe  
  
G:\TCG_FinalYearProject\Binary_WINNT>tcc -s Testcase8.tiny  
Testcase8.tiny: semantic error at line 3: redeclaration of variable 'a' occurs  
Testcase8.tiny: semantic error at line 5: incompatible operands used for '='  
Testcase8.tiny: semantic error at line 6: variable 'd' is not defined  
G:\TCG_FinalYearProject\Binary_WINNT>
```

### Remarks

The Error Handler is detecting and reporting the errors generated in semantic analysis.

## CHAPTER 9

### TINY – A PROGRAMMING LANGUAGE

Before discussing about Tiny, it is imperative to know what programming languages really are. Communication with a computer involves speaking a language which it understands. As Computers are digital electronic devices, the only language they understand is the binary language. Binary language consists of only two symbols, 0 & 1. As it is cumbersome and error prone for humans to write instructions for the computer in binary language, we need a language which we are more comfortable with. Unfortunately, due to technical limitations we don't have the luxury of writing programs in English, so we need to create other high level languages to achieve this. Like all other programming languages, Tiny is also one such attempt to create a high level language for writing computer programs.

#### 9.1 What is Tiny?

When it comes to development, there are numerous programming languages available such as C, C++, Python, Java, C# etc. Unfortunately these languages lack the ability of being novice user friendly. While they pack some brilliant features, but when it comes to introducing someone to programming, they don't quite fit well. Result of this is that people start to fear programming rather than enjoying it and this ultimately results in the serious shortage of quality programmers. This limits both the quality and quantity of software.

Tiny is a high level programming language which is developed keeping in mind the need of a language which is more suitable for educational purposes, rather than development purposes. Because of its easy syntax and simplicity it is more suited for someone who is very new to programming and wants to learn the art of programming the easy way.

Tiny is a sequential, case-sensitive and free form language.

#### 9.2 The Character Set

Table 9.1 shows the valid characters allowed in Tiny.

Table 9.1 – Tiny Character Set

Alphabets	A-Z a-z
Digits	0,1,2,3,4,5,6,7,8,9
Special symbols	, # % ^ & * ( ) _ - + =   \ [ ] : " ' < > .

### 9.3 Keywords

Keywords are the reserved words who carry a special meaning. They cannot be used as variable names. Tiny has 13 keywords. Fig. 9.1 lists these keywords.

char
else
enter
exit
fi
if
int
loop
pool
print
real
scan
str

Fig. 9.1 - Keywords

### 9.4 Constants

A constant is an entity that doesn't change.

#### 9.4.1 Type of Tiny Constants

There are 2 categories of constants in Tiny:

1. Primary Constants
2. Secondary Constants

These can be further categorised as shown in table 9.2.

Table 9.2 – Tiny Constants

Primary Constants	Secondary Constants
Integer Constant	Arrays
Character Constant	
Real Constant	
String Constant	

#### 9.4.2 Rules for Constructing Integer Constants

1. It must have at least one digit.
2. It must not have a decimal point.
3. It can be either positive or negative.

4. Without any sign, it is assumed to be positive.
5. No commas or blanks are allowed.
6. No characters other than digits are allowed.

Examples: 192, +192, -192 etc.

#### **9.4.3 Rules for Constructing Real Constants**

1. It must have at least one digit.
2. It must a decimal point.
3. It can be either positive or negative.
4. Without any sign, it is assumed to be positive.
5. No commas or blanks are allowed.
6. No characters other than digits and point are allowed.

Examples: 2.4, +2.567, -2.675, .76, 2. etc.

#### **9.4.4 Rules for Constructing Character Constants**

1. A character constant is a single alphabet, or a single digit or a single special symbol enclosed within inverted commas.
2. The maximum length of a character constant can be 1 character.

Examples: 'a', '+', '9' etc.

#### **9.4.5 Rules for Constructing String Constants**

1. A sting constant is any combination of allowed characters enclosed within inverted double commas.
2. There is no limit on the number of characters allowed in string constants.

Examples: "Hi, This is Tiny", "2+2=4" etc.

### **9.5 Variables**

An entity that may vary during program execution is called a variable. Variable names are names given to memory locations. Tiny supports 4 types of variables:

1. Integer variable (can contain integer values)
2. Real variable (can contain real values)
3. Character variable (can contain character values)
4. String variable (can contain string values)

#### **9.5.1 Rules for Constructing Variables**

1. A variable is any combination of alphabets, digits or underscores.
2. The first character in the variable name must be an alphabet or underscore.
3. No commas or blanks are allowed in variable names.
4. No special symbol except underscore (\_) are allowed in variable names.

## 9.6 Comments

Comments are used by the programmer for making the code more understandable. These are lines of code which are simply ignored by the compiler. In Tiny, only single line comments are supported. The comments start with a hash (#) symbol. All the characters which are found ahead of the hash symbol till the next new line are simply ignored by the compiler.

Example:      `char a|      # declaring a character variable, this line is ignored by the compiler`

## 9.7 Data Types

There are four primitive data types in Tiny.

1. Integer data type (declared using `int` keyword)
2. Character data type (declared using `char` keyword)
3. Real data type (declared using `real` keyword)
4. String data type (declared using `str` keyword)

All the data types except the string data type are signed. The default memory location of the variables declared using these data types is RAM and the default value is 0. Table 9.3 shows these details along with the default size of each type.

Table 9.3 – Data Types in Tiny

Data Type	Keyword	Default Size (in Bytes)	Location	Default Value	Sign
Integer	<code>int</code>	4	RAM	0	Signed
Character	<code>char</code>	2	RAM	0	Signed
Real	<code>real</code>	4	RAM	0	Signed
String	<code>str</code>	4	RAM	NULL	Unsigned

### 9.7.1 Data Type Variable Size

The default sizes of the data types in Tiny can be changed using the Variable Size Operator (:). For example, the default size of integer is 4 bytes but it can be changed to 1 byte or 2 bytes as follows.

`int a|      # declares an integer of 4 bytes`

`int: 2 a|      # declares an integer of 2 bytes`

`int: 1 a|      # declares an integer of 1 byte`

Similarly the sizes of character variables can also be changed. The size limits for real types cannot be changed, they are fixed as 4 bytes. Any attempt to define the size limit for real types will result in an error. Table 9.4 shows the valid size limits in Tiny.

Table 9.4 – Size Limits in Tiny

Data Type	Supported Size Limit (in Bytes)
Integer	1, 2, 4
Character	1, 2
Real	Not supported

### 9.7.2 Characters Count Operator (:) in String Types

The colon symbol acts as Characters Count Operator in string data type. The string data type without any size limit specifies a 4 byte integer pointer which can hold the address of a string constant. It cannot be used as a variable string memory location, it can only be used for dealing with the string constants. The unspecified size limit string data types will result in a runtime error if they are used with any operation which requires a string memory, for example the scan statement. They cannot be scanned using the scan statement. Hence the following code segment will give a runtime error.

```
str a|          # declaring a string variable with size limit not defined
scan a|         # results in a runtime error
```

This error can be solved by using size limit while declaring the string data type. A size limit in string data type specifies how many characters the string can contain. It is in contrast to other data types where the size limits specifies the space in memory the variable will take. For example, the following code will declare a string variable with size limit 20, so it can store up to 20 characters.

```
str:20 a|       # string a can store up to 20 characters.
```

So the error which was produced earlier can be solved using the following code.

```
str:10 a|       # declaring a string variable with storage up to 10 characters
scan a|         # no runtime error
```

## 9.8 Instructions

Tiny supports the following 4 kinds of instructions:

1. Type Declaration Instruction – It is used to declare the type of variables used in a program.
2. Control Instruction – It is used to control the sequence of execution of various statements in a program.
3. Arithmetic Instruction – It is used to perform arithmetic operations on constants and variables used in a program.
4. Input/Output (IO) Instruction – It is used to take input from the keyboard (standard input device) and give output to the console (standard output device).



### 9.8.1 Type Declaration Instruction

This instruction is used to declare the type of variables being used in the program. Any variable used in the program must be declared before using it in any statement.

Example: `int abc|`

`char p|`

There are various variations of the type declaration statement, these are discussed below:

1. Multiple variables can be declared in a single line using the comma (,) operator.  
Ex: `int a,b,c|`
2. Initialization to a constant is allowed while declaring variables.  
Ex: `int a=2|`
3. Initialization to another variable is allowed while declaring variables.  
Ex: `int b=2, a=b|`
4. Initialization to an expression is not allowed.  
Ex: `int b=3,a=b+2|` # not allowed
5. The order of declaration is from left to right.  
Ex: `int a,b|` # first declares a and then b

In Tiny, a variable can be declared anywhere in the program, and only after the declaration can it be used in the rest of the program. Using a variable without declaring it first will result in a semantic error.

### 9.8.2 Control Instruction

Control instructions determine the flow of control in a program. They enable us to specify the order in which the various instructions in a program are to be executed. Tiny is a sequential language i.e. all the statements and the instructions are executed line by line, one after another. This flow of control can be altered using control instructions in the following two ways:

1. Decision Control Instruction
2. Loop Control Instruction

#### 9.8.2.1 Decision Control Instructions (if – else – fi statements)

Tiny uses the **if** – **fi** keywords to implement the decision control instruction. The general form of **if** – **fi** keywords is given in Fig. 9.2.

<pre> if (condition)     ...statement 1...     ...statement 2...     ...statement 3... fi </pre>
--

Fig. 9.2 – if -fi keywords

The **if** statement will evaluate the condition, if the condition is true then it will execute the set of given statements until it finds the **fi** keyword, otherwise it will skip them. In Tiny, each **if** statement has to be closed using the **fi** keyword, failing to do so will result in a syntax error.

The second form of decision control instructions also uses an **else** statement. The general form of **if - fi - else** statement is given in Fig. 9.3.

```

if (condition)
    ...statement 1...
    ...statement 2...
    ...statement 3...
else
    ...statement 4...
    ...statement 5...
    ...statement 6...
fi

```

Fig. 9.3 – if-fi-else keywords

Again, the **if** statement will evaluate the condition, if the condition is true then it will execute the set of given statements until it finds the **else** keyword, otherwise it will skip them and will instead execute the statements given below the **else keyword**. Again the whole **if – else** statements have to be closed using the **fi** keyword, otherwise it will result in a syntax error.

The condition may or may not be placed in parenthesis, however it is recommended to use parenthesis for the sake of clarity and understand ability. The decision control instructions can be nested within each other up to any level.

#### 9.8.2.2 Loop Control Instruction

In Tiny, loops are used for repeating some portion of the program until a particular condition is being satisfied. Looping in Tiny is done through **loop - pool** statements. The general form of **loop - pool** statement is given in Fig. 9.4.

```

loop (condition)
    ...statement 1...
    ...statement 2...
    ...statement 3...
pool

```

Fig. 9.4 – loop-pool keywords

The **loop** statement will evaluate the condition, if the condition is true then it will execute the set of given statements until it finds the **pool** keyword. However, unlike the decision control statement, the loop will keep on repeatedly executing the statements until the

condition becomes false. The loop will break if and only if the condition becomes false. Proper caution must be taken in updating the conditions of the loop to prevent it from going into the infinite loop.

The condition may or may not be placed in parenthesis, however it is recommended to use parenthesis for the sake of clarity and understand ability. The loop control instructions can also be nested within each other up to any level.

### 9.8.3 Arithmetic Instruction

An arithmetic instruction consists of a variable on the left hand side of '=' and variables and/or constants on the right hand side of '=' which are connected by operators various operators.

Example:      `c=a+b|`            #addition operator  
                  `c=a<b|`            #Less than relational operator  
                  `c=a&&b|`            #logical AND operator

#### 9.8.3.1 Operators

Tiny supports the following types of operators which can be applied on characters, integers and real variables and constants:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment operator
5. Unary Minus Operator
6. Variable Size Operator

Table 9.5 shows these standard Tiny operators with remarks.

Table 9.5 – Standard Tiny Operators

Operator	Operator Type	Remarks
-	Unary Minus	Standard unary minus operator Ex: a=-b
=	Assignment	Assigns the value of RHS to LHS Ex: a=b
:	Variable Size	Sets the size of the variable in its declaration Ex: int:2 x
+	Arithmetic	Addition operator Ex: a=2+3
-	Arithmetic	Subtraction operator Ex: a=b-3
*	Arithmetic	Multiplication operator Ex: a=b*3

/	Arithmetic	Division operator Ex: a=b/3
%	Arithmetic	Standard modulus operator Ex: a=b%2
<	Relational	Less than operator Ex: a=3<4
<=	Relational	Less than or equal to operator Ex: a=b<=4
>	Relational	Greater than operator Ex: a=4>3
>=	Relational	Greater than or equal to operator Ex: a=b>=4
==	Relational	Equals to operator Ex: a==b
!=	Relational	Not equal to operator Ex: a=b!=2
	Logical	Logical OR operator Ex: a=b  c
&&	Logical	Logical AND operator Ex: a=b&&c
^^	Logical	Logical XOR operator Ex: a=b^^c

The above mentioned operators are applicable on char, int or real data types. For strings, Tiny supports the following operators:

1. String Assignment Operators
2. String Compare Operators
3. String Concatenation Operators
4. Characters Count Operator

Table 9.6 shows these string operators in Tiny with remarks.

Table 9.6 – String Operators in Tiny

Operator	Operator Type	Remarks
=	String Assignment	String assignment operator Assigns contents of string at RHS to string at LHS Ex: a=b
==	String Compare	Equals to operator Compares the contents of two string, returns 1 if the contents are same, else returns 0 Ex: a==b
!=	String Compare	Not equal to operator Compares the contents of two string, returns 0 if the contents are same, else returns 1 Ex: a!=b

+	String Concatenation	String concatenation operator Concatenates the contents of the two strings and stores the resultant string in the first string Ex: a=b+c
:	Characters Count	String character count operator Sets the maximum number of characters a string can contain in its declaration Ex: str:10 x

### 9.8.3.2 Operator Precedence and Associativity

Every operator in Tiny has equal precedence and the associativity of every operator is from right to left. So an expression written as 'a=b\*c+d<4-6&&e' will be evaluated as (a=(b\*(c+(d<(4-(6&&e)))))).

The precedence of operators can be modified by using a pair of explicit parenthesis around the expression. For example, without parenthesis a=b\*c+d will be evaluated as a=(b\*(c+d)), which is against the BODMAS rule. But if we apply parenthesis around '\*' operator then it will be evaluated before '+', hence a=(b\*c)+d will make the expression compatible with the BODMAS rule.

### 9.8.3.3 Workarounds for Logical NOT operation

While Tiny does not have the native logical NOT operators like it has native logical OR, logical AND & logical XOR operators, it is not impossible to use NOT functionality in Tiny.

Logical XOR operator can be used intelligently to function like a logical NOT operators as logically XORing 1 with any value will result in the logical NOT of that value. Ex: 1 (XOR) 1 = 0 and NOT(1) = 0, similarly 1 (XOR) 0 = 1 and NOT(0) = 1. So the following code in Tiny will do the logical NOT of a given variable 'x'.

```
Ex:  int x,not|      #defining a variable x and the result variable not
      not= 1^^x|     #not will now contain the logical NOT of x
```

### 9.8.4 Input/Output (IO) Instructions

Tiny uses the **print** keyword for outputting contents to the standard output device (console). Similarly it uses **scan** keyword for taking input from the standard input device (keyboard). The **print** statement can have the following two forms:

#### 1. Print only string constant

Ex: print "Hello, this is Tiny"

#### 2. Print any combination of strings and variables (separated by comma)

Ex: print "My first name is ", fname," and my last name is ",lname|

The **scan** statement can be used to input any number of variables, a comma is used for variable separation.

```
Ex:  scan a|        #inputting a single variable
      scan a,b|      #inputting two variables
```

```
scan a,b,c|    #inputting three variables
```

In case of inputting multiple variables, the newline character acts as a delimiter. So the second variable may be inputted after the first variable by pressing the ‘enter’ key and so on.

## 9.9 Escape Sequences

Escape sequences are special backslash character constants who allow easy entering of various special characters as constants which are not possible otherwise.

Tiny supports all the standard escape sequences. Table 9.7 shows these escape sequences along with their meaning.

Table 9.7 - Escape Sequences

Escape Sequence (Code)	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
'	Single quote
\\	Backslash
\v	Vertical tab
\a	Alert
\?	Question mark

## 9.10 Statements and Expressions

Operators, constants and variables are the constituents of expressions. An expression in Tiny is any valid combination of these elements.

Statements in Tiny are made up of Input/Output (IO) Instructions and assignment expressions. All the statements end with a ‘|’ symbol, failing to do so will result in a syntax error.

## 9.11 Type Conversion

Tiny does automatic type conversion of variables and constants according to the following rules:

1. An operation between a character and character always yields a character result.
2. An operation between a character and integer always yields an integer result.
3. An operation between a character and a real always yields a real result.

4. An operation between an integer and integer always yields an integer result.
5. An operation between an integer and a real always yields a real result.
6. An operation between a real and real always yields a real result.
7. An operation between a string and a string always yield a string result.
8. Any operation between a string and a non-string type is not allowed. Doing so will produce a semantic error.

## 9.12 Arrays

Arrays are a set of similar data types, stored in contiguous memory locations. Tiny supports One-Dimensional (1-D) Arrays for all the data types.

### 9.12.1 Array Declaration

Arrays are declared using the following syntax:

`<data type>:<variable size> <array name>[<array size>] |`

Example: `int x[5]`      #declaring an integer array of default size 4 bytes with array size 5

`int:2 y[10]`      #declaring an integer array of size 2 bytes with array size 10

Arrays can also be initialized while declaring them, doing so will initialize the first array element to the assigned value and the rest of the array elements to zero.

Example: `int p[4]=7`      #first element will be assigned the value 7, rest are filled with 0

### 9.12.2 Accessing Array Elements

The elements in an array may be accessed after it has been declared. This is done with subscript, the number in the brackets following the array name. This number specifies the element's position in the array. All the array elements are numbered, starting with 0. Hence, `a[0]` refers to the first element of the array and similarly `a[n-1]` refers to the  $n^{\text{th}}$  element in the array named 'a'.

## 9.13 A Few Programs in Tiny

### 9.13.1 Program to Concatenate Two Strings

```
enter
    str:20 a,b|
    print "Enter String 1\n"|
    scan a|
    print "Enter String 2\n"|
    scan b|
    a=a+b|
    print "The concatenated string is ",c|
exit
```

**Output**

```

Enter String 1
Greater
Enter String 2
Noida
The concatenated string is GreaterNoida

```

**9.13.2 Program to Compare Two Numbers**

```

enter
    int num1,num2|
    print "Enter number 1 and number 2\n"|
    scan num1,num2|
    if num1>num2
        print "Number 1 is greater than number 2"|
    else
        if num1<num2
            print "Number 2 is greater than number 1"|
        else
            print "Number 1 and number 2 are equal"|
        fi
    fi
exit

```

**Output**

```

Enter number 1 and number 2
-4
7
Number 2 is greater than number 1

```

**9.13.3 Program to Print Reverse of a Number**

```

enter
    int num,rev|
    print "Enter the number\n"|
    scan num|
    print "The reverse of ",num," is "|
    loop num!=0
        rev=(num%10)|
        print rev|
        num=num/10|

```



```

    pool
exit

```

### **Output**

```

Enter the number
12345
The reverse of 12345 is 54321

```

### **9.13.4 Program to Print Sum Up to a Given Number**

```

enter
    int sum,num,temp|
    print "Enter number\n"|
    scan num|
    if num<=0
        print "Number is invalid\n"|
    else
        temp=num|
        loop temp
            sum=sum+temp|
            temp=temp-1|
        pool
        print "Sum upto ",num," is ",sum|
    fi
exit

```

### **Output**

```

Enter number
10
Sum upto 10 is 55

```

### **9.13.5 Program to Input & Display Array**

```

enter
    int var[5],i|
    print "Enter the array\n"|
    loop i<5
        scan var[i]|
        i=i+1|
    pool
    print "The entered array is:\n"|
    i=0|

```

```
    loop i<5
        print var[i],"\n"|
        i=i+1|
    pool
exit
```

### **Output**

```
Enter the array
1
2
3
4
5
The entered array is:
1
2
3
4
5
```

## CHAPTER 10

### IMPLEMENTATION

This chapter shows the implementation of the compiler in Microsoft Windows NT. The implementation shows the usage and output of each and every feature offered by the compiler. Usage and implementation of the compiler in Linux is also similar.

#### 10.1 Program (Error Free)

The source code of a sample program has been shown in Fig. 10.1. The program finds the sum up to a given number.

```
#Program to find the sum up to a given number, contains no errors

enter
    int sum,num,temp|
    print "Enter number\n"|
    scan num|
    if num<=0
        print "Number is invalid\n"|
    else
        temp=num|
        loop temp
            sum=sum+temp|
            temp=temp-1|
        pool
        print "Sum upto ",num," is ",sum|
    fi
exit
```

Fig. 10.1 – Sample Program (Error Free)

The program first inputs the number from user, then checks if the number is less than or equal to zero. If true, then it aborts the program by printing an invalid number message. If the number is greater than zero, then it calculates the sum from 1 up to that number and then displays it to the console.

##### 10.1.1 Displaying Help Menu

The help menu can be displayed by passing “-h” command line argument to TCC as shown in Fig. 10.2.

```

C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc -h
Tiny Constructive Compiler <TCC>
Copyright <C> 2014 - Kushagra Gaur & Sonu Kushwaha

Usage: tcc <option> <source file>

Options:
-h      Display help
-t      Display only tokens, don't proceed further
-p      Check only syntax & display parse tree, don't proceed further
-s      Check only semantics, don't proceed further
-i      Display only Intermediate Representation <IR>, don't proceed further
-a      Generate only assembly file, don't create executable
-e      Generate assembly file with executable <requires GCC>

By default, both the assembly file & the executable is generated <requires GCC>
Mail bug reports to <tcc.developers@gmail.com>

G:\TCC_FinalYearProject\Binary_WINNT>_

```

Fig. 10.2 – Displaying Help Menu

### 10.1.2 Displaying Program Tokens

The program lexemes as well as the corresponding tokens can be displayed using “-t” command line argument as shown in Fig. 10.3.

```

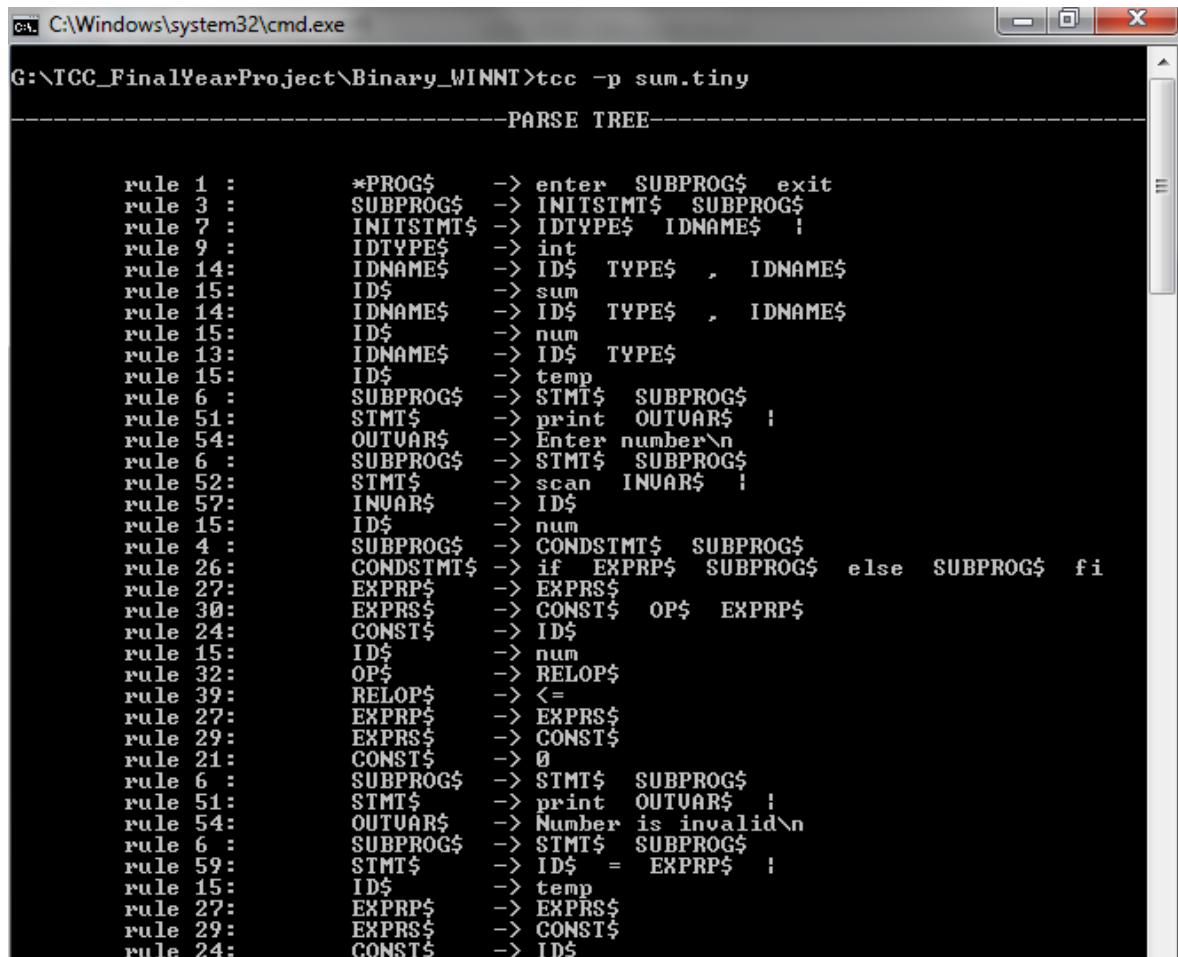
C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc -t sum.tiny
-----TOKENS GENERATED-----
[lexeme]:  enter      ->  enter      :[token]
[lexeme]:  int        ->  int        :[token]
[lexeme]:  sum        ->  IDENTIFIER :[token]
[lexeme]:  ,          ->  COMOP      :[token]
[lexeme]:  num        ->  IDENTIFIER :[token]
[lexeme]:  ,          ->  COMOP      :[token]
[lexeme]:  temp       ->  IDENTIFIER :[token]
[lexeme]:  !          ->  ENDOP      :[token]
[lexeme]:  print      ->  print      :[token]
[lexeme]:  Enter number\n ->  STRCONST   :[token]
[lexeme]:  !          ->  ENDOP      :[token]
[lexeme]:  scan       ->  scan       :[token]
[lexeme]:  num        ->  IDENTIFIER :[token]
[lexeme]:  !          ->  ENDOP      :[token]
[lexeme]:  if         ->  if         :[token]
[lexeme]:  num        ->  IDENTIFIER :[token]
[lexeme]:  <=         ->  LTEOP      :[token]
[lexeme]:  0          ->  INTCONST   :[token]
[lexeme]:  print      ->  print      :[token]
[lexeme]:  Number is invalid\n ->  STRCONST   :[token]
[lexeme]:  !          ->  ENDOP      :[token]
[lexeme]:  else       ->  else       :[token]
[lexeme]:  temp       ->  IDENTIFIER :[token]
[lexeme]:  =          ->  ASSOP      :[token]
[lexeme]:  num        ->  IDENTIFIER :[token]
[lexeme]:  !          ->  ENDOP      :[token]
[lexeme]:  loop       ->  loop       :[token]
[lexeme]:  temp       ->  IDENTIFIER :[token]
[lexeme]:  sum        ->  IDENTIFIER :[token]
[lexeme]:  =          ->  ASSOP      :[token]
[lexeme]:  sum        ->  IDENTIFIER :[token]
[lexeme]:  +          ->  ADDOP      :[token]
[lexeme]:  temp       ->  IDENTIFIER :[token]
[lexeme]:  !          ->  ENDOP      :[token]
[lexeme]:  temp       ->  IDENTIFIER :[token]
[lexeme]:  =          ->  ASSOP      :[token]
[lexeme]:  temp       ->  IDENTIFIER :[token]
[lexeme]:  -          ->  SUBOP      :[token]

```

Fig. 10.3 – Displaying Tokens

### 10.1.3 Displaying Parse Tree

The parse tree representation of the program can be displayed using “-p” command line argument as shown in Fig. 10.4.



```

C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc -p sum.tiny

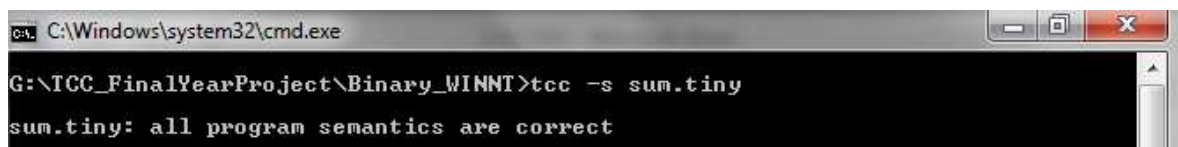
-----PARSE TREE-----

rule 1 :      *PROG$      -> enter SUBPROG$ exit
rule 3 :      SUBPROG$    -> INITSTMT$ SUBPROG$
rule 7 :      INITSTMT$   -> IDTYPE$ IDNAME$ !
rule 9 :      IDTYPE$     -> int
rule 14:      IDNAME$     -> ID$ TYPE$ , IDNAME$
rule 15:      ID$         -> sum
rule 14:      IDNAME$     -> ID$ TYPE$ , IDNAME$
rule 15:      ID$         -> num
rule 13:      IDNAME$     -> ID$ TYPE$
rule 15:      ID$         -> temp
rule 6 :      SUBPROG$    -> STMT$ SUBPROG$
rule 51:      STMT$       -> print OUTVAR$ !
rule 54:      OUTVAR$     -> Enter number\n
rule 6 :      SUBPROG$    -> STMT$ SUBPROG$
rule 52:      STMT$       -> scan INVAR$ !
rule 57:      INVAR$     -> ID$
rule 15:      ID$         -> num
rule 4 :      SUBPROG$    -> CONDSMT$ SUBPROG$
rule 26:      CONDSMT$    -> if EXPRP$ SUBPROG$ else SUBPROG$ fi
rule 27:      EXPRP$      -> EXPRP$
rule 30:      EXPRP$      -> CONST$ OP$ EXPRP$
rule 24:      CONST$     -> ID$
rule 15:      ID$         -> num
rule 32:      OP$         -> RELOP$
rule 39:      RELOP$      -> <=
rule 27:      EXPRP$      -> EXPRP$
rule 29:      EXPRP$      -> CONST$
rule 21:      CONST$     -> 0
rule 6 :      SUBPROG$    -> STMT$ SUBPROG$
rule 51:      STMT$       -> print OUTVAR$ !
rule 54:      OUTVAR$     -> Number is invalid\n
rule 6 :      SUBPROG$    -> STMT$ SUBPROG$
rule 59:      STMT$       -> ID$ = EXPRP$ !
rule 15:      ID$         -> temp
rule 27:      EXPRP$      -> EXPRP$
rule 29:      EXPRP$      -> CONST$
rule 24:      CONST$     -> ID$
  
```

Fig. 10.4 – Displaying Parse Tree

### 10.1.4 Checking Program Semantics

The program semantics (meaning) can be checked by using the “-s” command-line argument as shown in Fig. 10.5. It verifies if the source program is semantically correct or not.



```

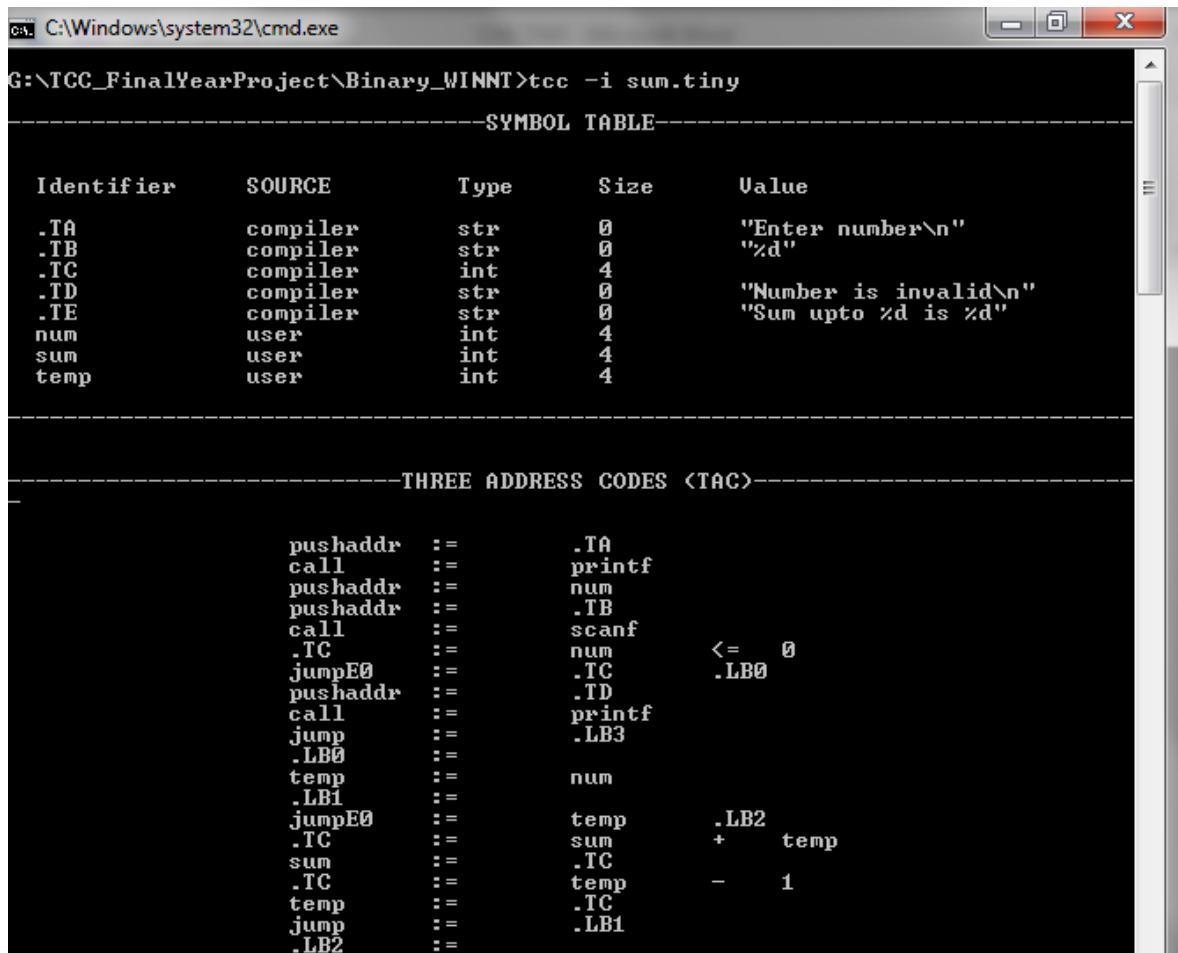
C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc -s sum.tiny

sum.tiny: all program semantics are correct
  
```

Fig. 10.5 – Check Program Semantics

### 10.1.5 Displaying Symbol Table & Program Intermediate Representation (IR)

The program symbol table as well as the IR code in the form of quadruples (Three Address Codes) can be displayed using the “-i” command line argument as shown in Fig. 10.6.



```

C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc -i sum.tiny

-----SYMBOL TABLE-----

Identifier    SOURCE      Type      Size      Value
.TA           compiler   str       0         "Enter number\n"
.TB           compiler   str       0         "%d"
.TC           compiler   int       4
.TD           compiler   str       0         "Number is invalid\n"
.TE           compiler   str       0         "Sum upto %d is %d"
num           user       int       4
sum           user       int       4
temp          user       int       4

-----THREE ADDRESS CODES (TAC)-----

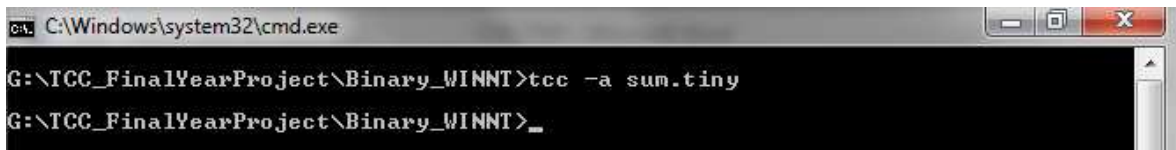
pushaddr := .TA
call := printf
pushaddr := num
pushaddr := .TB
call := scanf
.TC := num <= 0
jumpE0 := .TC .LB0
pushaddr := .TD
call := printf
jump := .LB3
.LB0 := num
temp := num
.LB1 := temp
jumpE0 := .LB2
.TC := sum + temp
sum := .TC
.TC := temp - 1
temp := .TC
jump := .LB1
.LB2 :=
  
```

Fig. 10.6 – Display Symbol Table & IR

### 10.1.6 Generating Intel x86 Assembly Code (GNU Syntax)

The target assembly code can be generated by using the command-line argument “-a” as shown in Fig. 10.7. This argument will not attempt to create the executable code, it will only create the assembly file. The syntax of the target assembly code is the GNU syntax. The target assembly code (Windows NT) for the sample input has been shown in Fig. 10.8.

The name of the assembly file is same as that of the source file except the “.s” extension. For example, if the name of the input source file is “sum.tiny” then the name of the output assembly file is “sum.s”.



```

C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc -a sum.tiny
G:\TCC_FinalYearProject\Binary_WINNT>_

```

Fig. 10.7 – Generating Assembly Code

```

#-----Intel x86 assembly code (AT&T syntax) - to be used with GNU assembler
& linker (Windows NT - 32 bit only)-----#

        .file    "sum.s"

# Declaring program variables:

.TA:
        .ascii   "Enter number\n\0"
.TB:
        .ascii   "%d\0"
        .comm    .TC, 4
.TD:
        .ascii   "Number is invalid\n\0"
.TE:
        .ascii   "Sum upto %d is %d\0"
        .comm    num, 4
        .comm    sum, 4
        .comm    temp, 4
.AT:
        .ascii   "\0"

# Finished declaring program variables

        .text
        .globl   _main
_main:

# Printing epilog:

        pushl    %ebp
        movl     %esp, %ebp

# Epilog finished

# Program Instructions:

        pushl    $.TA
        call     _printf
        pushl    $num
        pushl    $.TB

```

```

    call    _scanf
    movl    num, %eax
    cmpl    $0, %eax
    setle   %al
    movzbl %al, %eax
    movl    %eax, .TC
    movl    .TC, %eax
    testl   %eax, %eax
    je      .LB0
    pushl    $.TD
    call    _printf
    jmp     .LB3
.LB0:
    movl    num, %eax
    movl    %eax, temp
.LB1:
    movl    temp, %eax
    testl   %eax, %eax
    je      .LB2
    movl    sum, %eax
    addl    temp, %eax
    movl    %eax, .TC
    movl    .TC, %eax
    movl    %eax, sum
    movl    temp, %eax
    subl    $1, %eax
    movl    %eax, .TC
    movl    .TC, %eax
    movl    %eax, temp
    jmp     .LB1
.LB2:
    pushl    sum
    pushl    num
    pushl    $.TE
    call    _printf
.LB3:

    leave
    ret

```

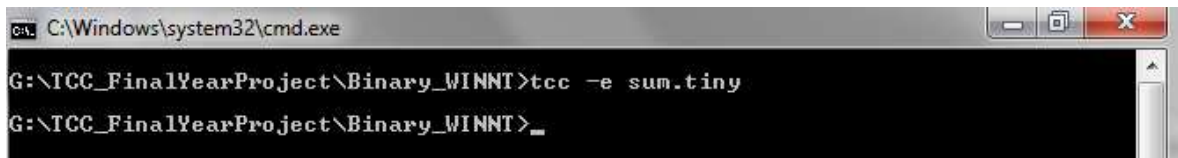
Fig. 10.8 – Target Assembly Code for Sample Input

### 10.1.7 Generating Intel x86 Assembly Code with Executable

The assembly code as well as the binary executable code can be generated by using the “-e” command line argument as shown in Fig. 10.9. The executable code will only be generated if the GNU Assembler and Linker are found on the system otherwise only the assembly file will be generated. This is also the default behaviour of TCC if no command line options are



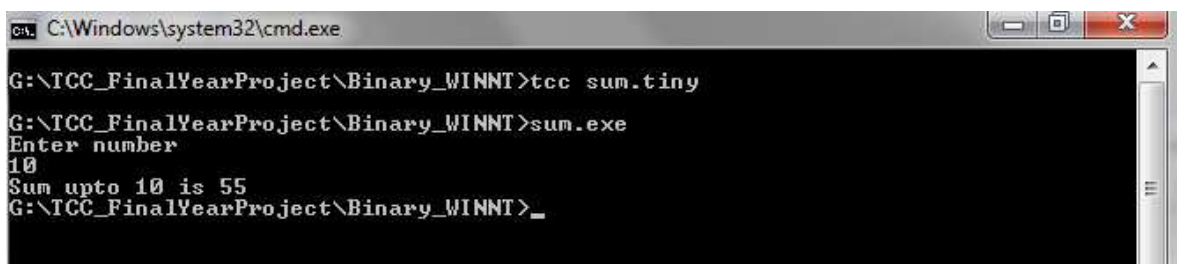
passed. GNU Assembler & Linker are installed by default on Linux System whereas they can be installed in Microsoft Windows NT by installing GCC for Windows.



```
C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc -e sum.tiny
G:\TCC_FinalYearProject\Binary_WINNT>_
```

Fig. 10.9 – Generating Assembly Code with Executable

The executable generated is of the same name as the source file except that its extension will be “.exe” in Windows and it will be extension less in Linux. Fig. 10.10 shows how to generate and run the executable in Windows NT.



```
C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc sum.tiny
G:\TCC_FinalYearProject\Binary_WINNT>sum.exe
Enter number
10
Sum upto 10 is 55
G:\TCC_FinalYearProject\Binary_WINNT>_
```

Fig. 10.10 – Running the Executable

## 10.2 Program (Having Errors)

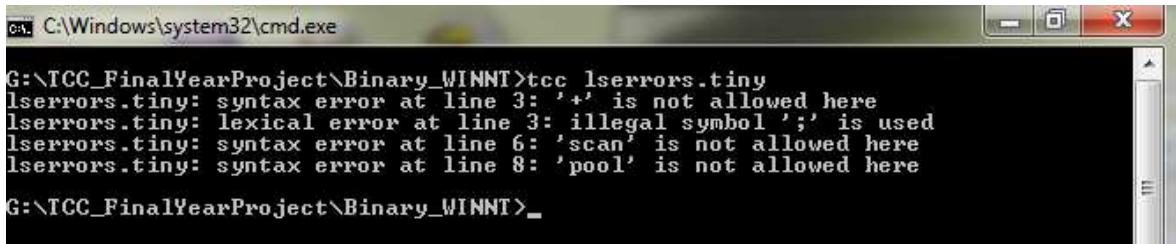
### 10.2.1 Program Having Lexical & Syntax Errors

The Fig. 10.11 shows a sample Tiny program having a few lexical and syntax errors.

```
#Program having syntax & lexical errors
enter
    + int var[5],i| ;
    print "Enter the array\n"|
    loop i<5 =
        scan var[i]| i=i+1|
    pool
exit
```

Fig. 10.11 – Sample Program with Lexical & Syntax Errors

Fig. 10.12 shows the output of the above program.



```

C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc lerrors.tiny
lerrors.tiny: syntax error at line 3: '+' is not allowed here
lerrors.tiny: lexical error at line 3: illegal symbol ';' is used
lerrors.tiny: syntax error at line 6: 'scan' is not allowed here
lerrors.tiny: syntax error at line 8: 'pool' is not allowed here
G:\TCC_FinalYearProject\Binary_WINNT>_

```

Fig. 10.12 – TCC Catches the Lexical & Syntax Errors

### 10.2.2 Program Having Semantic Errors

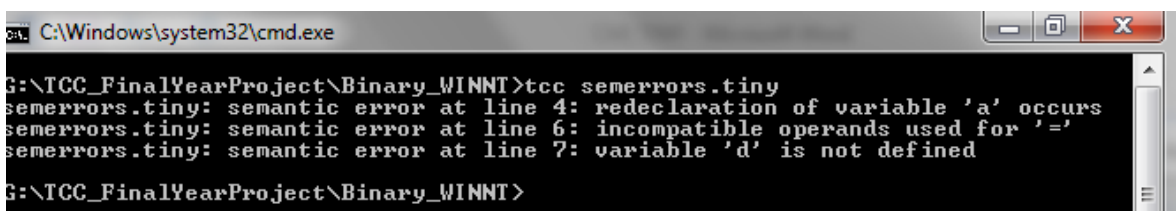
The Fig. 10.13 shows a sample Tiny program having a few semantic errors. Fig. 10.14 shows its output.

```

#Program having semantic errors
enter
    char a|
    int a,b|      #variable a is re declared
    str c|
    c=b+2|      #b+2 is of int type & c is of string type
    print d|     #d is not defined before using
exit

```

Fig. 10.13 – Program with Semantic Errors



```

C:\Windows\system32\cmd.exe
G:\TCC_FinalYearProject\Binary_WINNT>tcc semerrors.tiny
semerrors.tiny: semantic error at line 4: redeclaration of variable 'a' occurs
semerrors.tiny: semantic error at line 6: incompatible operands used for '='
semerrors.tiny: semantic error at line 7: variable 'd' is not defined
G:\TCC_FinalYearProject\Binary_WINNT>

```

Fig. 10.14 – TCC Catches the Semantic Errors

## CHAPTER 11

### CONCLUSION

Tiny Constructive Compiler (TCC) is an entry level attempt of creating a basic new programming language called Tiny along with its multi feature compiler.

The Tiny language has all the features which a language should have considering its specific educational domain. The language has not been developed for serious application development, rather it is more suited for educating the people and introducing them to the programming world. The compiler which has been developed for this language is complete in all respects and offers some very unique features which are not seen in compilers elsewhere.

However, there are a few improvements which could be made to the compiler. The error messages which it generates, especially the syntax error messages, could be made more informative. A separate code optimization phase could be introduced into this compiler, for both the intermediate code and the target assembly code. The compiler uses GNU syntax for the assembly language which is the standard syntax developed by AT&T Bell Laboratories, US. Other assembly language syntaxes, such as the NASM syntax, could also be introduced to the compiler.

The compiler is able to handle floating point (real) data type only up to intermediate code generation. The final assembly code generated by the compiler does not deal with floating point numbers. This limitation would be dealt with in the future releases of TCC.

The Tiny language could also be made better by introducing a few features such as the pre mature loop break and a separate logical NOT operator. String operators, other than what the language already has, could also be introduced. These are some of the areas where the project maintenance phase would look into.

Although the compiler has been thoroughly tested against all possible test cases, it might still have bugs. The bug reports with proper reference and sufficient details regarding the source input program, the underlying hardware as well as the Operating System can be mailed to [tcc.developers@gmail.com](mailto:tcc.developers@gmail.com).

The project development team has made some serious and sincere efforts towards the project and is continuously striving to add more features to it, so that that its scope and the user base can be increased.





## APPENDIX C

### GRAMMAR RULES

1. prog\$	-> enter subprog\$ exit
2. subprog\$	-> $\epsilon$
3. subprog\$	-> initstmt\$ subprog\$
4. subprog\$	-> condstmt\$ subprog\$
5. subprog\$	-> loopstmt\$ subprog\$
6. subprog\$	-> stmt\$ subprog\$
7. initstmt\$	-> idtype idname
8. initstmt\$	-> idtype\$ : INTCONST idname\$
9. idtype\$	-> int
10. idtype\$	-> char
11. idtype\$	-> str
12. idtype\$	-> real
13. idname\$	-> id\$ type\$
14. idname\$	-> id\$ type\$ , idname\$
15. id\$	-> IDENTIFIER
16. id\$	-> IDENTIFIER [ exprs\$ ]
17. type\$	-> $\epsilon$
18. type\$	-> = const\$
19. type\$	-> = - const\$
20. const\$	-> STRCONST
21. const\$	-> INTCONST
22. const\$	-> CHARCONST
23. const\$	-> REALCONST
24. const\$	-> id\$
25. condstmt\$	-> if exprp\$ subprog\$ fi
26. condstmt\$	-> if exprp\$ subprog\$ else subprog\$ fi
27. exprp\$	-> exprs\$
28. exprp\$	-> - exprs\$
29. exprs\$	-> const\$
30. exprs\$	-> const\$ op\$ exprp\$
31. const\$	-> (exprp\$)
32. op\$	-> relop\$
33. op\$	-> alop\$
34. op\$	-> logop\$
35. op\$	-> =
36. relop\$	-> <
37. relop\$	-> >
38. relop\$	-> <=
39. relop\$	-> >=
40. relop\$	-> ==
41. relop\$	-> !=
42. alop\$	-> +
43. alop\$	-> -

44. alop\$	-> *
45. alop\$	-> /
46. alop\$	-> %
47. logop\$	-> ^^
48. logop\$	-> &&
49. logop\$	->
50. loopstmt\$	-> loop exprp\$ subprog\$ pool
51. stmt\$	-> print outvar\$
52. stmt\$	-> scan invar\$
53. outvar\$	-> id\$
54. outvar\$	-> STRCONST
55. outvar\$	-> id\$ , outvar\$
56. outvar\$	-> STRCONST , outvar\$
57. invar\$	-> id\$
58. invar\$	-> id\$ , invar\$
59. stmt\$	-> id\$ = exprp\$

**Notes:**

1. 'prog\$' is the start symbol.
2. 'ε' denotes the epsilon.
3. Variables suffixed with '\$' denote non-terminals, ex: subprog\$, id\$, op\$, etc.
4. Rest of the symbols are all terminals, ex: enter, +, |, STRCONST, etc.

## APPENDIX D

### PHASES OF COMPILER DESIGN

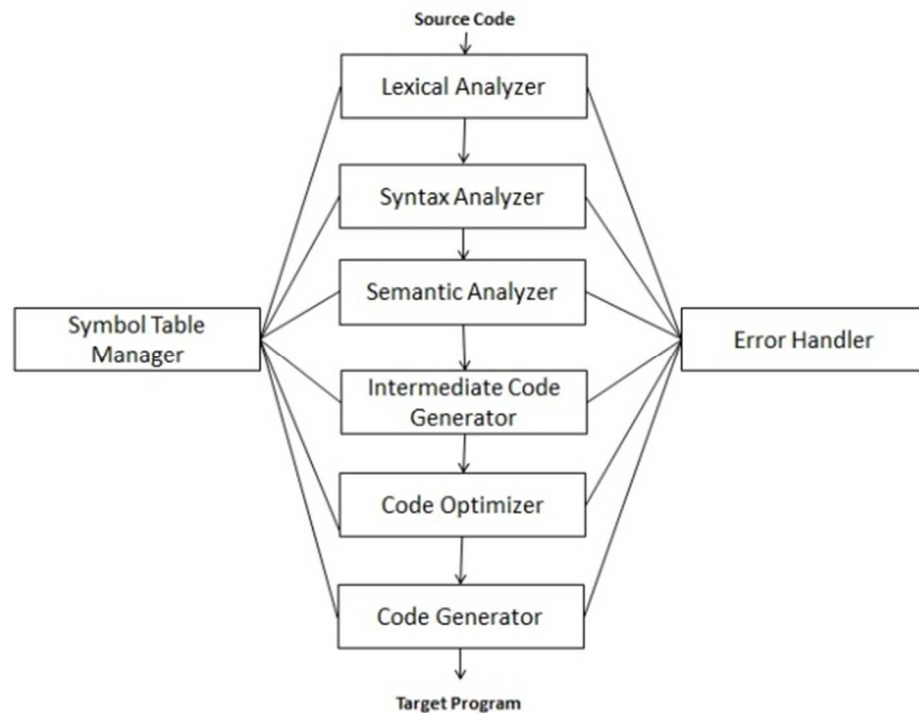


Fig. D.1 – Phases of Compiler Design

### 1. Lexical Analysis

In lexical Analysis:

- a) A stream of characters from the source file is read left to right and grouped into tokens that are sequence of characters having a collective meaning.
- b) These tokens are returned one by one when required by the parser to make parse trees or the code generator, as and when required.

#### 1.1 Lexical Grammar

The specification of a programming language often includes a set of rules which defines the lexical analyser. These rules usually consist of regular expressions, and they define the set of possible character sequences that are used to form individual tokens or lexemes.



## 1.2 Tokens

A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA etc.). The process of forming tokens from an input stream of characters is called tokenization, and the lexical analyser categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

## 1.3 Lexeme

A lexeme, however, is only a string of characters known to be of a certain kind (e.g., a string literal, a sequence of letters). In order to construct a token, the lexical analyser needs a second stage, the evaluator, which goes over the characters of the lexeme to produce a value. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser.

## 1.4 Pattern

Patterns are the rules describing a set of lexemes belonging to a token. Ex: “letter followed by letters & digits” & “non-empty sequence of digits”, etc.

# 2. Syntax Analysis

In computer science and linguistics, parsing, or, more formally, syntactic analysis, is the process of analysing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar. Parsing can also be used as a linguistic term, for instance when discussing how phrases are divided up in garden path sentences.

Parsing is also an earlier term for the diagramming of sentences of natural languages, and is still used for the diagramming of inflected languages, such as the Romance languages or Latin. The term parsing comes from Latin *pars*, meaning part (of speech).

Parsing is a common term used in psycholinguistics when describing language comprehension. In this context, parsing refers to the way that human beings, rather than computers, analyse a sentence or phrase (in spoken language or text) "in terms of grammatical constituents, identifying the parts of speech, syntactic relations, etc." This term is especially common when discussing what linguistic cues help speakers to parse garden-path sentences.

Fig. D.2 shows the steps of parsing.

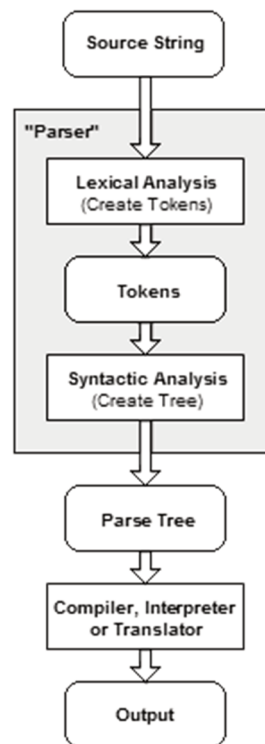


Fig. D.2 – Steps of Parsing

## 2.1 Types of Parsers

There are the following 2 types of parsers:

1. Top Down Parser
2. Bottom Up Parser

### 2.1.1 Top Down Parser

Top-down parsing is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. LL parsers are a type of parser that uses a top-down parsing strategy. Top-down parsing is a strategy of analysing unknown data relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. It occurs in the analysis of both natural languages and computer languages. Top-down parsing can be viewed as an attempt to find left-most derivations of an input-stream by searching for parse-trees using a top-down expansion of the given formal grammar rules. Tokens are consumed from left to right. Inclusive choice is used to accommodate ambiguity by expanding all alternative right-hand-sides of grammar rules.

Fig. D.3 shows top down parsing using a parse tree.

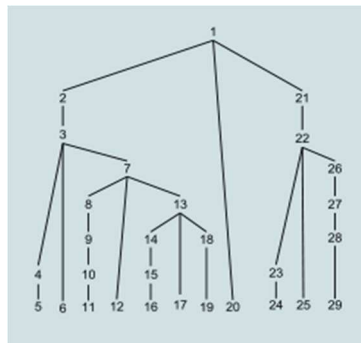


Fig. D.3 – Top Down Parsing

### 2.1.2 Bottom Up Parsing

The bottom-up name comes from the concept of a parse tree, in which the most detailed parts are at the bushy bottom of the (upside-down) tree, and larger structures composed from them are in successively higher layers, until at the top or "root" of the tree a single unit describes the entire input stream. A bottom-up parse discovers and processes that tree starting from the bottom left end, and incrementally works its way upwards and rightwards. A parser may act on the structure hierarchy's low, mid, and highest levels without ever creating an actual data tree; the tree is then merely implicit in the parser's actions. Bottom-up parsing lazily waits until it has scanned and parsed all parts of some construct before committing to what the combined construct is.

Fig. D.4 shows bottom up parsing using a parse tree.

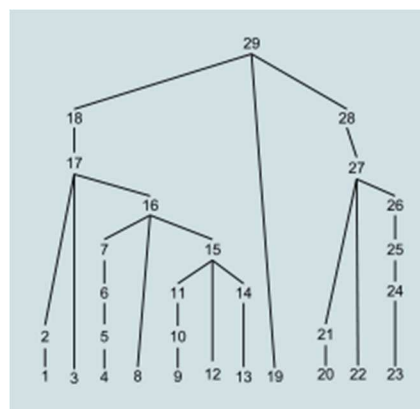


Fig. D.4 – Bottom Up Parsing

## 3. Semantic Analysis

Semantic Analysis will check for syntactically correct programs for meaningful readings. It involves two major tasks:

1. Processing the declarations and building/updating the symbol table
2. Examining the rest of the program to ensure that the identifiers are used correctly adhering to the type compatibility conventions defined by the language.

Semantic analysis also decorates the abstract parse tree generated by the syntax analysis phase with more information on the data types.

There are the following methods for semantic analysis:

1. **Translation interleaved with parsing:** The evaluation of semantics happens during parsing itself. It is efficient as no parse tree or dependency graph is created, but is only applicable for L-attributed grammar definitions.
2. **Parse tree method:** Creates a parse tree, makes a dependency graph and evaluates the attributes based on the topographic sort of the dependency graph. It is less efficient but is applicable to any grammar provided there are no cycles in the dependency graph.
3. **Rule-based method:** Creates a parse tree, traverses it in proper order to evaluate the attributes and realise the Syntax Directed Translation (SDD). The proper order is determined before compilation, by analysing the attributes and productions. It is reasonably efficient and can be applied to any grammar.

#### 4. Intermediate Code Generation

After syntax and semantic analysis, some computers generate explicit intermediate representation of source program. The intermediate expression is a program for abstract machine. This representation should have 2 important properties:

1. It should be easy to produce
2. Easier to translate into target code

Commonly this represented with instruction having one operator and 2 operands. The following methods are quite popular for intermediate representation:

1. Postfix notation
2. Three Address Codes (TAC)
3. Abstract Syntax Tree (AST)

#### 5. Final Code Generation

In this phase, the intermediate code is translated into machine or assembly code. This phase is characterized by associating memory locations with the variables and choose the appropriate assembly instructions depending on the target processor. The output of this phase is the target code in the form of assembly language of the target hardware.

## 6. Code Optimisation

Code optimization is an optional phase which deals with reducing the space and time complexities of the generated code. It is done using:

1. **Common sub expression elimination (based on available expression information):** If an expression occurs several times, compute it once and reuse the result.
2. **Partial redundancy elimination (PRE):** Perform code motion to eliminate partially redundant code, i.e. code that is at least on one path redundant.
3. **Partial dead code elimination (PDE):** Perform code motion to eliminate partially dead code i.e. code that is at least on one path dead.
4. **Loop-invariant code motion:** Move loop-invariant code out of loops.
5. **Constant propagation:** Determine the value of a variable, if it is a constant.

## 7. Symbol Table Management

A symbol table is a special table which stores various details about the identifiers used in the program such as its name, type, value, size, location, offset, etc.

This phase is linked to most other phases as the other phases often build/update the symbol table according to the source program.

## 8. Error Handling

Programs submitted to a compiler often have errors of various kinds. With regard to errors, most compilers are not very effective at communication since they deliver their comments to the user without any knowledge of the user's "intent" for a sentence. Since no user has a perfect knowledge of syntax and semantics, errors are inevitable.

The compiler detects an error, and then attempts to recover from the error so that it can detect more errors. That is, even in the presence of errors, the compiler tries to parse the entire the program in order to detect as many errors as possible.

The compiler has, as one of its parts, an error handler. The error handler is invoked when the symbols in a sentence do not match the compiler's current position in the syntax diagram. The error handler warns the programmer by issuing an appropriate message.

## APPENDIX E

### PROJECT SOURCE CODE

Appendix E contains the source code of the main module of the project. This module gives a basic idea of the flow of control to other required modules. The full project source code has not been attached here because of the large size and complexity of the code. However, the full project source could be found in the CD-ROM attached with this document.

```

/*
 * Tiny Constructive Compiler (TCC) - Main UI
 *
 * Copyright 2014 Kushagra <kg@kg-desktop>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"vardef.h"
#include"tlex.h"
#include"tparse.h"
#include"errhandle.h"
#include"tsemantic.h"
#include"symtab.h"

#ifdef WINNT

#define WINNT 0
#include"x86linux.h"

#else

#define WINNT 1

```

```

#include"x86win.h"

#endif

char *_file; //to hold source file

//To display help
inline static void help()
{
    printf("\nTiny Constructive Compiler (TCC)\nCopyright (C) 2014 - Kushagra
    Gaur & Sonu Kushwaha\n%s%s%s%s%s%s%s%s%s%s",
    "\nUsage: tcc <option> <source file>\n",
    "\nOptions:\n\n-h\tDisplay help",
    "\n-t\tDisplay only tokens, don't proceed further",
    "\n-p\tCheck only syntax & display parse tree, don't proceed further",
    "\n-s\tCheck only semantics, don't proceed further",
    "\n-i\tDisplay only Intermediate Representation (IR), don't proceed further",
    "\n-a\tGenerate only assembly file, don't create executable",
    "\n-e\tGenerate assembly file with executable (requires GCC)\n",
    "\nBy default, both the assembly file & the executable is generated (requires
    GCC)\n",
    "\nMail bug reports to <tcc.developers@gmail.com>\n\n");
    exit(0);
}

/*-----Program starting point-----*/
int main(int _argc, char *_argv[])
{
    if(_argc==1 || _argc>=4) //Wrong input
        errmain(ERRMAIN0);
    else if(*_argv[1]=='-') //if option is given
    {
        //if invalid option is given
        if(*(_argv[1]+2) || (!_argv[2] && *(_argv[1]+1)!='h'))
            goto errlabel;
        _file=_argv[2];
        switch(*(_argv[1]+1))
        {
            case 'h': help(); //Display help
            case 't': return !printtok(); //Display only tokens
            case 'p': return !tparse(TRUE); //Display parse tree only
            case 's': return !tsemantic(TRUE); //Perform semantic analysis
            case 'i': return !printIR(); //Display IR
            case 'a': if(WINNT)
                return !wincodegen(); //Generate assembly code
                return !linuxcodegen();
            case 'e': goto label; //Generate assembly code with executable
            errlabel:
            default: errmain(ERRMAIN0); //if wrong option is given
        }
    }
}

```

```

        return 1;
    }
}
_file=_argv[1]; //Generate assembly code with executable (requires GCC)
//first generate assembly file, then create executable using it
register unsigned short __status; //contains compiler status
label:
if(WINNT) //if OS is Windows NT
    __status=wincodegen();
else //if OS is Linux
    __status=linuxcodegen();
if(!__status)
    return 1;
char __gcc[50];
_file[strlen(_file)-2]=0;
if(WINNT)
    sprintf(__gcc,"gcc -o %s.exe %s.s > NUL 2>&1",_file,_file);
else
    sprintf(__gcc,"gcc -o %s %s.s >/dev/null 2>&1",_file,_file);
if(system(__gcc)) //checking if GCC exists
{
    fprintf(stderr,"\nOnly x86 assembly file '%s.s' is created\nError(s) occurred
while creating the executable, Please ensure:\n1. GNU C Compiler
(GCC) is installed on this system\n2. The correct PATH environment
variable is set for GCC\n3. The GCC used is a 32 bit compiler\n",_file);
    return 1;
}
return 0;
}

```



## REFERENCES

1. V Raghavan, Principles of Compiler Design, New Delhi, Tata McGraw Hill Education, 2010.
2. Alfred V. Aho, et. al., Compilers: Principles, Techniques and Tools, 2nd Edition, New Delhi, Pearson Education, 2007.
3. “Syntax Analysis”, class notes for CSE, Department of Computer Science, NIET, Uttar Pradesh Technical University, March 2013.
4. Wikipedia, “Compiler Construction”, [http://en.wikipedia.org/Compiler\\_construction](http://en.wikipedia.org/Compiler_construction).
5. Stack Overflow, “Learning to write a compiler”, February 2014, <http://stackoverflow.com/questions/1669/learning-to-write-a-compiler>.
6. Wikipedia, “x86 instruction listings”, [http://en.wikipedia.org/X86\\_instructions\\_listings](http://en.wikipedia.org/X86_instructions_listings).
7. Aditya Agarwal, et. al., General Cross Compiler, Department of Computer Science, NIET, Greater Noida, June 2013.