**SMALL COMPILER**

Submitted in the partial fulfillment of the requirements of the award of the degree of

**BACHELOR OF ENGINEERING**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

****

Submitted to: Mr Vikas Wassan

Submitted by: Nitesh Kumar(16BCS1253)

Harpreet Kaur(16BCS1267)

Abhinay Shukla(16BCS1284)

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**Chandigarh University, Gharuan**

**ACKNOWLEDGEMENT**

I would like to express my special thanks of gratitude to my mentor (Mr. Vikas Wassan) as well as our Dean department who gave us the golden opportunity to` this wonderful project on the topic ( COMPILER DESIGN ), which also helped me in doing a lot of Research and I came to know about so many new things. I am really thankful to them.

Secondly I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

**ABSTRACT**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

****

**Syntax Analysis**

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

**Semantic Analysis**

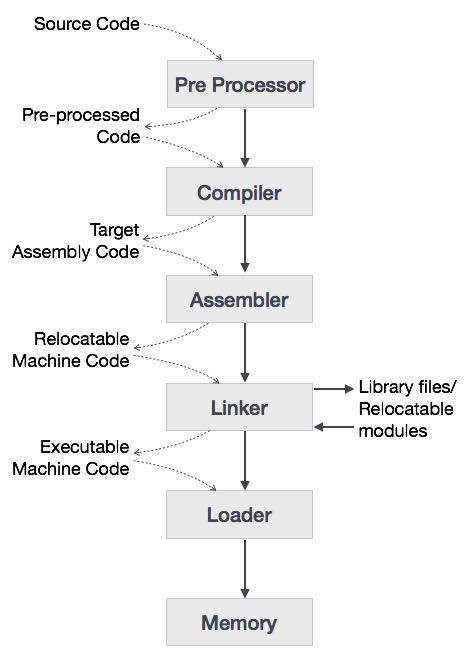
Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

**Intermediate Code Generation**

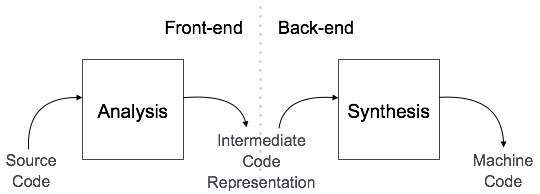
After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

**List of Figures**

**Fig 1 : Language Processing System**

****

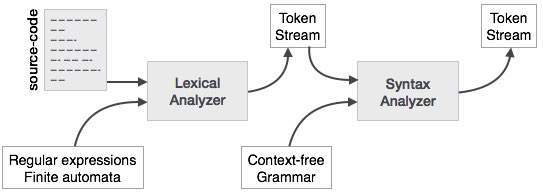
**Fig 2 : Phases of compiler**

****

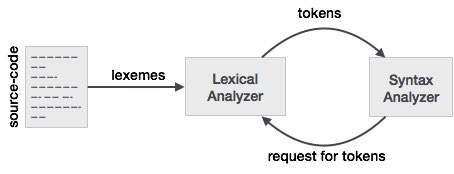
**Fig 3 : Phases of Compiler**

****

**Fig 4 : Syntax anaylser**

****

**Fig 4 : Token Passing**

****

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **S.no** | **Topic** | **Page No.** |
| 01 | Acknowledgement |  |
| 02 | Abstract |  |
| 03 | List of Figures |  |
| 04 | Table of Contents |  |
| 05 | Chapter-1 Introduction |  |
| 06 | Chapter-2 SRS |  |
| 07 | Chapter-3 Architecture Diagram |  |
| 08 | Chapter-4 Project Methodology |  |
| 09 | Chapter-5 Screenshots |  |
| 10 | Chapter-6 Conclusion and Future Scope |  |
| 11 | Refrences |  |

**Chapter-1**

**INTRODUCTION**

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space. Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end. Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and cumbersome task for computer programmers to write such codes, which is why we have compilers to write such codes.

## Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.

The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

* User writes a program in C language (high-level language).
* The C compiler, compiles the program and translates it to assembly program (low-level language).
* An assembler then translates the assembly program into machine code (object).
* A linker tool is used to link all the parts of the program together for execution (executable machine code).
* A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

**Preprocessor**

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

**Interpreter**

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

**Assembler**

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

**Linker**

Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

**Loader**

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

**Cross-compiler**

A compiler that runs on platform

(A) and is capable of generating executable code for platform

(B) is called a cross-compiler.

**Source-to-source Compiler**

A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compiler.

**Chapter-2**

**SRS**

**1. Introduction**

**1.1 Purpose**

This document provides all of the requirements for the “Small compiler”. The document primarily aims at enhancing the user’s and developer’s information about the various aspects of the project. This project is basically concerned about a small compiler which will compile small programs.

**1.2. Problem Definition**

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). Compilers are a type of translator that support digital devices, primarily computers. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

**1.3 Product Scope**

Compiler is an application software which helps user to write programs to solve problems (real life also). Code written in high level language is translated into machine level language with the help of compiler.

We are aimed at making two phases of compiler and further expanding it.

**1.4 Technique Involved**

Small compiler: Basic java programming is used in designing this compiler.

**1.5 Overview**

This document specifies SRS for “Small C like compiler” which is a program written in java language for compiling small programs or for converting high level language into machine code.

The purpose of this software is to allow the user to write the program in human understandable format and then converting into machine understandable format.

**2. Overall Description**

**2.1 Product Perspective**

**2.1.1 User interfaces**

GUI (Graphical User Interface) will be used in this application.

**2.1.2 Hardware interfaces**

This project is intended to be platform independent. Therefore no specific hardware is excluded. But it will at least work on x86 systems without any additional porting efforts. Moreover, no special hardware is needed for software operation. More details are covered in section 3.1.2.

**2.1.3 Software interfaces**

This project is intended to work on any operating system as it has been programmed in java which is a platform independent language. We have tested the software to run on windows. More details are covered in section 3.1.3.

**2.1.4 Communication interfaces**

• Network protocols for program update information.

• System I/O protocols for local file access.

**2.1.5 Memory constraints**

The project is expected to use 512 MB of RAM or more and 4 GB of external storage or more.

**2.1.6 Operations**

* Token generation.
* Error generation.
* Different tables as per requirement.
* Compilation.

**2.1.7 Site adaptation requirements**

User interface must exist in English.

**2.2 Product Functions**

Source program

Lexical analysis

Syntax analysis

Semantic analysis

Symbol Table

Error

Intermediate code generation

Code optimization

Storage assignment

Machine code generation

Machine code

**2.3 User Classes and Characteristics**

This is a small C like compiler capable of compiling small programs. It converts small programs into machine code.

**2.4 Operating Environment**

The project is intended to be operating system independent. Therefore no specific operating system is excluded. But it has been developed in windows OS. The project will use java programming.

**2.5 Design and Implementation Constraints**

Java programming is used in designing this compiler.

**2.6 Assumptions and Dependencies**

Future versions of this project should use primarily java as the medium of programming.

**3. Specific requirements**

**3.1 External Interface Requirements**

**3.1.1 Hardware interfaces**

The hardware specifications on which the project has been developed:

Processor: Intel i5 7200U (Minimum - Pentium processor and above)

X 86 compatible processor

RAM: 8 GB (Minimum - 512 MB and above)

HDD: 1 TB (Minimum - 80 GB or above)

**3.1.2 Software interfaces**

The software specifications on which the project has been developed:

Operating System: Windows 10

Front End: Java (Eclipse Java Oxygen)

Back End: Java (Eclipse Java Oxygen)

**3.1.3 Communication interfaces**

Section 2.1.4 of this document already provides all details about communication interfaces.

**4. Other Nonfunctional Requirements**

**4.1 Performance Requirements**

Most public key ciphers rely on high computational cost operations. Therefore, keeping performance considerations in mind, we have used symmetric key encryption for data encryption/decryption. Also, Huffman’s coding algorithm is faster in comparison to other compression algorithms and more efficient too. Hence, we have used the mentioned algorithm.

**4.2 Safety and Security Requirements**

User is required to remember his password that he/she used to encrypt data (or lock password safe) because most of secure cryptographic algorithms implemented in this suite are secure enough so that no algorithms better than brute-force can be used to recover lost password.

**4.3 Software Quality Attributes**

**• Portability:** The code is platform independent; it should be easily portable to different architectures and operating systems.

**• Reliability:** Serious attempts are made to make sure code is reliable and of enterprise quality.

**• Usability:** The project is still in planning stage.

**5. Conclusion**

This project on “Small C like compiler” is a user friendly application. The developed program will be helpful in converting small programs to machine code.

**Chapter-3**

**ARCHITECTURE DIAGRAM**

****

**Chapter-4**

**PROJECT METHODOLOGY**

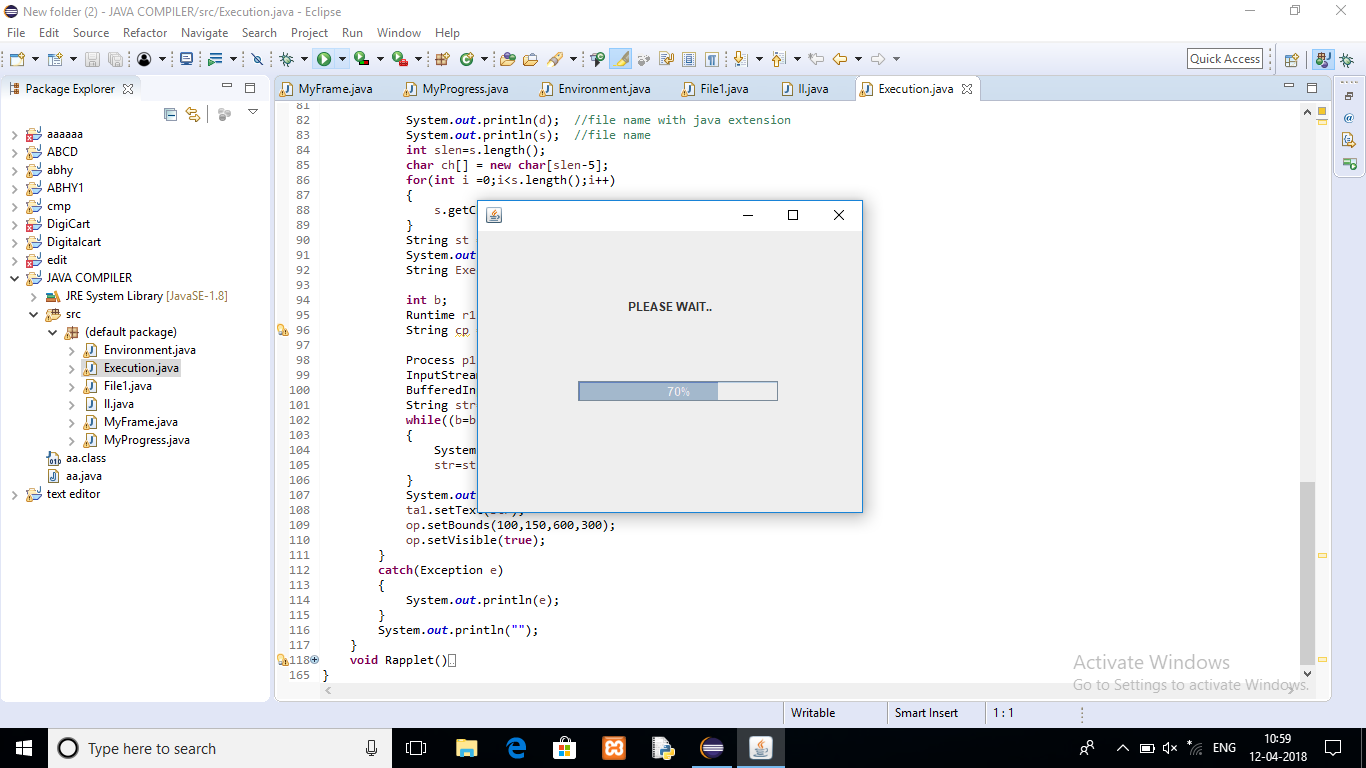
To avoid the consequences of the situation created by the historical evolution of the conventional compilers we need to develop a new methodology for language design and implementation that while accommodating the current programming tools would allow programmers to manufacture their own languages and compilers adapted to their own machines and problem domains. Such a methodology becomes feasible if it is based on mathematical concepts of programming language and programming language translation that are independent of the computer and computer user and could be easily mastered by the programmers. Since there is already a rather long history of using the universal algebra as a framework for language specification   and some successful experiments on compiler modeling by algorithms for homomorphism computation   we are seeking the new methodology for language processing in the framework of universal algebras.

The difficulty in using the algebraic methodology for the development of a new technology for language processing resides in the way programming languages evolved as ``notations with which people can communicate algorithms to computers and to one another" and the manner in which algebraic mechanisms have been used to specify such a notation. In other words, while the concept of a programming language evolved as a notation of well understood but unspecified computations, the compiler design is based on formal specifications of both the computations expressed by a programming language and the notation used to express these computations. The notation part of the programming language, henceforth called the syntax, was precisely described within the framework of formal languages. The computations expressed by the syntax, henceforth called the semantics, have been formalized using the domain theory   which is a different framework. These formalizations of syntax and semantics of a programming languages allowed us to understand the compilation process and to develop current technology for programming language processing but did not mature into a formal concept of a programming language and into a mathematical model of a compiler. The cause of this state of the art might be found in the lack of a mathematical framework naturally integrating the syntax, the semantics, and the compilation process.

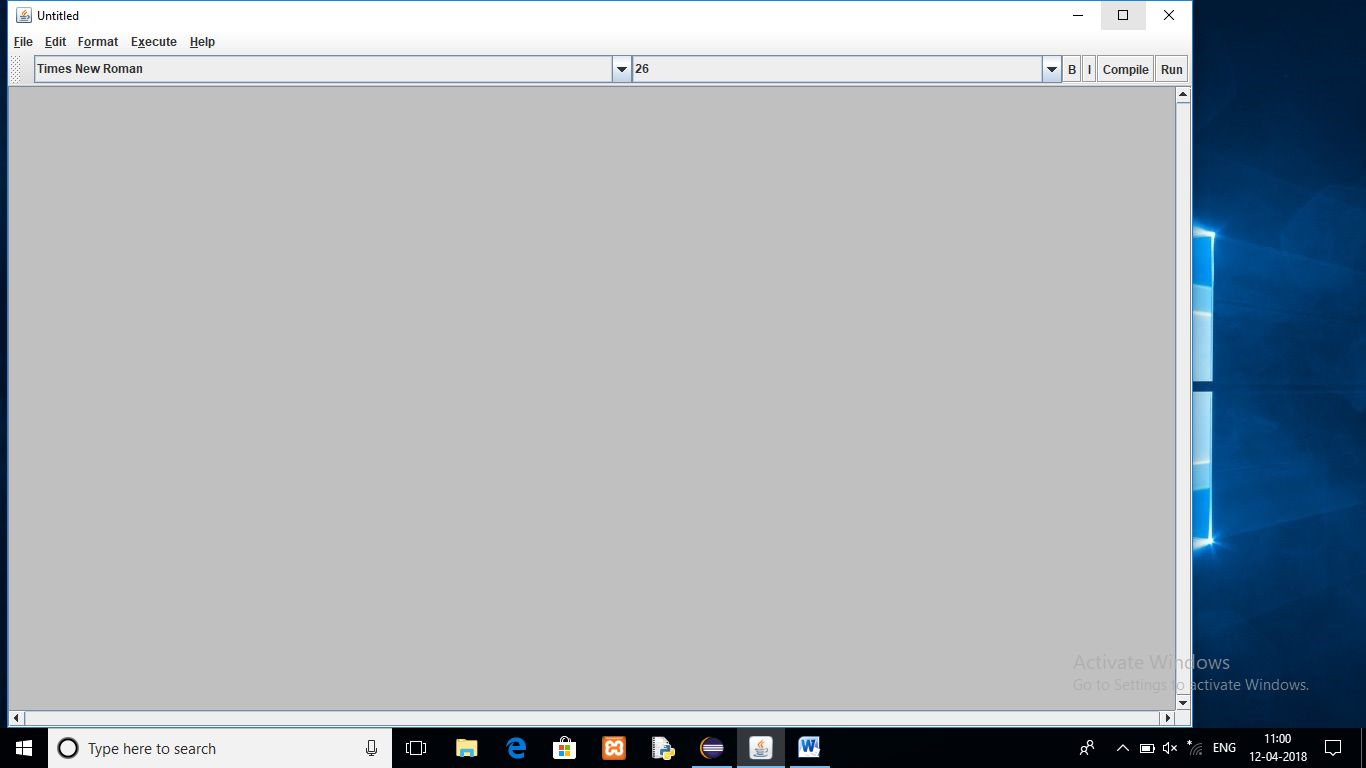
**Chapter-5**

**SCREENSHOTS**

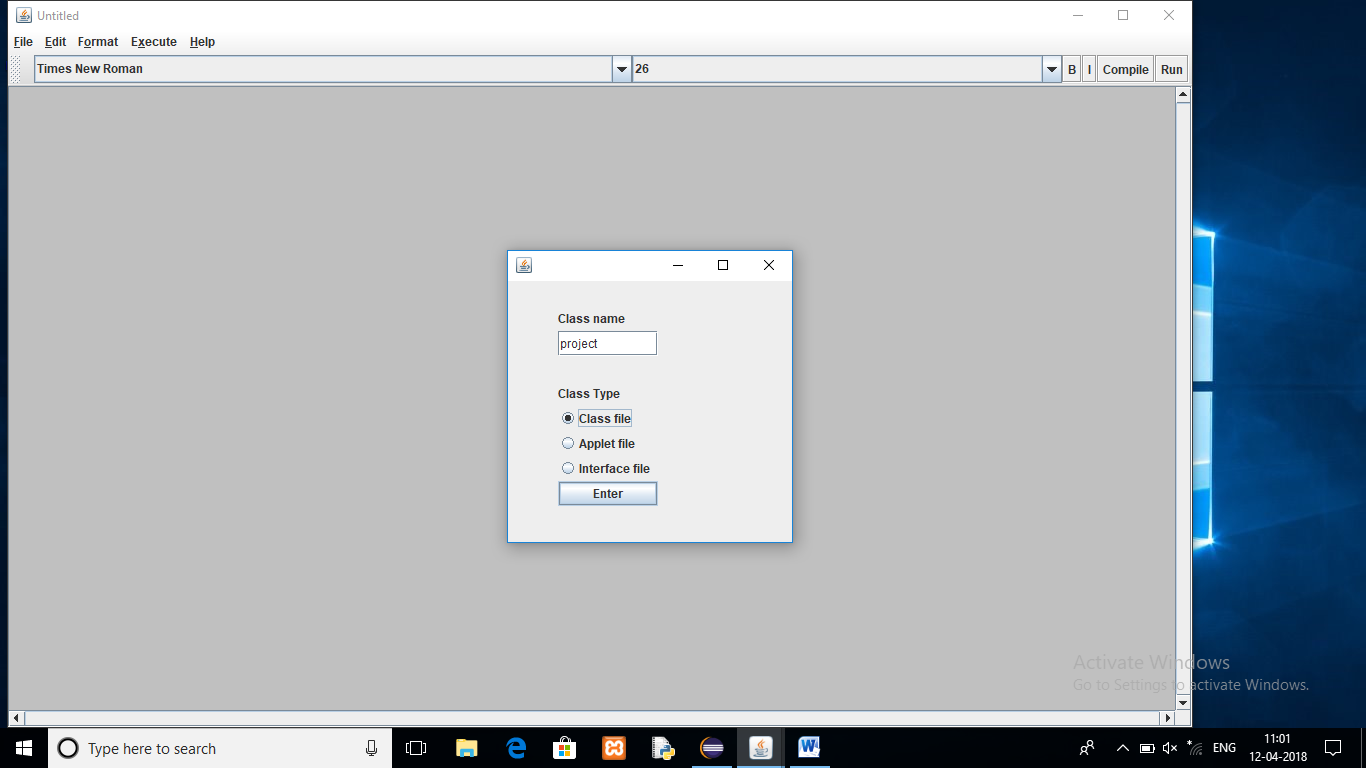
**1 .**When the code starts executing



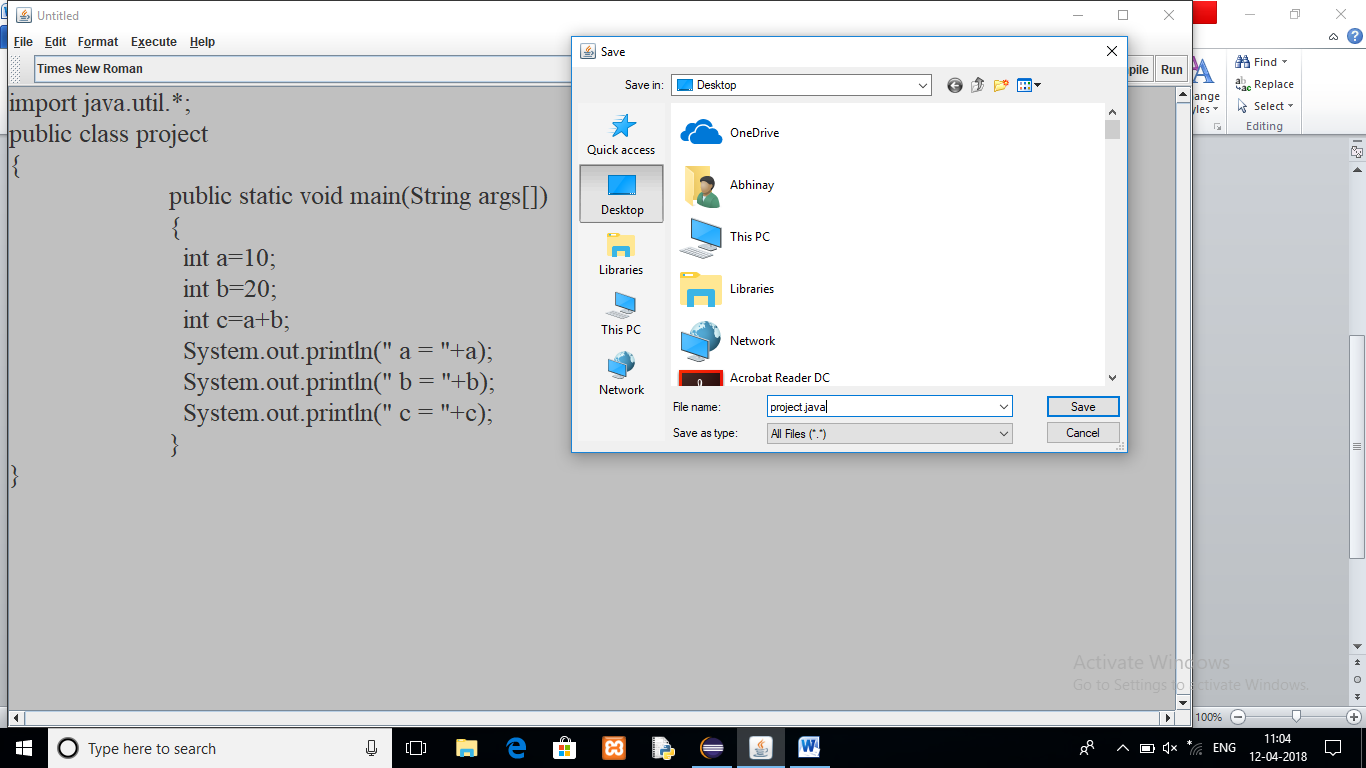
2. First Window that appears after execution



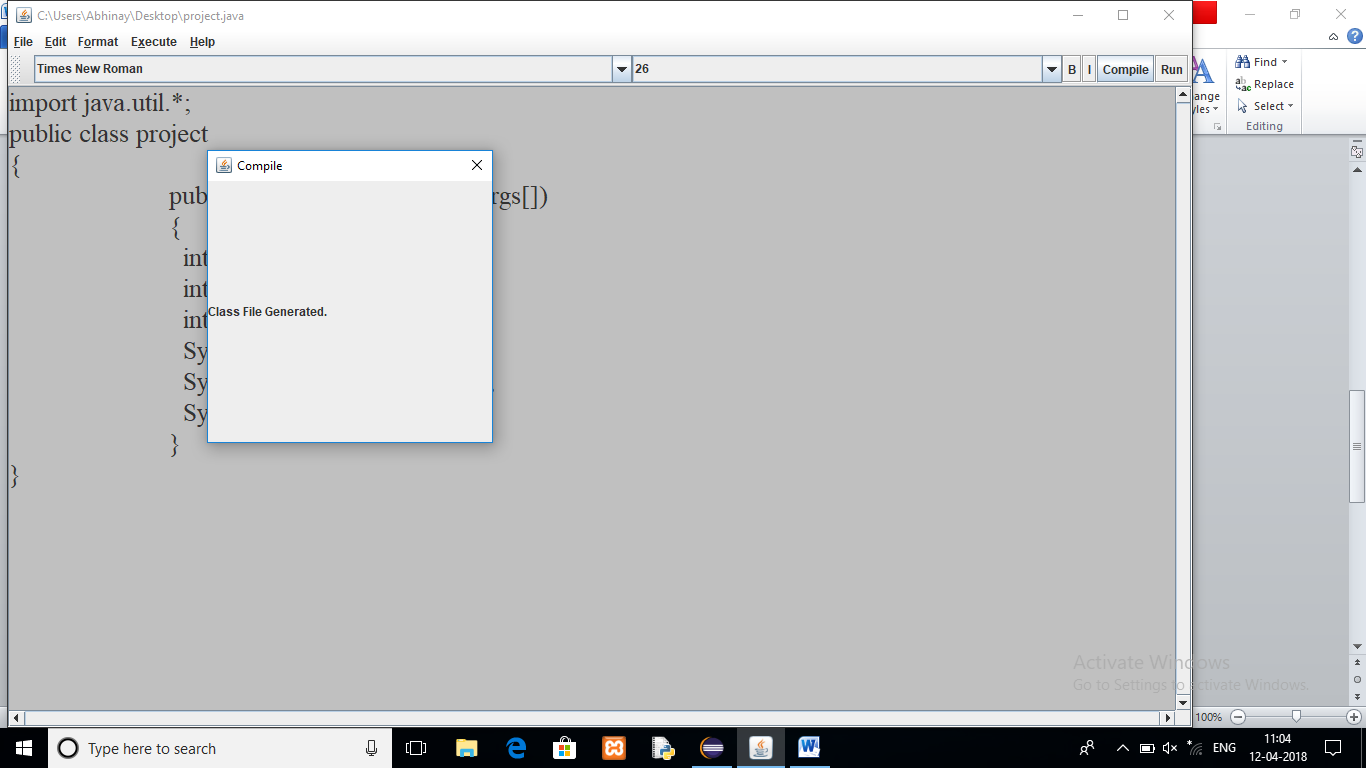
3. Creating the class(while writing the code)



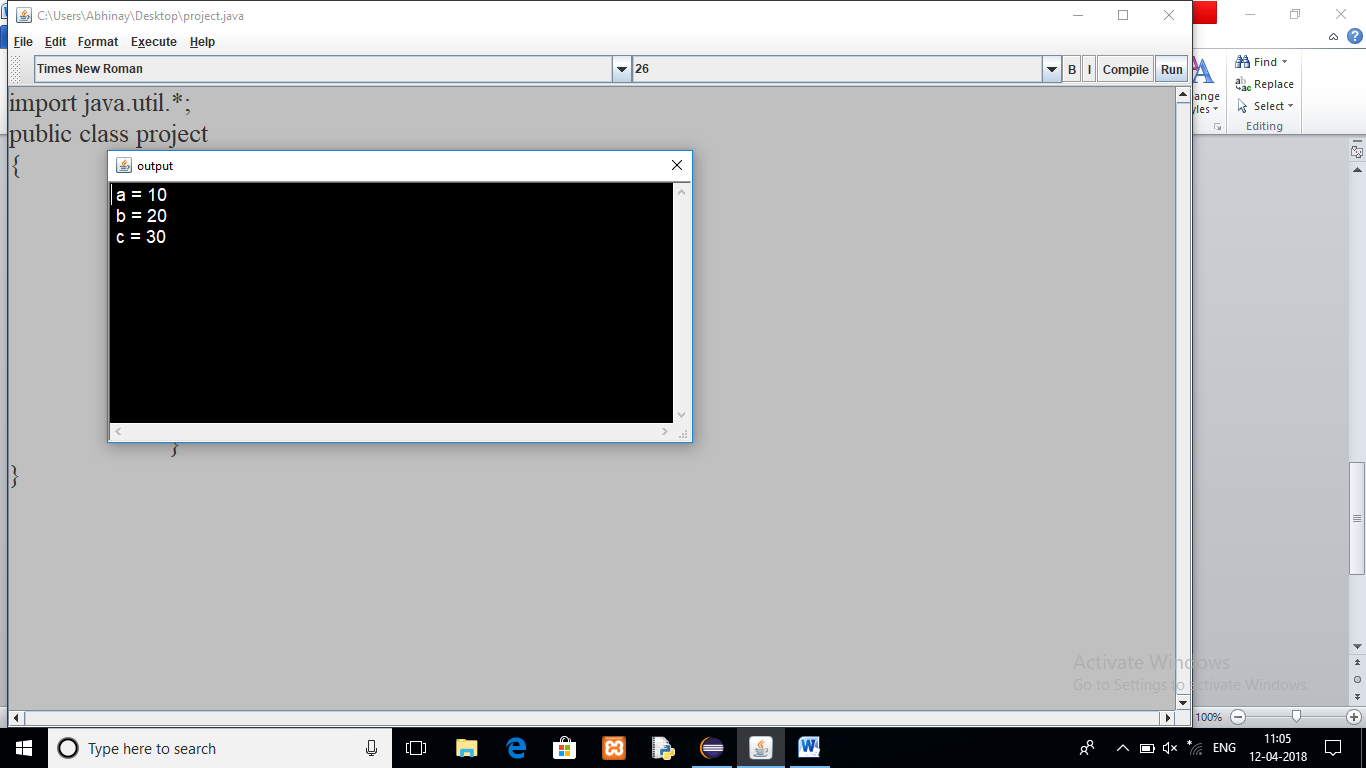
4. Saving the code



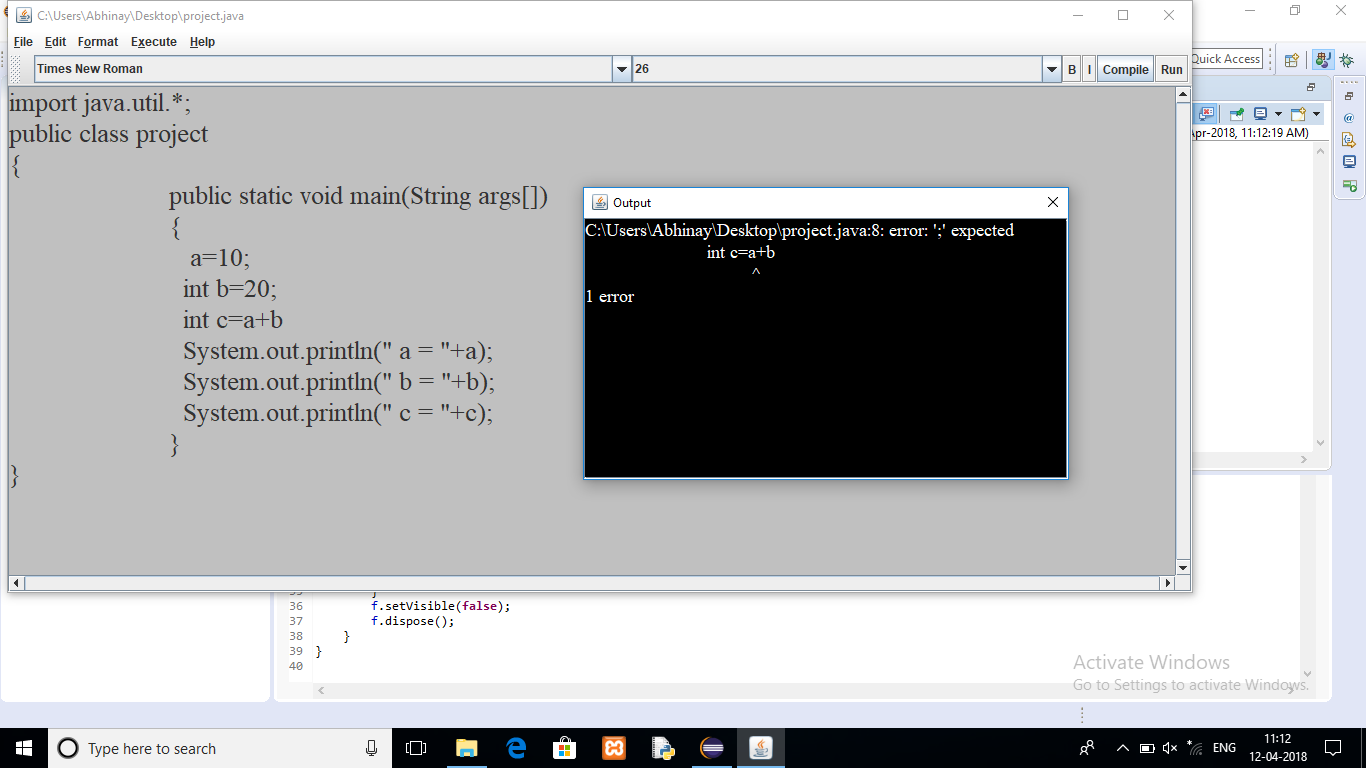
5. On successful compilation of the code



6. Output of the Program



7. When there is Error in the Code



**Chapter-6**

**CONCLUSION AND FUTURE SCOPE**

**Conclusion:** With the advent of multi-core systems, the era of multi-core programming has arrived which allows the users to achieve parallelism with the support of multiple processors. Compilation is also an area of interest which still needs sufficient work to achieve parallelization in the field. The previous efforts describe only theoretical work. The practical implementation of those have not been seen yet. The work presented in the thesis mainly focuses on parallelizing compilation process by making efficient use of multi-core systems. We started our work by studying various phases of a compiler and multi-core programming. In this thesis we have presented some techniques to parallelize the existing sequential code by identifying pivot locations in the programs. Three techniques have been discussed to identify pivot locations for division of code. These techniques use construct, white space characters and new line characters. Based on aforementioned techniques parallel lexical analysers are generated which perform parallel lexical analysis of a program. With the experimental results it is clear that construct based code division technique performs better than new-line and white-space block splitting methods. For further improvement, the same technique is extended to its optimal scheduling version in which speedup of almost 4.14 is achieved. Since work is based on detection of pivot locations, which requires extra time for preprocessing, a smart editor is developed for the purpose. We also attempted parallel lexical analysis using OpenMP. The speedup increased upto 6.8 but this technique works only with memory blocks, it does not work with files. An approach for parallel lexical analysis of multiple files which can use multi-core machines is presented. It is clear from experiments that substantial amount of time can be saved in lexical analysis phase by distributing files across number of CPUs. With the increase in number of processors the overall time in compilation would definitely be far too less as compared to all-serial approach. Parsing is an important phase of a compiler for parallel implementation. Various issues in 89 implementation of parallel parsing algorithms on multi-core machines are discussed. Parallel syntax analysis of multiple C source code files was presented. It is assumed that lexical analysis of individual files that were scheduled on a processor for syntax analysis is done on the same processor. The speed up obtained for 7 CPUs is 6.31 which is quite reasonable. The speedup would further increase with more number of processors. Though this increment in speedup would be decreasing as the time devoted to distribute files would increase. An automatic C source code generator which generates the syntactically correct code in C language, which is helpful for the performance evaluation and analysis of parallel programs is developed. Since bench mark programs are not commonly available for researchers to test the performance and analysis of algorithms for lexical and syntax analysis, this tool is quite helpful for such research and academic purposes. The current version of the tool generates only looping and decision making constructs with only assignment and print statements. It can be enhanced to generate functions and pointers and other programming constructs/features. From the study and experimentation it is clear that by using multi-core machines the compilation process can also be parallelized which improves overall compilation time. The research can also be extended to other phases of a compiler. By applying the techniques mentioned in the work undertaken large software like GCC, OS Kernel etc. can be compiled and installed in much less time as compared to their sequential counterparts.

**Future Scope:**

Common Symbol Table: In the present work lexical and syntax analyzers operate on individual symbol tables, i.e. each thread executing lexical analyzer has its own symbol table. On all such tables, corresponding syntax analyzers operate. Though it suffices our purpose but subsequent phase i.e. code generation would require a single symbol table. The work can be taken up to consider the sharing of single symbol table by all threads.

Parallel Code Generation: One should attempt to explore generation of code in parallel. This is definitely going to help a lot when it comes to compiling huge software like OS Kernel and GCC etc.

Parallel Semantic Analysis: This was also not explored. This would involve challenges of type checking, type casting, abstract syntax tree traversal and possible re-writing in parallel. This would also enhance the overall compilation.

Parallel Error Recovery: Like other phases, error recovery should also be thought of to be done in parallel. This would substantially reduce debug time for huge software.

**REFRENCES**

* Java 2: The Complete Reference
* Wikipedia
* Tutorialspoint
* GeeksforGeeks
* Nptel
* Compiler Design: Analysis and Transformation