DAYANANDA SAGAR UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING
DAYANANDA SAGAR UNIVERSITY
KUDLU GATE
BANGALORE - 560068

MINI PROJECT REPORT

ON

"DESIGNING A COMPILER FOR A SIMPLE PROGRAMING LANGUAGE"

SUBMITTED TO THE VIth SEMESTER COMPILER DESIGN SYSTEM SOFTWARE LABORATORY-2019

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE & ENGINEERING

Submitted by

K DEVA BHARGAV-(ENG17CS0107)
KIRAN M-(ENG17CS0111)
MANASA S-(ENG17CS0119)
MANOJ KUMAR D-(ENG17CS0122)

Under the supervision of

Prof. Nazmin Begum ,Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING
DAYANANDA SAGAR UNIVERSITY
KUDLU GATE, BANGALORE – 560068

DAYANANDA SAGAR UNIVERSITY

School of Engineering, Kudlu Gate, Bangalore-560068

CERTIFICATE

This is to certify that Mr./Ms. KIRAN M *bearing USN* ENG17CS0111 *has satisfactorily completed his/her Mini Project as prescribed by the University for the* 5TH *semester B.Tech. programme in Computer Science & Engineering during the year* 2019-2020 *at the School of Engineering, Dayananda Sagar University., Bangalore.*

Date: _____

_____

Signature of the faculty in-charge

Max Marks	Marks Obtained

_____

Signature of Chairman

Department of Computer Science & Engineering

DECLARATION

We hereby declare that the work presented in this mini project entiltled- "**Designing a compiler for a simple programming language**", has been carried out by us and it has not been submitted for the award of any degree,diploma or the mini project of any other college or university.

K DEVA BHARGAV-(ENG17CS0107)
KIRAN M-(ENG17CS0111)
MANASA S-(ENG17CS0119)
MANOJ KUMAR D-(ENG17CS0122)

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We are especially thankful to our **Chairman ,Dr. M K Banga**,for providing necessary departmental facilities, moral support and encouragement.

We are very much thankful to **Prof Nazmin Begum**, for providing help and suggestions in completion of this mini project successfully.

We have received a great deal of guidance and co-operation from our friends and we wish to thank all that have directly or indirectly helped us in the successful completion of this project work.

K DEVA BHARGAV-(ENG17CS0107)
KIRAN M-(ENG17CS0111)
MANASA S-(ENG17CS0119)
MANOJ KUMAR D-(ENG17CS0122)

TABLE OF CONTENTS

TABLE OF CONTENTS (Figures)

Abstract

The overall purpose of the project is to design a compiler that converts the high-level language that is very similar to the python 3 to its assembly equivalent code for further compilation. The compiler is a program that converts instruction into a machine-code or lower-level form so that they can be read and executed by a computer. We are designing a program that will analyse the program in the input file and generate a introspector (Tree Structure of the code) and the output file that has the code structure and object code. The compiling process is described as a state transformation with a continuation based method. Therefore, this compiler specification technique is very similar to the denotational semantic technique used for specifying the semantics of programming languages. This work is a little different from our previous work where the semantics of source language and the semantics of target language applying to the same abstract states. In this work, the semantics of source language and the semantics of target language are described in terms of different abstract states. We have also investigated the compilation techniques for a programming language with procedures and stack-based run-time environment. The trends found are that the generated output file that has the assembly code is distinguished line by line separated by comments that specify the exact line number below which the assembly code of the line is written.

I. Introduction

The compiler has a set of rules that acts as the syntax while we write a program in our own language, these rules act as the base for the compiler design

1.1 Why our topic is important?

The compiler is a the most important part of the compiler (First phase of the compiler), its function is to convert the program written by the programmer to the assembly code that is understood by the machine.

1.2 Where is it used for? Applications

Our topic is a very basic and essential phase of any compiler design so it is used in almost all the compiler designs. Our topic is very similar to the python 3 compiler, other compilers for languages like C, C++, JAVA also use the very similar process to convert the programs written in the respective languages to the equivalent assembly language code.

1.3 What will we talk about?

We will be talking about the design of the compiler in detail. We are talking about

- Parser
- Identification
- Type Checking
- Offset
- Code Generation

Rules for the project

1)A program is a sequence of variable and function definitions.

2)The syntax of a variable definition is a non-empty enumeration of comma-separated identifiers followed by a ":" and a type. Variable definitions must end with the ";" character.

3)Functions are defined using the def keyword, the function identifier and a list of coma separated parameters between (and) followed by ":" and the return type or the "void" keyword. The return type and parameter types must be built-in (i.e., no arrays or records). The function body goes between { and }.

4)The bodies of functions are sequences of variable definitions followed by sequences of statements.

5) Both must end with the ";" character.

6)The last and mandatory function is "main", which returns void and receives no parameters. Built-in types are "int", "double" and "char".

7)Array types can be created with the "[]" type constructor, specifying the size of the array with an integer constant (like C) followed by any type.

8)The "struct" type constructor is added for specifying record types. Records have no type identifier, and fields are declared as var definitions between { and }.

9)Type definition (i.e., typedef) is not supported.

10) The syntax of a write statement is the "print" keyword followed by a non-empty comma separated list of expressions. The read statement is similar, but using the "input" keyword.

11) Assignments are built with two expressions separated by the "=" operator.

12) If / else and while statements follow the Python syntax. The statement body goes between { and }.

13) The return <expression> statement is also supported (the expression is mandatory).

14) A function invocation may be an expression or a statement. A procedure invocation is always a statement.

15) The cast expression follows the C syntax.

16) Expressions are built with:

- Integer, real and character constants without sign.
- Identifiers.
- The following operators applied to one or two expressions (descending order of precedence):

( )	Non associative
[ ]	Non associative
.	Left associative
CAST	Non associative
-(unary)	Non associative
!	Non associative
***/%**	Left associative
+-	Left associative
++--	Left associative
+=-=*=/=	Left associative
>>=<<=!===	Left associative
&&||	Left associative
=	Right associative

II. Problem Statement

To build a compiler in java that compiles the written code and produces Introspector (Tree Structure of the code) and the Output file that has the code structure and object code. The compiler as to show the programmer few possible errors like type error, identification error and etc. We use a classic approach towards the compiler design and include all the basic parts or phases of a compiler, these phases are similar to all most all the languages.

III. Objective

The major object of our project is to generate the assembly code for the given program in high-level language. To achieve this we have attain perfection of code generation in several different phases that are:

1. **Lexical Analyzer** –
   It is also called scanner. It takes the output of preprocessor (which performs file inclusion and macro expansion) as the input which is in pure high level language. It reads the characters from source program and groups them into lexemes (sequence of characters that "go together"). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (for e.g. erroneous characters), comments and white space.

2. **Syntax Analyzer –** It is sometimes called as parser. It constructs the parse tree. It takes all the tokens one by one and uses Context Free Grammar to construct the parse tree.
   Why Grammar ?
   The rules of programming can be entirely represented in some few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.
   Syntax error can be detected at this level if the input is not in accordance with the grammar.

3. **Semantic Analyzer –** It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree.It also does type checking, Label checking and Flow control checking.

4. **Intermediate Code Generator –** It generates intermediate code, that is a form which can be readily executed by machine We have many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.
   Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

5. **Code Optimizer –** It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered.

Optimization can be categorized into two types: machine dependent and machine independent.

6. **Target Code Generator –** The main purpose of Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection etc. The output is dependent on the type of assembler. This is the final stage of compilation.

IV. Methodology

4.1 Block Diagram

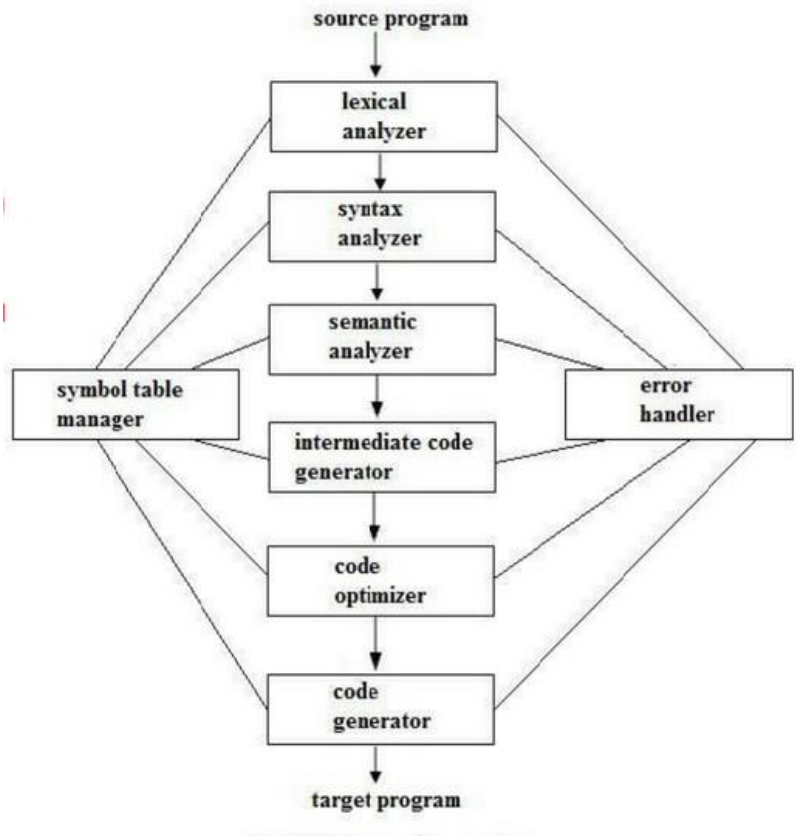

Fig. 4.1.1 : The block diagram of a compiler

4.2 Proposed Work

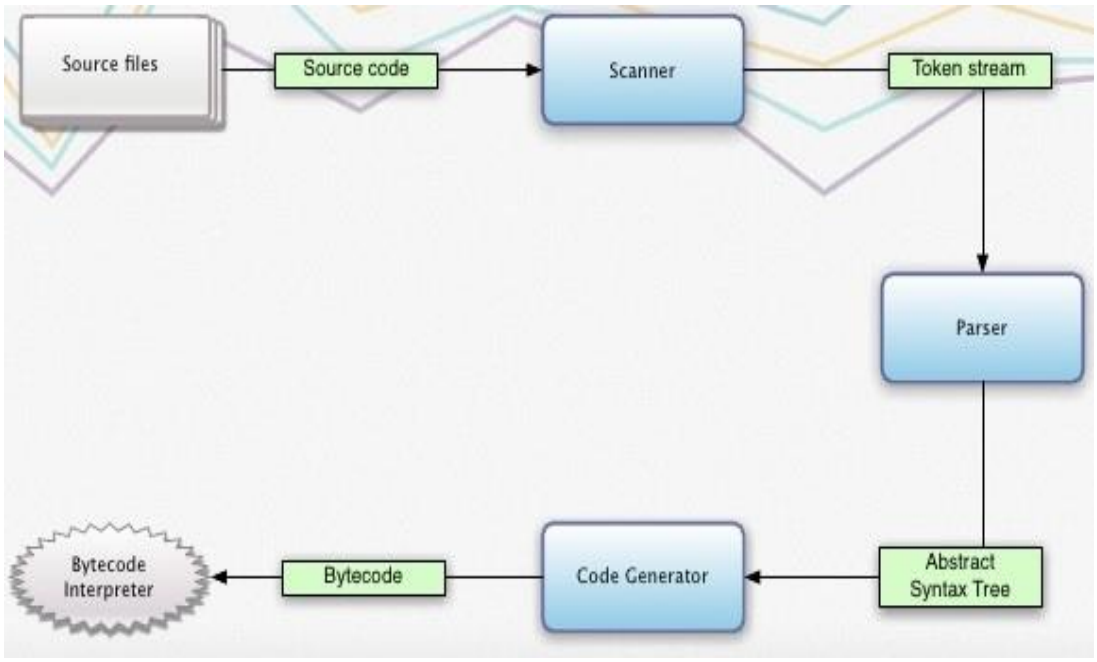

Fig. 4.2.1: The proposed working of the project

V. Software Requirement

5.1 Front end:

- Eclipse IDE for JAVA developer version -2020-03
- JAVA JDK Introspector

5.2 Back end:

- JAVA JDK-13
- Windows 10 OS

VI. Results

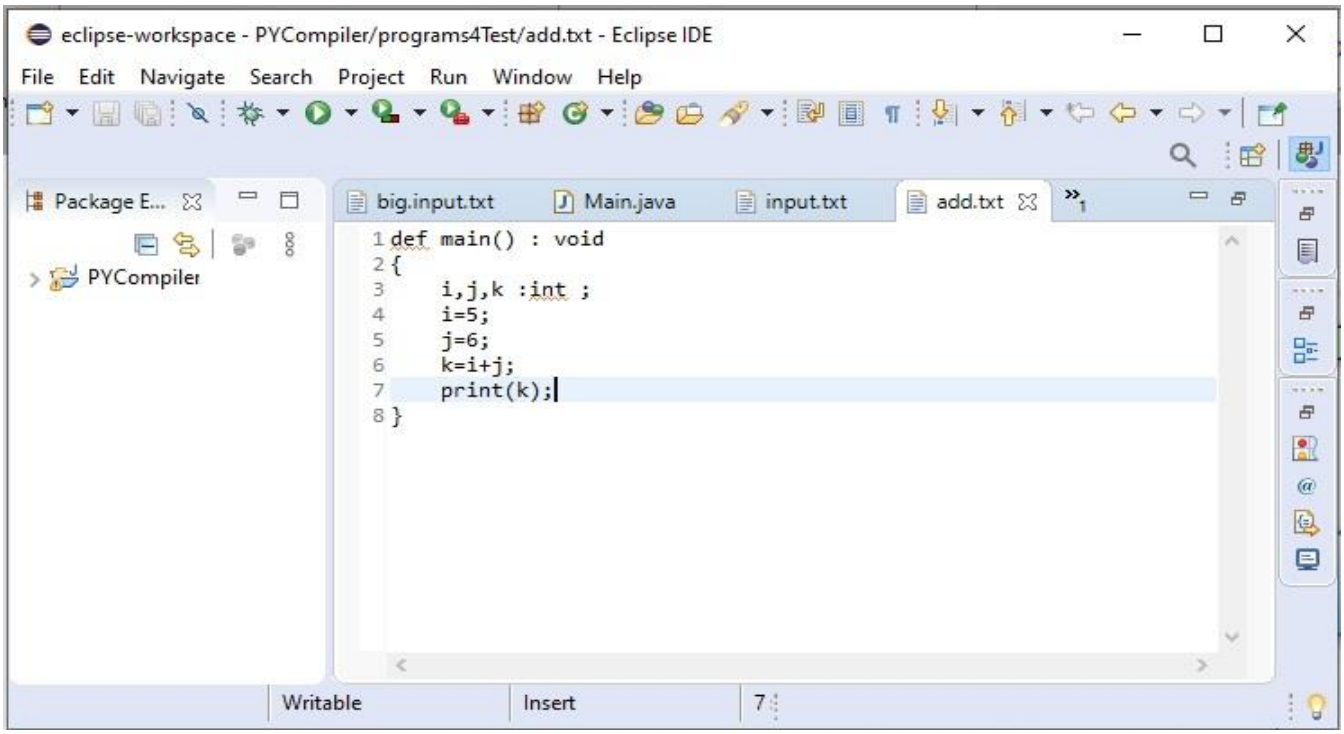

Fig 6.1.1 : The sample program to add two numbers

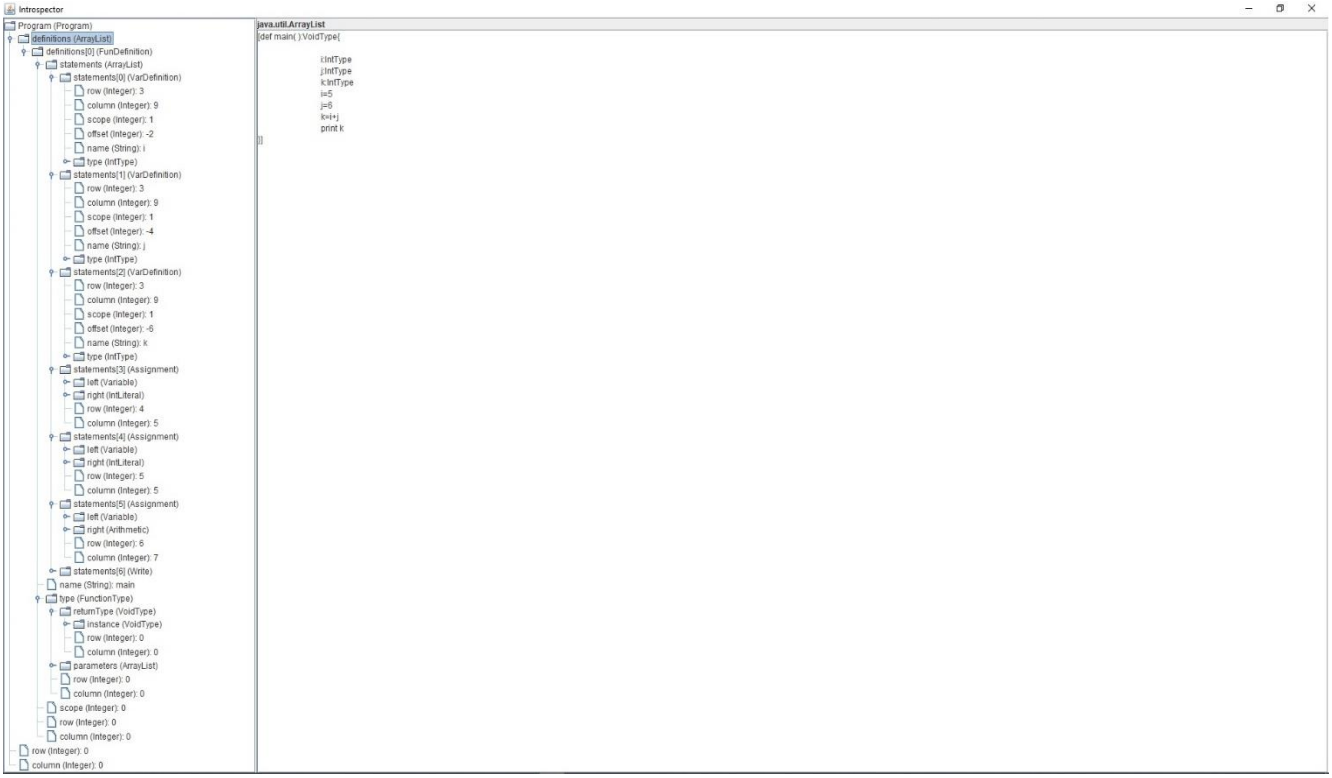

Fig 6.1.2 : The sample tree structure for adding of two numbers

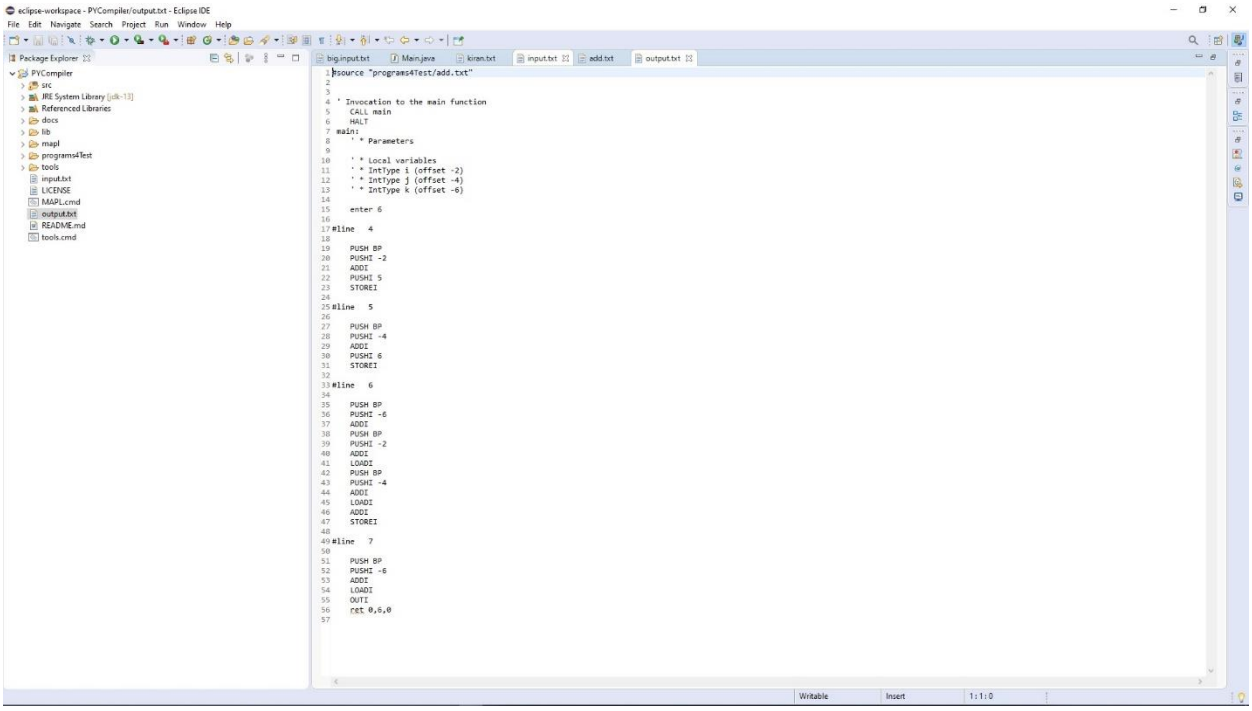

Fig 6.1.3 : The sample output text file that has the final output of the addition of two numbers code in assembly language

VII. Conclusion

We have accomplished building of a compiler that is fully capable of converting the High-level language to assembly equivalent code. We use Python3 as a base to design the language, we could have used languages like JAVA or C++ but we use Python3 as it supports more features then any other language, so our rebuilt version can also support almost all the features similarly to Python3. For future we have many features that only Python3 supports we can include those features to our project, few of them are

- Advanced unpacking
- Keyword only arguments
- Chained exceptions
- Fine grained OS-Error subclasses
- No more comparison of everything to everything

VIII. References

[1]Marmot: an optimizing compiler for Java by:- Robert Fitzgerald Todd B. Knoblock.
[2]Design and evaluation of dynamic optimizations for a Java just-in-time compiler .
by:-Toshio Suganuma,Toshiaki Yasue,Motohiro Kawahito,Hideaki Komatsu,Toshio Nakatani
https://dl.acm.org/doi/abs/10.1145/1075382.1075386