 (ISO 9001:2015 Certified), Accredited with ‘A’ Grade by NAAC

08258 - 281039 – 281263, Fax: 08258 – 281265

**Department of Computer Science and Engineering**

B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018 to 30-6-2021

**REPORT ON**

COMPILER DESIGN MINI PROJECT

**Course Code: 18CS702**

Academic Year – 2021-2022

Course Name: **COMPILER DESIGN**

Semester: 7 Section: C

**Submitted To,**

**Course Instructor:**

***Ms. Asmita Poojary***

***Assistant Professor-II***

***Department of CSE,***

***NMAMIT, Nitte.***

Submitted By:

**1**. **USN**: 4NM18CS140 **NAME**: S DHRUVA

**2.** **USN**: 4NM18CS152 **NAME**: SAMIT D MANVAR

**Date of submission: 20-12-2021**

# 

# TABLE OF CONTENTS

Title Page 1

[Table of Contents 2](#_TOC_250008)

[Introduction](#_TOC_250007) 3

[Problem Statement](#_TOC_250006) 6

[Objectives](#_TOC_250005) 7

[Methodology](#_TOC_250003) 7

[Implementation](#_TOC_250002) 9

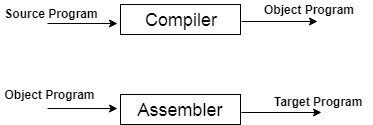
Results 18

[Conclusion and Future Scope](#_TOC_250001) 19

[References](#_TOC_250000) 19

**INTRODUCTION**

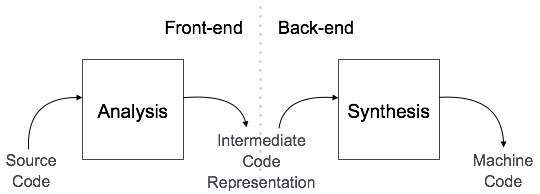
A compiler is a translator that converts the high-level language into the machine language. High-level language is written by a developer and machine language can be understood by the processor. Compiler is used to show errors to the programmer. The main purpose of compiler is to change the code written in one language without changing the meaning of the program. When you execute a program which is written in HLL programming language then it executes into two parts. In the first part, the source program compiled and translated into the object program (low level language). In the second part, object program translated into the target program through the assembler.



A compiler can broadly be divided into two phases based on the way they compile.

### **Analysis Phase**

Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



### **Synthesis Phase**

Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

**Compiler Phases**

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.



**Fig 1: Phases of Compiler**

### **Lexical Analysis:**

### Lexical analyser phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyser represents these lexemes in the form of tokens.

### **Syntax Analysis**

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

### **Semantic Analysis**

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyser keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

### **Intermediate Code Generation**

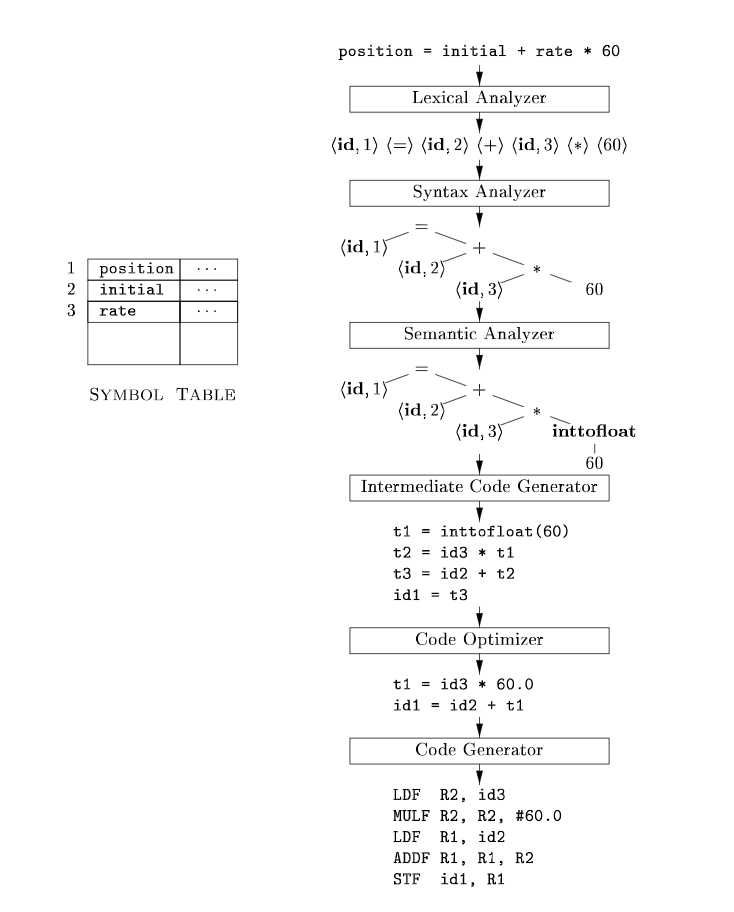
In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

### **Code Optimization**

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

### **Code Generation**

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.



**PROBLEM STATEMENT**

To design a compiler (Lexical and Parser Phase) for the given hypothetical language

Hypothetical language:

**int main ()**

**begin**

**int n, i, sum = 0;**

**for (i=1; i <= n; ++i)**

**begin**

**expr= expr+expr;**

**end**

**End**

**OBJECTIVES**

* To demonstrate the first phase of the compiler – lexical analysis for the given hypothetical language.
* To demonstrate the working of Parser phase – Syntax analysis.
* To demonstrate the working of CLR Parser and parse the given string.

**METHODOLOGY**

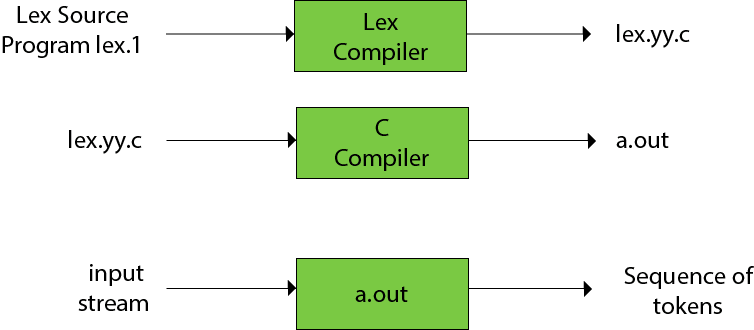
**Lexical Analysis:** Lexical analysis is the first phase of a compiler. It takes modified source code from language pre-processors that are written in the form of sentences. The lexical analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

## **Tokens**

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

In our project we can generate tokens for the given pseudocode with help of lex. A lex is a tool for automatically generating a lexer or scanner given a lex specification.



With help of lex program we are able to generate tokens. The rules defined to identify a valid token is described as pattern.

Lex program is divided into 3 sections

* Global C and Lex declarations section
* Lex rules section
* C code section.

These sections are delimited by %%.

... Definition section ...

%%

... Lex rules ...

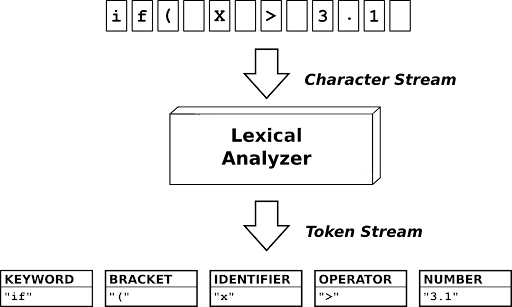
%%

... C subroutines ...

**Global C and Lex declarations section**: -This is the place to define macros, declare C variables and import header files written in C.

**Lex Rules**: - Lex rules are of the form **pattern {action}.** E.g., digit [0-9], letter [a-Za-z] etc.

**C subroutines**: -The C code section contains C statements and functions.



**Parser Phase**: - Parser phase is implemented using python program. Here we are going to implement CLR Parser for parsing the string.

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the greater number of states as compare to the SLR (1) parsing. LR (1) item is a collection of LR (0) items and a look ahead symbol. The look ahead is used to determine that where we place the final item. The look ahead always adds $ symbol for the argument production

**LR (1) item = LR (0) item + look ahead**

**IMPLEMENTATION**

**Implementation of Lexical Analyser:** - To generate tokens from given input (pseudocode) we use lex. We have defined patterns and corresponding actions associated with it. For example, when a pattern named keyword is matched corresponding action – printing the keyword takes place. In this way we generate tokens. For each token – keywords, identifiers, special symbols, constants, operators we have defined different lex rules (regular definition). Some important lex functions we need to remember.

* **yylex ()**: - The function that starts the analysis. It is automatically generated by Lex. (This is the entry point to LEX)
* **yywrap ():** - This function is called by LEX when end of file (or input) is encountered. If this function returns 1, the parsing stops.
* **yyin** of the type FILE\*. This points to the current file being parsed by the lexer. (Input file).

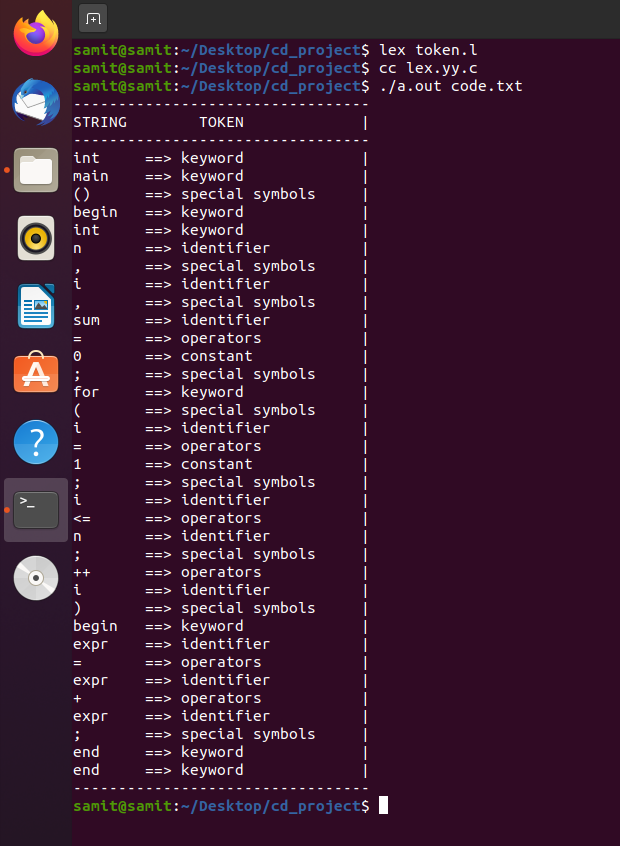


Fig 3: Generating tokens using lex

* First, we have declared pre-processor directives (header files) in global c declaration section.
* Next, we have defined the regular definition for corresponding tokens. For example, a keyword includes for, int, void, main, while, break etc. Similarly, it is defined for other tokens as well.
* We have defined pattern {action} for each token. When a string is identifier is found it prints the corresponding string as an identifier.
* We declare yywrap () and in the main () function we have c subroutines. Here we declare yyin to take input from file and also declare yylex () function. The input is read from a file name input.txt with help of yyin. We are opening the file in read mode

**Steps to run the lex program to obtain tokens**

1. Create a lex file with .l as extension. Here our lex file is token.l.
2. The input file which is the input to lex is input.txt. The input file contains the pseudocode for which we want to generate tokens
3. To run the lex program open terminal and type **lex token.l** and press Enter key.
4. Next, type **cc lex.yy.c** and press the Enter Key.
5. Finally type **./a.out input.txt** and press the enter key. The output is shown below.



**Fig 4 : Output of Lexical Analyser (Generated Tokens )**

**Implementation of Parser Phase: -** Parser phase is implemented by constructing a CLR Parser with help of python. CLR Algorithm is implemented using python 3.

For constructing a CLR Parser we need to follow certain steps

* For the given input string write a context free grammar
* Add Augment production in the given grammar
* Create Canonical collection of LR (0) items
* Construct a CLR (1) parsing table

Context Free Grammar: - Context free grammar G can be defined by four tuples as:

**G= (V, T, P, S)**

**G** describes the grammar

**T** describes a finite set of terminal symbols.

**V** describes a finite set of non-terminal symbols

**P** describes a set of production rules

**S** is the start symbol.

For our project we have defined production rules or context free grammar for the given pseudocode. The grammar is:

S->im()A

A->biv,v,vod;B

B->f(vod;vov;oov)C

C->beoeoe;yz

Where S, A, B, C are non-terminals.

Terminal symbols are i, m, b, v, o, d, f, b, e, y, z where

* i stands for int
* m stands for main
* b stands for begin
* f stands for for
* v stands for identifier
* o stands for operators
* e stands for expr
* y stands for end
* Z stands for End

Add augmented production in given grammar: - Before we start generating parse table, we need to add augmented production for example S’ -> S etc. For our pseudocode we have added S’ -> S .

S’ -> S //Augmented grammar

S -> im()A

A ->biv,v,vod;B

B ->f(vod;vov;oov)C

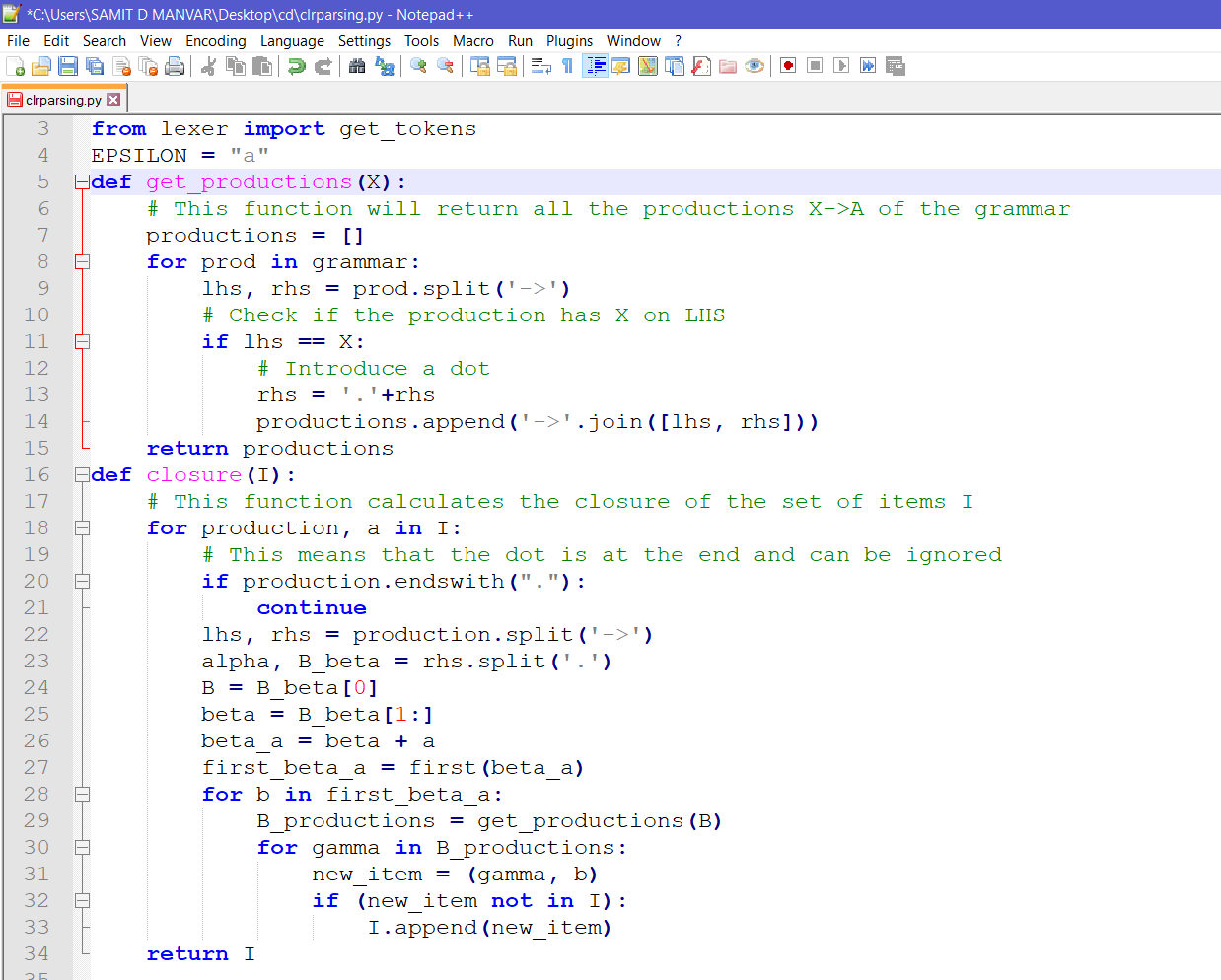
C ->beoeoe;yz

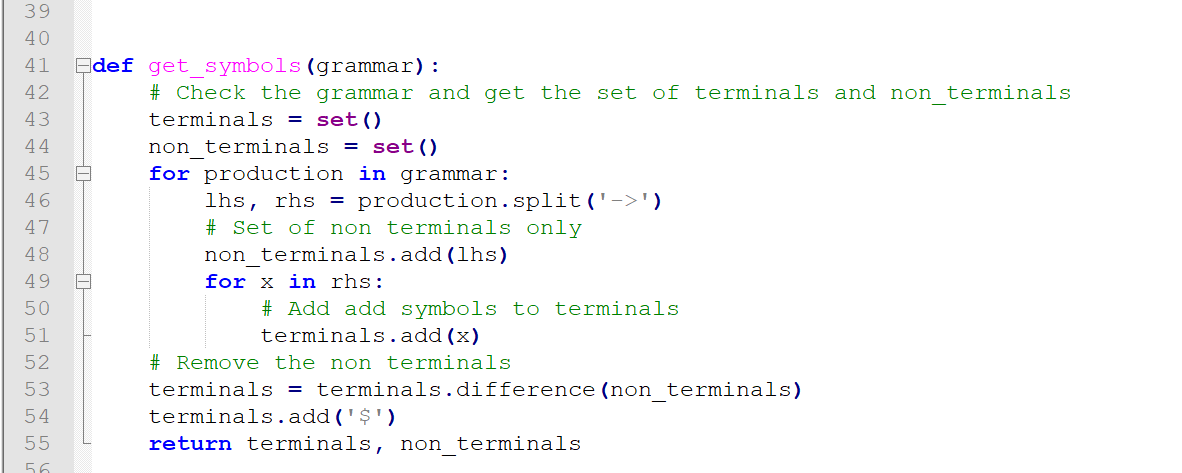
Then we create CLR Parser code with help of Python 3 and finally create parsing table and parse the string from CLR Parser code written in python.

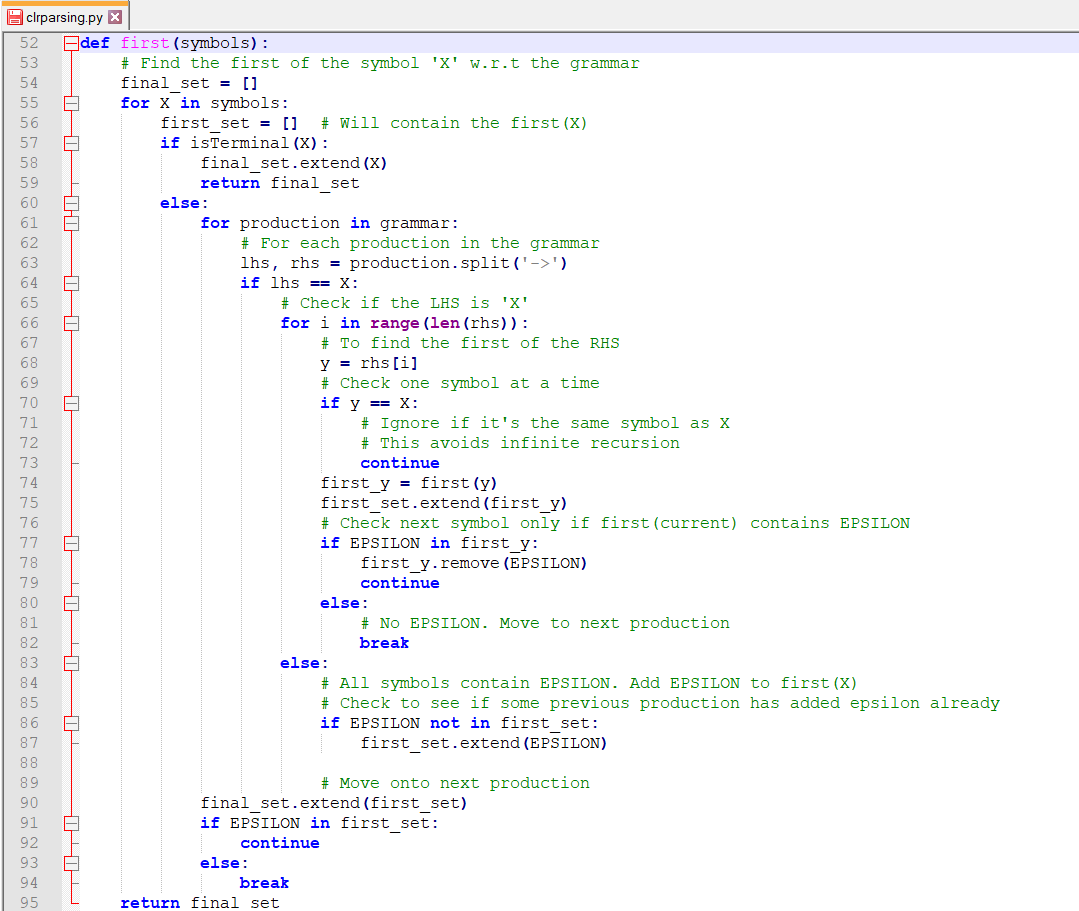
In CLR algorithm implemented, we have already defined the file name to be taken as input for production rules and for parsing string. The file grammarrules contains the production rules and the pseudocode contains the string/code to be parsed.

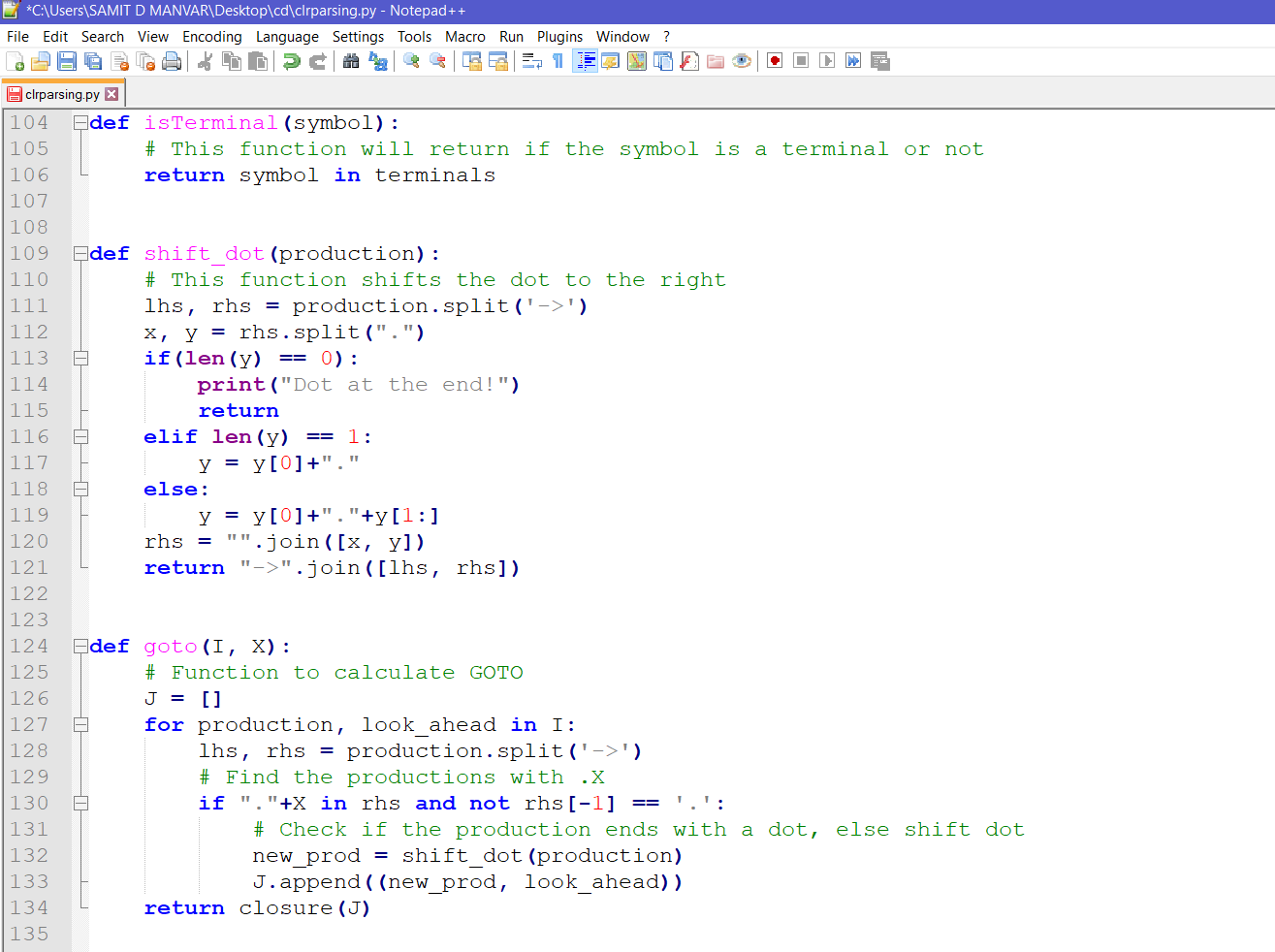
To run the clr parser: -In the command prompt type clrparsing.py. This file has the clr algorithm. Press Enter and the Parser table is generated. Also, the string is parsed and the result is displayed.

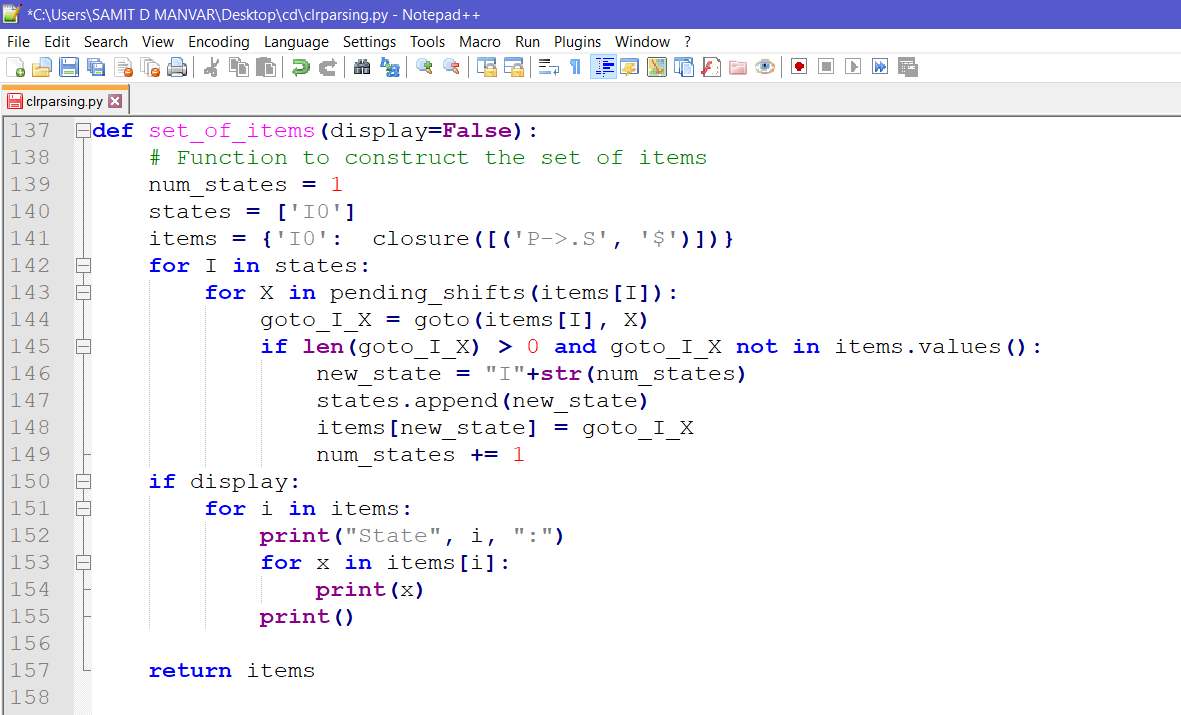
Code Snippets for implementing CLR Parser Algorithm

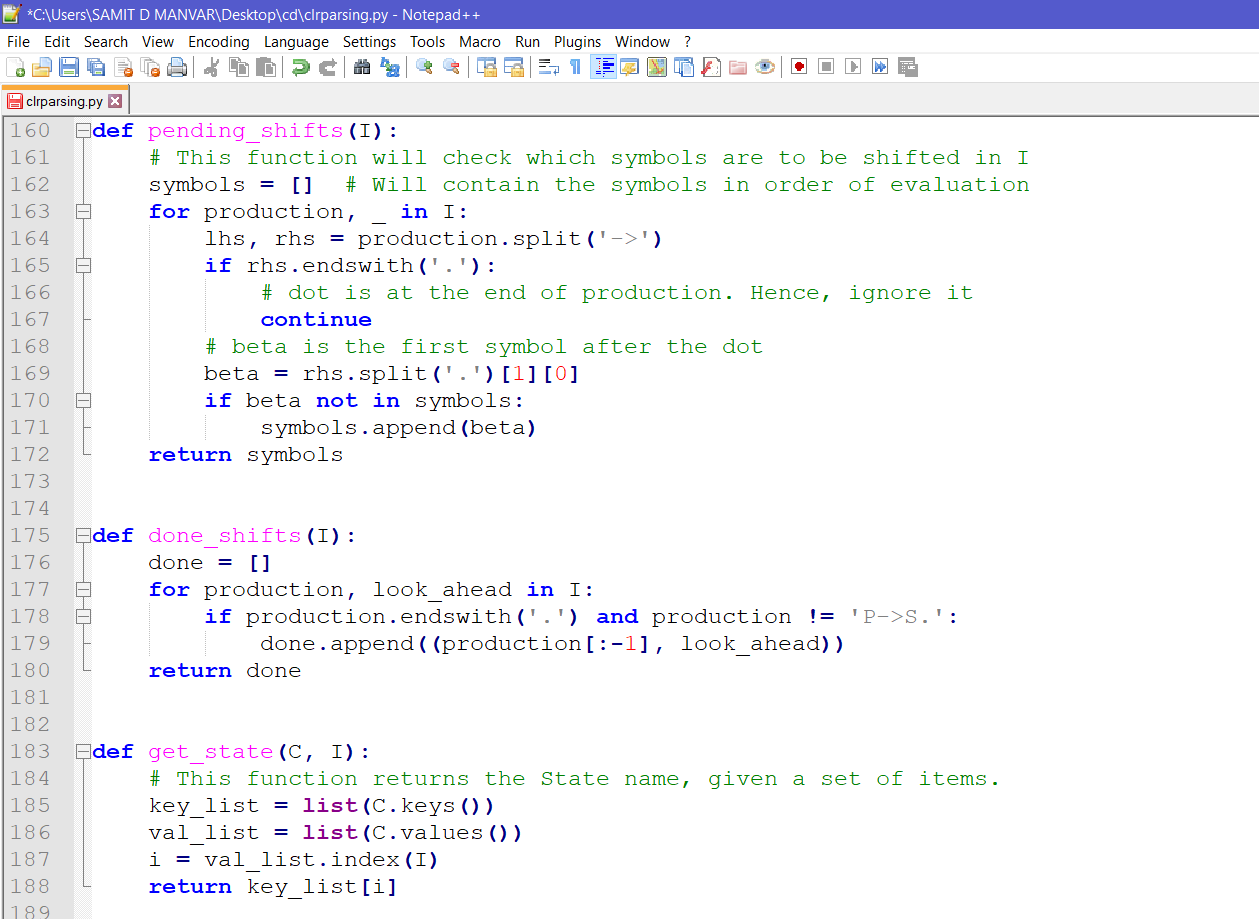


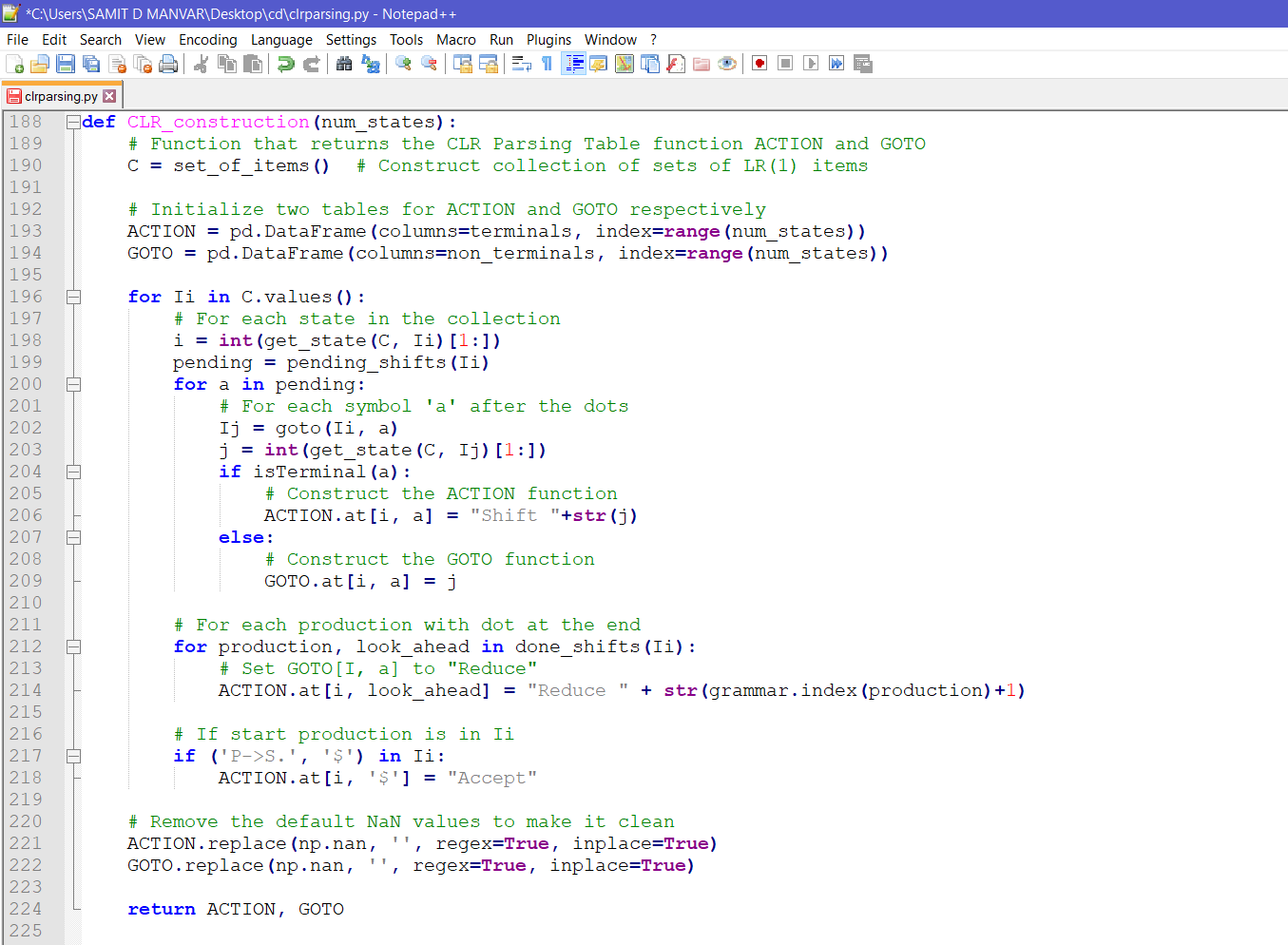


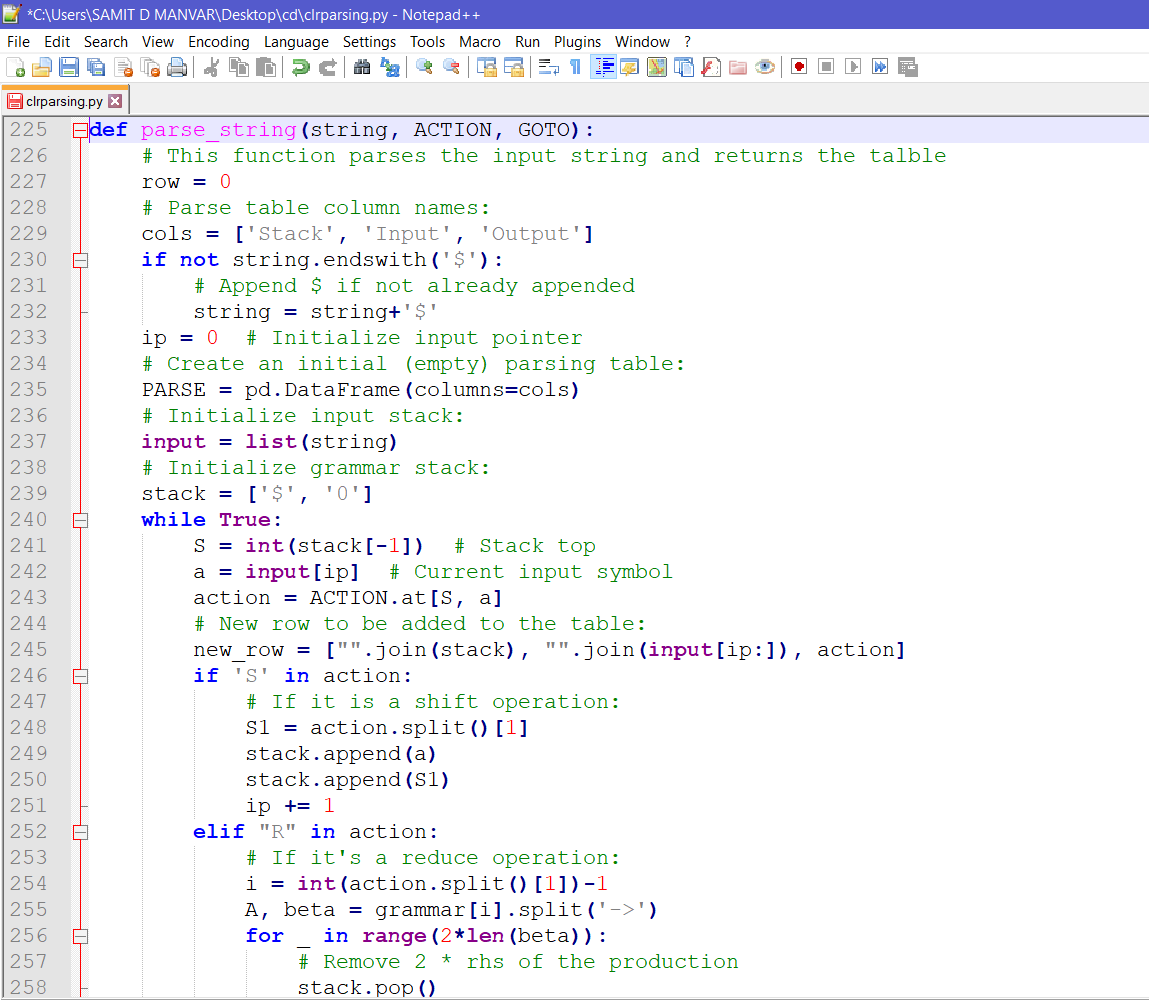


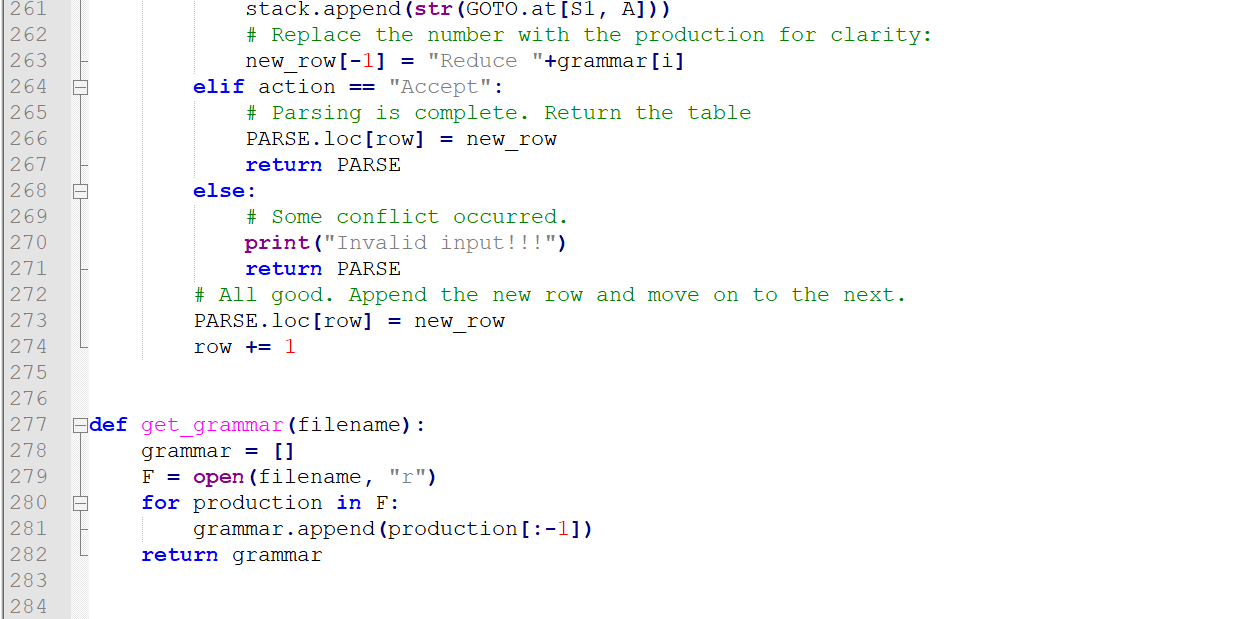


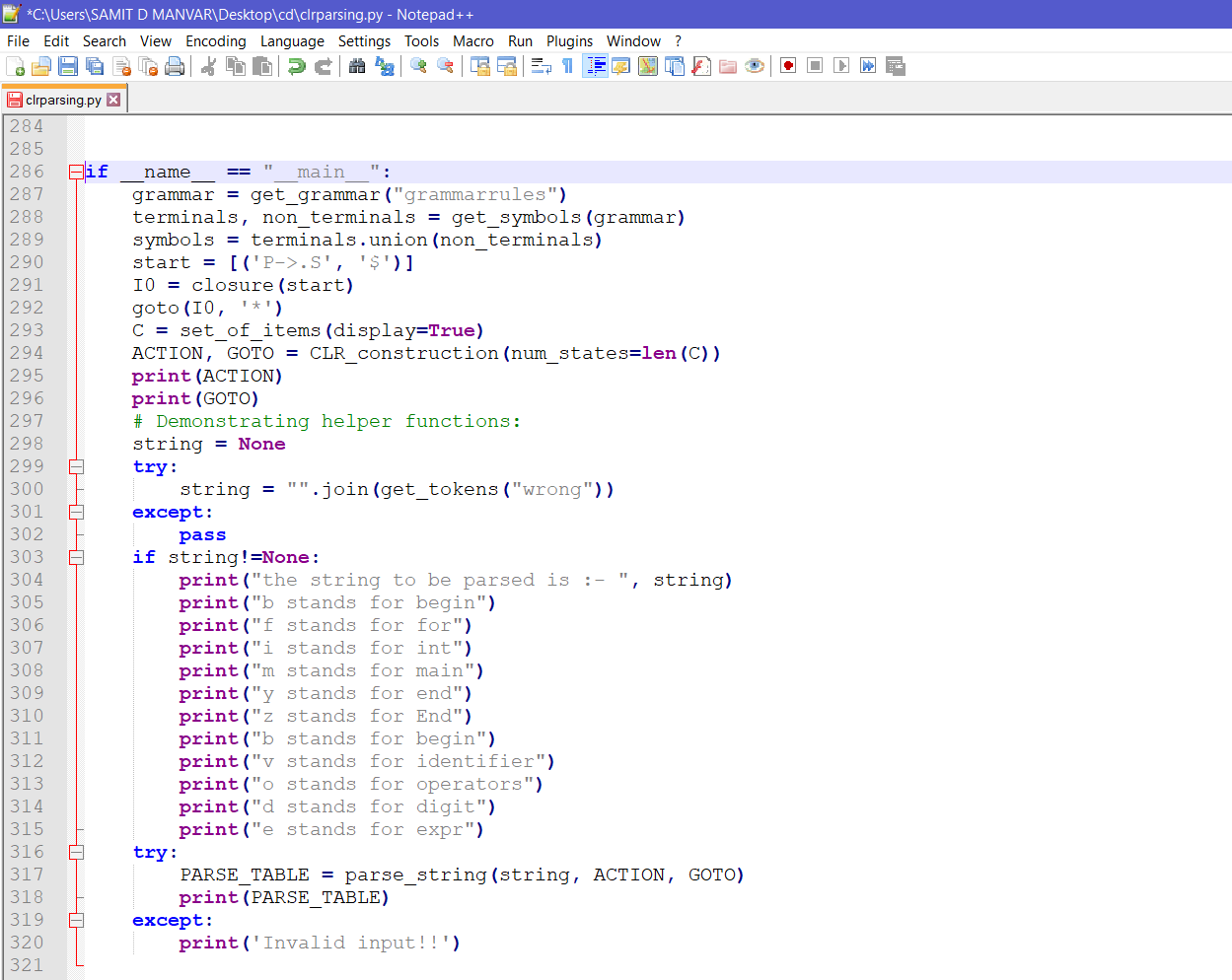








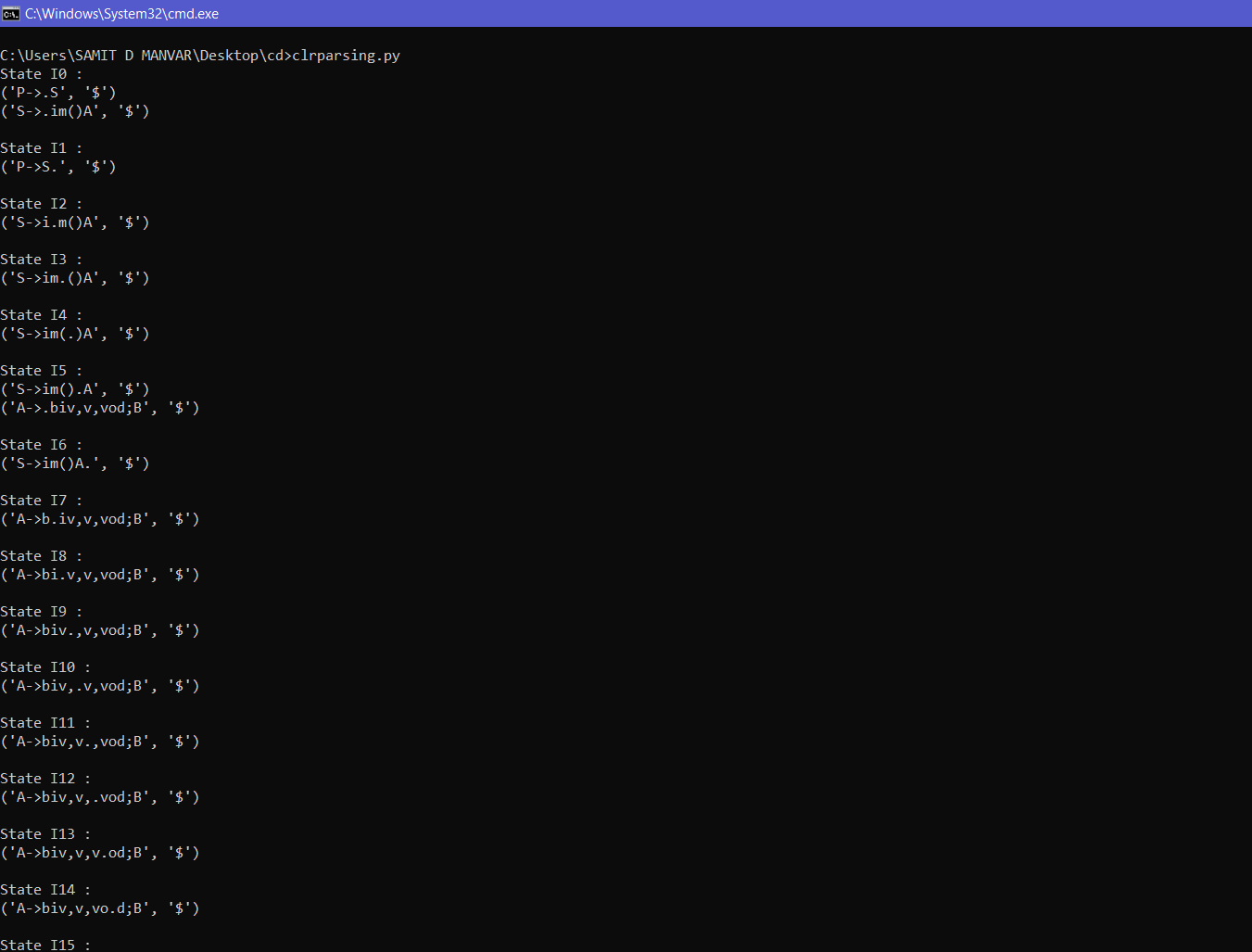


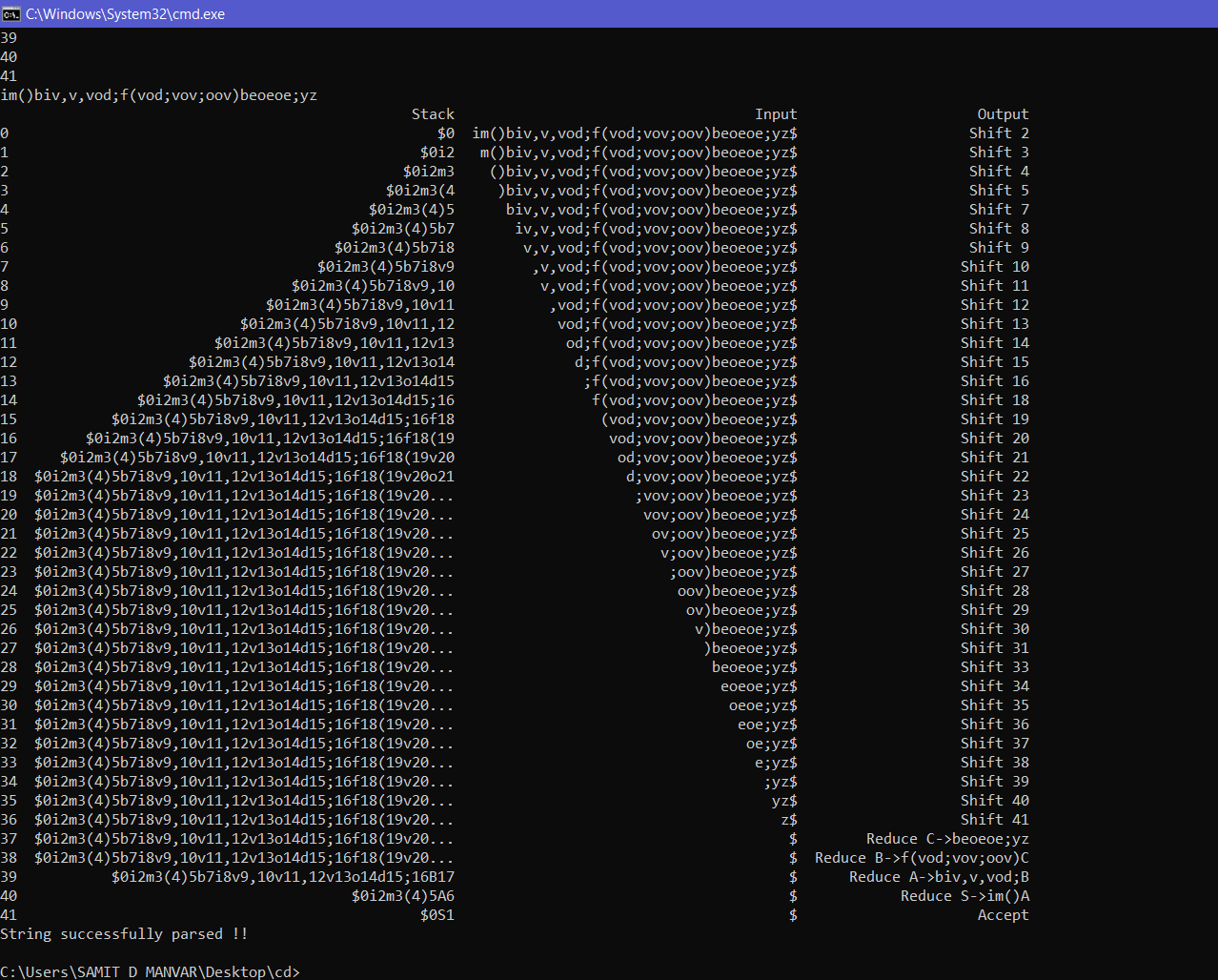


**RESULT**

The CLR Parsing algorithm applied to the given pseudocode or Hypothetical language produces 41 states – I0 TO I40. The string is successfully parsed as shown below.







**CONCLUSIONS AND FUTURE WORK**

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.

In our mini project we have successfully implemented first two phases of compiler – Lexical analyser and Parser Phase (Syntax analyser). We have successfully generated tokens in lexical phase and successfully implemented CLR Parser to generate parsing table and parse the string using python. Future works can include implementing other phases of compiler such as sematic analysis, intermediate code generator, code optimization, code generation.

**REFERENCES**

[1] <https://www.javatpoint.com/compiler-tutorial>

[2] <https://www.tutorialspoint.com/compiler_design/index.htm>

[3] <https://www.geeksforgeeks.org/compiler-design-tutorials/>