**Implementation of Simple Python Clone Compiler**

Project submitted to the

SRM University – AP, Andhra Pradesh

for the partial fulfillment of the requirements to award the degree of

**Bachelor of Technology**

In

**Computer Science and Engineering**

**School of Engineering and Sciences**

Submitted by

**Candidate Name**

**P. Durga Mahesh - AP21110010911  
G.Jashit - AP21110010921  
M.Karthik - AP21110010967  
N.Durga Naveen - AP21110010959**

**A picture containing text

Description automatically generated**

Under the Guidance of

**Dr. Jaya Lakshmi Tangirala**

**SRM University–AP  
Neerukonda, Mangalagiri, Guntur   
Andhra Pradesh – 522 240  
Nov, 2023**

# Certificate

Date: 2-Dec-23

This is to certify that the work present in this Project entitled “**Implementation of Simple Python Clone Compiler**” has been carried out by **Durga mahesh, Jashit Goli, Durga Naveen, Karthik** under Jaya Lakshmi Tangirala supervision. The work is genuine, original, and suitable for submission to the SRM University – AP for the award of Bachelor of Technology/Master of Technology in **School of Engineering and Sciences**.

**Supervisor**

(Signature)

Dr. Jaya Lakshmi Tangirala

Assistant Professor,

Affiliation.

# Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Dr. Jaya Lakshmi Tangirala, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of developing and writing of this project. I could not have imagined having a better advisor and mentor for my project . I cannot express enough thanks to my group members for their continued support and encouragement. I offer my sincere appreciation for the learning opportunities provided by my committee.

# Table Of Contents

Contents

[Certificate 1](#_Toc152437483)

[Acknowledgements 2](#_Toc152437484)

[Table Of Contents 3](#_Toc152437485)

[1.Introduction 4](#_Toc152437486)

[1.1 Data Types 4](#_Toc152437487)

[1.2 Architecture of Language 4](#_Toc152437488)

[Methodology 5](#_Toc152437489)

[Lexical Analysis 5](#_Toc152437490)

[Syntax Analysis (Parser) Design and Implementation 5](#_Toc152437491)

[Semantic Analysis 5](#_Toc152437492)

[Intermediate Code Generation Design and Implementation 5](#_Toc152437493)

[Target Code Generator 5](#_Toc152437494)

[Tools used 6](#_Toc152437495)

[2.1Context Free Grammer (CFG) 6](#_Toc152437496)

[2.1.1 Function 6](#_Toc152437497)

[2.1.3 While 7](#_Toc152437498)

[Code Implementation 8](#_Toc152437499)

[3.1 Function 8](#_Toc152437500)

[3.1.1 Function.l 8](#_Toc152437501)

[3.1.2 Function.y: 9](#_Toc152437502)

[3.2 List 11](#_Toc152437503)

[3.2.1 List.l 11](#_Toc152437504)

[3.2.2 List.y 12](#_Toc152437505)

[3.3 While 13](#_Toc152437506)

[3.3.1 While.l 13](#_Toc152437507)

[3.3.2 While.y 13](#_Toc152437508)

[4.Result 15](#_Toc152437509)

[4.1 Function 15](#_Toc152437510)

[4.2 List 16](#_Toc152437511)

[4.3 while 16](#_Toc152437512)

[5.Conclusion 17](#_Toc152437513)

# 1.Introduction

This Mini Compiler , contains three phases of compiler has been made for the language Python by using C language and we used Python + HTML language itself for target parser generation as well.

The constructs have been focused on ‘Function declaration’ and ‘List’ statements. Syntax Errors are also handled by the lex tool.

## Data Types

1.Integer

2.float

3.String

## 1.2 Architecture of Language

For this mini-compiler, the following aspects of the Python language syntax have been covered:

•Constructs like ‘Def’ and ‘list’ and the required indentation for these loops(While).

Specific Error Messages are displayed based on the type of error. Syntax errors are handled using the yyerror() function, while the semantic errors are handled by making a call to a function that searches for a particular identifier in the symbol table. It displays that we have entered an wrong syntax. As a part of error recovery, panic mode recovery has been implemented for the Lexer. It recovers from errors in declaration. In case of identifiers, when the name begins with a digit, the compiler neglects the digit and considers the rest as the identifier name.

Languages used to develop this project:

YACC and LEX

Target Language:

Python+HTML

# Methodology

## Lexical Analysis

## 

The input source code was tokenized at the lexical analysis stage. To identify and classify lexical items including operators, literals, identifiers, and keywords, a lexer was put into place. Lexer rules were largely defined via finite automata and regular expressions.

## Syntax Analysis (Parser) Design and Implementation

Analysis of syntax based on an assumed context-free language should be possible for the parser. The source code's hierarchical structure should be represented by a Parse Tree that is created. The parser should also be able to handle syntax problems with ease and produce helpful error messages. The implementation of the parser needs to give priority to accuracy and efficiency.

## Semantic Analysis

The goal of the semantic analysis stage was to verify the source code's meaning by representing it as an Abstract Syntax Tree . To keep track of identifiers and their characteristics, symbol tables were utilized. To ensure the accuracy of the programs, type checking, scope resolution, and other semantic checks were included.

## Intermediate Code Generation Design and Implementation

There are several stages involved in designing an intermediate code generator. The output of a compiler's syntax analysis stage is an Abstract Syntax Tree (AST), which is taken by the generator and converted into an intermediate representation that can be further optimized before being translated into the target code.

## Target Code Generator

The phase of code generation converted the verified source code into the desired machine code. The target assembly language was translated into, and intermediate code was produced, optimized, and translated. Target design and optimization options were taken into consideration.

# Tools used

We have used the lex and yacc tools to run and compile the code upto semantic analysis.

Filename.l :- It is a Lexical Analyzer file which defines all the terminals of the productions stated in the yacc file. It contains regular Expressions. It will generate a file lex.yy.c for getting the output.

Filename.y :- Yacc file is where the productions for the conditional statements and expressions are mentioned. This file also contains the semantic rules defined against every production necessary. After running the command bison -d Filename.y it will generate two files one file is Filename.tab.c and Filename.tab.h which will contain parser generator and header tags which were needed by the lex file to run by using GCC.

The output will be stored in the a.exe file

## 2.1Context Free Grammer (CFG)

### 2.1.1 Function

F=> DEF ‘(‘ N’)’ ‘:’ V  
 N => Identifier ‘(‘ P’)’  
 P => String ‘,’P/Numbers ‘,’P /Decimal ‘,’ P/Identifiers‘,’P/String/Numbers/Decimal/Identifiers  
 V => Starprint ‘(‘E’)’ end print V/return ‘(‘E’)’  
 E => Identifier/Identifer op Identifer  
 DEF => def  
 String => [a-z A-Z] \*  
 Numbers => [0-9] \*  
 Decimal => [0-9] \* \ . [0-9] \*   
 Identifier => [ a-z A-Z] [a-z A-Z 0-9] \*  
 Op => +/-/\*/%”/</>/==  
 Start print => <print>  
 End print = > </print>

**2.1.2** List

Identifier '=' V

V=> '('A')'

A=> String ‘,’A/Numbers ‘,’A /Decimal ‘,’ A/String/Numbers/Decimal

Identifier=> [ a-z A-Z] [a-z A-Z 0-9]\*

String=> [a-z A-Z]\*

Numbers=> [0-9]\*

Decimal=> [0-9]\*\.[0.9]\*

### 2.1.3 While

WHILE '('c')' ':'P E

C => T/T Relop c

T => Identifier /N

N => Number/Decimal.

E => Else':'P

﻿P => start print '(' T ')' End print.

WHILE => while.

ELSE => Else.

Start Print => <Print>

End Print => </Print>

Number => [0-9]\*

Decimal => [0-9]\*\[0-9]\*

String => [a-zA-Z]\*

Identifier => [a-zA-Z][a-zA-Z0-9 ]\*

Relop => </>/<=/>=/=/==/<>

# Code Implementation

## 3.1 Function

### 3.1.1 Function.l

%{

#include "function.tab.h"

%}

%%

"def" { return DEF; }

"<print>" { return STARTPRINT; }

"</print>" { return ENDPRINT; }

"return" { return RETURN; }

[0-9]\* { return NUMBER; }

[0-9]\*\.[0-9]\* { return DECIMAL; }

\'[a-zA-Z]\*\' { return STRING; }

[a-zA-Z][a-zA-Z0-9]\* { return IDENTIFIER; }

["+" | "-" | "\*" | "%" | "//" | "<" | ">" | "=="] { return OP; }

\n return 0;

. return yytext[0];

%%

### 3.1.2 Function.y:

%{

#include<stdio.h>

#include<string.h>

%}

%token NUMBER

%token DECIMAL

%token STRING

%token IDENTIFIER

%token OP

%token DEF

%token STARTPRINT

%token ENDPRINT

%token RETURN

%%

F: DEF '(' N ')' ':' V{ printf("correct");}

;

N: IDENTIFIER '(' P ')'

;

P :

| STRING','P

| NUMBER','P

| DECIMAL','P

| IDENTIFIER','P

| STRING

| NUMBER

| DECIMAL

| IDENTIFIER

;

V : STARTPRINT '(' E ')' ENDPRINT

| RETURN '(' E ')'

;

E : IDENTIFIER

| IDENTIFIER OP IDENTIFIER

;

%%

int main(){

yyparse();

}

int yywrap(){

return 1;

}

void yyerror(char \*s){

printf("Error %s",s);

}

## 3.2 List

### 3.2.1 List.l

%{

#include "list.tab.h"

%}

%%

[0-9]\* {yylval=atoi(yytext);

return NUMBER;

}

[0-9]\*\.[0-9]\* {yylval=atof(yytext);

return DECIMAL;

}

\'[a-zA-Z]\*\' {yylval=yytext;

return STRING;

}

[a-zA-Z][a-zA-Z0-9]\* {yylval=yytext;

return IDENTIFIER;

}

\n return 0;

. return yytext[0];

%%

### 3.2.2 List.y

%{

#include<stdio.h>

%}

%token NUMBER

%token DECIMAL

%token STRING

%token IDENTIFIER

%%

S: L { printf("correct");}

;

L: IDENTIFIER'='V { $$ = $1 = $3; } ;

V: '('A')' { $$ = ( $2 ); }

;

A :

| STRING','A { $$ = $1,$3; } | NUMBER','A { $$ = $1,$3; } | DECIMAL','A { $$ = $1,$3; } | STRING { $$ = $1; }

| NUMBER { $$ = $1; }

| DECIMAL { $$ = $1; }

;

%%

int main(){

yyparse();

}

int yywrap(){

return 1;

}

void yyerror(char \*s){

printf("Error %s",s);

}

## 3.3 While

### 3.3.1 While.l

%{

#include "while.tab.h"

%}

%%

"while" { return WHILE; }

"else" { return ELSE; }

"<print>" { return STARTPRINT; }

"</print>" { return ENDPRINT; }

[0-9]\* { return NUMBER; }

[0-9]\.[0-9] { return DECIMAL; }

\'[a-zA-Z]\*\' { return STRING; }

[a-zA-Z][a-zA-Z0-9]\* { return IDENTIFIER; } ["<"|">"|"<="|">="|"="|"=="|"<>"] { return RELOP; } \n return 0;

. return yytext[0];

%%

### 3.3.2 While.y

%{

#include<stdio.h>

%}

%token NUMBER

%token DECIMAL

%token STRING

%token IDENTIFIER

%token RELOP

%token WHILE

%token STARTPRINT

%token ENDPRINT

%token ELSE

%%

L: WHILE '(' C ')' ':' P E { printf("correct"); } ;

C: T

| T RELOP C

;

T: IDENTIFIER

| N

;

N: NUMBER

| DECIMAL

;

E:

| ELSE ':' P

;

P: STARTPRINT '(' T ')' ENDPRINT

;

%%

int main(){

yyparse();

}

int yywrap(){

return 1;

}

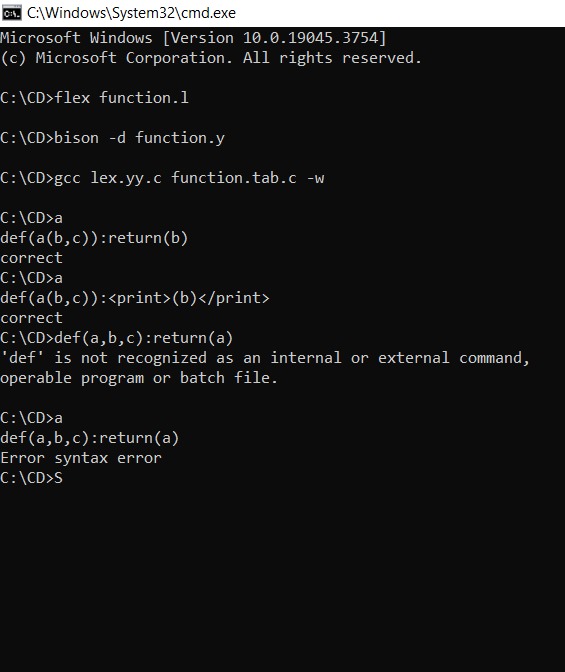
void yyerror(char \*s){

printf("Error %s",s);

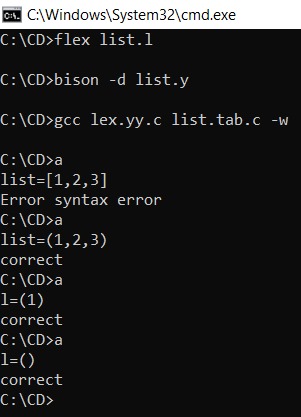
}

# 4.Result

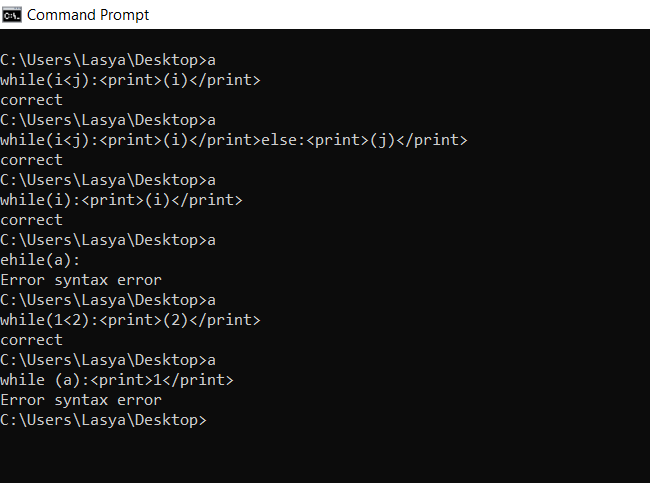
## 4.1 Function



## 4.2 List



## 4.3 while



# 5.Conclusion

This mini-compiler can be enhanced to a complete compiler for the Python language by making a few improvements. User defined functions can be handled and the functionality of importing libraries and calling library functions can be taken care of.  
The output can be made to look more enhanced and beautiful. The overall efficiency and speed of the program can be improved by using some other data structures, functions or approaches.