CPyter: C Interpreter using Python

Sonali Suri
sosuri@ucsc.edu

***Abstract*—There are a number of computer languages which are used for writing computer applications, but amongst those, the computer programming language C is the most popular language worldwide. Micro-controllers to operating systems, everything is written in C since it is very flexible and versatile and allows maximum control with minimal commands. C programming language is recognized worldwide and used in a multitude of applications, including advanced scientific systems and operating systems. On the other hand, Python is a high level, interpreted and general-purpose dynamic programming language that focuses on code readability.[2] It has fewer steps when compared to C. The main difference between C and Python is that, C is a structure-oriented programming language, while Python is an object-oriented programming language. In general, C is used for developing hardware operable applications, and Python is used as a general-purpose programming language. Python has fully formed built-in and pre-defined library functions, but C has only a few built-in functions. We know that the "standard" Python Interpreter is written in C (also known as CPython). This project is about writing a Compiler for C using Python as that would help us explore the syntax and semantics for both the languages and understand the AST (Abstract Syntax Tree) for C programming language while developing the Parser and Interpreter for C using Python.**

I. Introduction

The motivation for this project was to learn more about Compilers and enjoy the experience by building an Interpreter for C. This report, introduces a C compiler using Python called CPyter. CPyter has been created from basic Python programming without any third-party libraries. The rest of the report is as follows: Section II we go over the scope of the project, in Section III we describe the architecture of CPyter, in Section IV we discuss the working of Cpyter in detail, in Section V we show the results of the Cpyter on different test cases.

II. Scope

The scope of the project was to create a simple C Compiler with Python called CPyter. [1] CPyter will have following features:

1. Supports the following arithmetic operators:

Operator	Description	Example
+	Add operands	x+y
-	Subtract second operand from first	x-y
*	Multiply both operands	x*y
/	Divide numerator by denominator	x/y
%	Modulus operation	x%y
++	Increment value by 1	x++
–	Decrement value by 1	x–

2. Supports the following logical operators, assuming variable **x** holds True and **y** holds False:

Operator	Description	Example
&&	AND operator	(x&&y) = False
||	OR operator	(x||y) = True
!	NOT operator	!(x&&y) = True

3. Supports the following relational operators, assuming variable **x** holds 2 and **y** holds 3:

Operator	Description	Example
==	Double equals operator	(x==y) = False
!=	Not equals operator	(x!=y) = True
>	Greater then operator	(x>y) = False
<	Less then operator	(x<y) = True
>=	Greater then equals to operator	(x>=y) = False
<=	Less then equals to operator	(x<=y) = True

4. Supports the following assignment operators:

Operator	Description
=	Assignment operator
+=	Add and assignment operator
-=	Subtract and assignment operator
/=	Divide and assignment operator
*=	Multiply and assignment operator

5. Supports the following loops:

Loop	Description
while	Repeats one or more statements when a condition is true
for	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable
nested	Using more than one loop in nested fashion

6. Creating functions and calling the functions.
7. Support for standard input/output i.e. supporting user given input and giving output to the terminal.
8. Support for comments.

III. Architecture of Cpyter

Fig 1. shows the Architecture of Cpyter. Cpyter has four main components which are as follows:

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Interpreter

Lexical Analyzer takes source code as input and gives Tokens as output. The output Tokens from Lexical Analyzer is given as input to Syntax Analyzer for syntax check, and Syntax Analyzer gives the Abstract Syntax Tree as output.[6] The Abstract Syntax Tree from Syntax Analyzer is provided as input to Semantic Analyzer for semantic check, and Semantic Analyzer gives the same Abstract Syntax Tree as output. The Interpreter takes Abstract Syntax Tree as input from Semantic Analyzer for evaluation and outputs the results of the input source code.

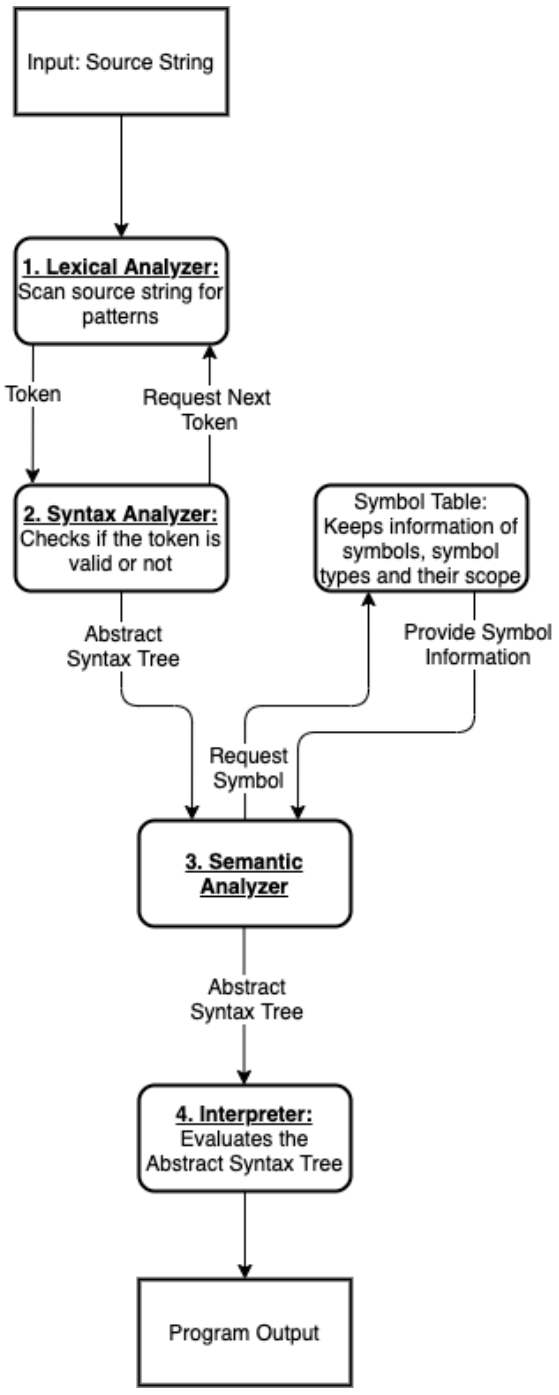

Fig.1. Architecture of Cpyter

IV. Working of Cpyter

In this section, we go over the working of Cpyter in detail by describing the working of four main individual architecture components mentioned in the previous section.

A. Lexical Analyzer

Lexical Analyzer is also called as Lexer. Lexer is a first phase of the Cpyter which takes the program as input. The Lexer scans the input program. After scanning the program characters the Lexer converts these sequence of characters into a sequence of Tokens, this process done be Lexer is also called as Tokenization. Lexer looks for specific patterns while scanning and matches these patterns and creates the Tokens for these patterns. These specific patterns which are tokenized are also called as Lexeme, which corresponds to a word in linguistics. The scanning of the patterns and tokenizing is based on Finite State Machine. Fig 2. shows the lexer flow diagram.

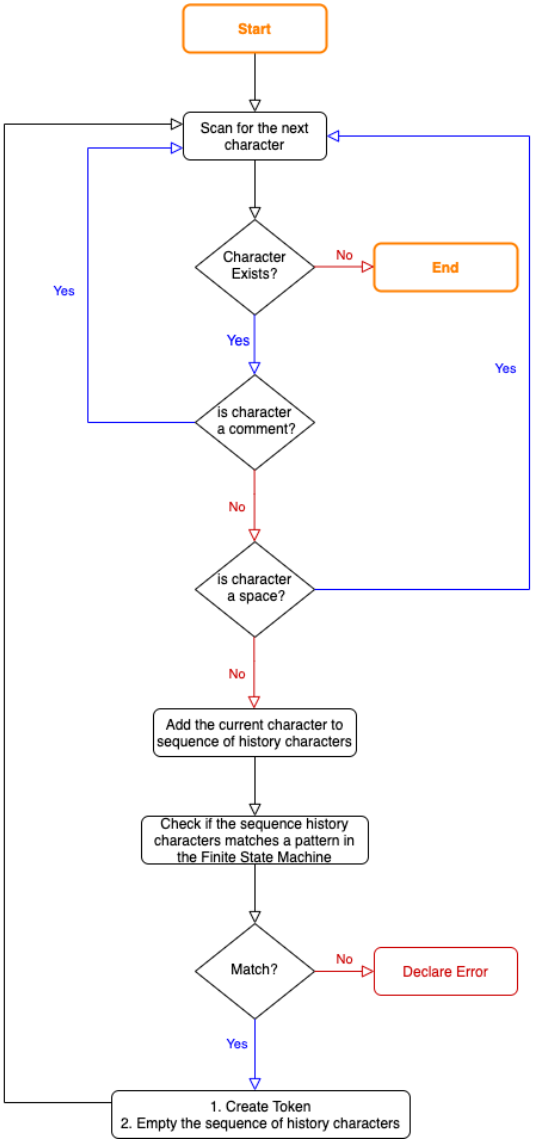

Fig.2. Lexer Flow Diagram

The definitions for Some Tokens that Lexer scans for are as follows:

1. Identifier: Programmer chosen names
2. Keyword: Built in names in the program language
3. Separator: Punctuation Characters
4. Operators: Symbols to perform operations
5. Comment: line of human readable information
6. Literal: numeric, logical, reference literals

Some example Lexemes for the above tokens are as follows:

Sr No	Token Name	Lexeme
1	identifier	x, col, down
2	keyword	switch, while, for
3	separator	{, }
4	operator	-, ++
5	comment	// this is comment
6	literal	true, false

B. Syntax Analyzer

Syntax Analyzer is the second phase component of the Compiler. The Syntax Analyzer is also called Parser. Fig 3. shows the flow diagram of the Parser.

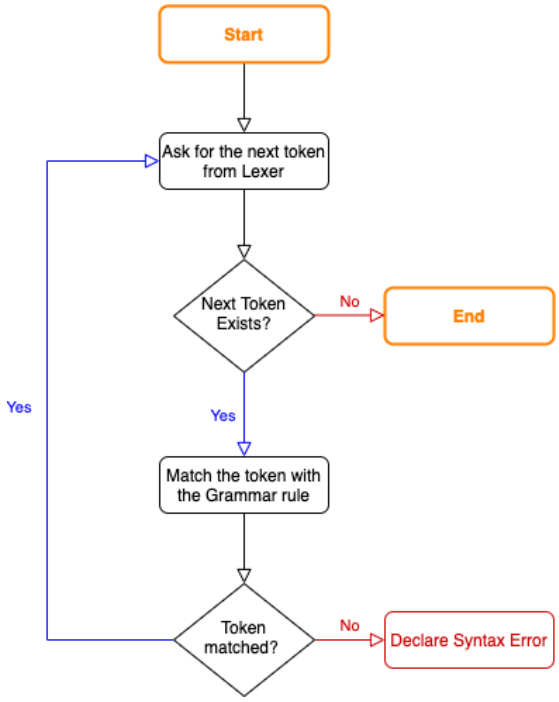

Fig.3. Parser Flow Diagram

Fig.4. shows how lexer generates the tokens for different loops.

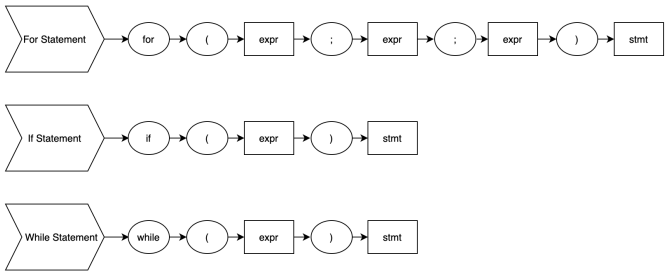

Fig.4. Syntax diagram for control statements

Parser takes the tokenized program from Lexer as input. [3] The Parser compares the tokenized program with the Context Free Grammar of the program language and gives a Parsed Abstract Syntax Tree as output. If the tokenized program violates the rules of Context-Free Grammar, then the Parser will give a syntax error. In general, there are two types of Parsers:

1. Top-down Parser: Top-down Parser builds the Abstract Syntax Tree from root to leaf. Top-down Parser follows the leftmost derivations, where the leftmost non-terminal in each sentential is always chosen.
2. Bottom-up Parser: Bottom-up Parser builds the Abstract Syntax Tree from leaves to root. Bottom-up Parser follows the rightmost derivations, where the Parser reduces the input string to start symbol of the grammar.[4]

Our Compiler project uses a top-down Parser to parse the tokenized program. In our project, CPyter Parser continuously interacts with the CPyter Scanner. Parser asks for the current Token from the Lexer, and then the Parser tries to match the Token with the Grammar rule of the language. If the match of the Token to the Grammar rule is successful, then the Parser asks for the next Token from the Lexer.

C. Semantic Analyzer

Semantic Analyzer provides meaning to the language's constructs, like Tokens and syntax structure. Semantic Analyzer tells us whether the input source program is meaningful or not. For example, consider the below code:

```
char variableName = "hello";
```

The code is correct lexically and syntactically. The above code will result in an error by Semantic Analyzer. Semantic Analyzer gives interpreting information for symbols, their data types and relation between the symbols. Semantic Analyzer captures all the symbols by storing the symbol's name, type and scope in a symbol table. With the help of the Symbol Table, the Semantic Analyzer analyzes the semantics of Abstract Syntax Tree provided by the Parser and gives this Abstract Syntax Tree to Interpreter.

D. Interpreter

The Interpreter is the last component of the Compiler. Interpreter evaluates the Abstract Syntax Tree. Interpreter executes all the operations in the Abstract Syntax Tree and prints the output in the terminal.[5]

V. Test Cases

This section shows the results of CPyter on different input programs.[7] Fig 5. is a simple test case which prints ”Hello World”, and the result shows that CPyter can handle print statements.

```
#include<stdio.h>
void main()
{
printf("Hello World!");
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_1.c
Hello World!
```

Fig.5. Print Hello World!

Fig 6. is a test case ”Square of a number” which demonstrates CPyter can handle arithmetic operations.

```
#include<stdio.h>
int square(int x)
{
return x*x;
}
void main()
{
int a=5;
printf("The value for a is %d\n",a);
printf("The square of the number is %d ",square(a));
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_2.c
The value for a is 5
The square of the number is 25
```

Fig.6. Square of a number.

Fig 7. is a test case ”User input” which demonstrates CPyter can user inputs from the terminal.

```
#include<stdio.h>
void main()
{
int x=0;
scanf("%d",&x);
printf("The value for x is %d",x);
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_3.c
10
The value for x is 10
```

Fig.7. User input

Fig 8. is a test case ”For Loop” which demonstrates CPyter can handle ”for” loops.

```
#include<stdio.h>
void main()
{
int r=5;
int i=0;
for (i=0;i<=r;i++)
{
printf("%d\n",i);
}
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_4.c
0
1
2
3
4
5
```

Fig.8. For Loop

Fig 9. is a test case ”While Loop” which demonstrates CPyter can handle ”while” loops.

```
#include<stdio.h>
void main()
{
int num1=0;
int num2=1;
int result=0;
int n=0;
int i=0;
printf("Enter the number of fibonacci terms ");
scanf("%d ",&n);
printf("%d ",num2);
while(i<n-1)
{

result=num1+num2;
printf("%d ",result);
i++;
num1=num2;
num2=result;
}
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_10.c
Enter the number of fibonacci terms 11
1 1 2 3 5 8 13 21 34 55 89
```

Fig.9. While Loop

Fig 10. is a test case ”Sum of two numbers” which demonstrates CPyter can handle arithmetic operations.

```
#include<stdio.h>
int sum(int x,int y)
{
return (x+y);
}

void main()
{
int x=0;
int y=0;
int result=0;
printf("Enter the first number ");
scanf("%d",&x);
printf("Enter the second number ");
scanf("%d",&y);
result= sum(x,y);
printf("The sum of %d and %d is %d",x,y,result);
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_7.c
Enter the first number 12
Enter the second number 18
The sum of 12 and 18 is 30
```

Fig.10. Sum of two numbers

Fig 11. is a test case ”Comments” which demonstrates CPyter can handle ”C” language comments.

```
#include<stdio.h>
void main()
{
/*Hello World*/

printf("This is an example for comments");
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_8.c
This is an example for comments
```

Fig.11. Comments

Fig 12. is a test case ”Prefix / Postfix increments” which demonstrates CPyter can handle arithmetic operations.

```
#include<stdio.h>
void main()
{
printf("Prefix example\n");
int a = 1;
int b=++a;
printf("a= %d\n",a);
printf("b= %d\n",b);

printf("Postfix example\n");
int d = 1;
int e = d++;   // d = 1
int f = d;
printf("d= %d\n",d);
printf("e= %d\n",e);
printf("f= %d\n",f);
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_9.c
Prefix example
a= 2
b= 2
Postfix example
d= 2
e= 1
f= 2
```

Fig.12. Prefix / Postfix increments

Fig 13. is a test case ”Other operations” which demonstrates CPyter can handle logical operations, relational operations and ”if” statements.

```
#include<stdio.h>
void main()
{
if (0<1 && 5>=4)
{
if (7<=6 || 3>1)
{
printf("Condition is true");
}
}
else
{
printf("Condition is false");
}
}
```

```
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ vim test_11.c
(base) Sonalis-MacBook-Pro:CCompiler sonalisuri$ python3 ccompiler.py -f test_11.c
Condition is true
```

Fig.13. Other operations

VI. Conclusion and Future Work

In this paper, we introduced a new C compiler with Python called CPyter, which can perform logical operations, arithmetic operations, assignments to symbols, execute operations in loops, and can perform function calls. Currently, CPyter can not interpret C functionalities other than ones mentioned in the scope like working with pointers, performing memory allocation, error handling, etc.. Everything else has been left for future work of the CPyter.

References

[1] https://github.com
[2] https://en.wikipedia.org/wiki/Compiler
[3] “Context Free Grammar” [Online]. Available: https://www.geeksforgeeks.org/classification-of-context-free-grammars/
[4] “Role of Parser” [Online]. Available: https://www.geeksforgeeks.org/introduction-of-parsing-ambiguity-and-parsers-set-1/
[5] C. Xing. “How Interpreters Work: An Overlooked Topic in Undergraduate Computer Science Education,” Proc. In CCSC Southern Eastern Conference, JCSC Vol. 25,Iissue 2. December 2009
[6] R. Sebesta, Concepts of Programming Languages, 10th Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2012.
[7] “Welcome to Python.org,” Python.org. [Online]. Available: https://www.python.org/.