
# 50.051 Programming Language Concepts

## Project Final Report

**Python-to-C**

Nicholas Gandhi (1005295)

Cheh Kang Ming (1005174)

Goh Yu Fan (1005054)

**Link to codebase:**
https://github.com/fish-r/Python-To-C

# 1.Problem Statement

In the realm of programming languages, interoperability and code portability have become essential aspects of software development. Bridging the gap between Python and C, this project introduces a novel tool designed to compile Python source code into equivalent C code. By leveraging the capabilities of the C programming language, known for its efficiency and performance, this tool aims to provide a pathway for integrating Python code seamlessly into C-based applications.

# 2.Methodology

**The task for the problem statement is as such:**
Given an input Python(input.py) file, we should generate the output file(s), output.c and output.h that reflects the execution of the Python code but in C.

## Constraints

To achieve this, we have several underlying constraints. These constraints must be observed in order for the programme to be working accurately.

**Syntax and Semantics of input Python file**
The input python file must be syntactically and semantically correct. This means that the Python file should be able to execute successfully without errors. Additionally, there should be no redeclaration of existing variables.

**Syntax and Semantics of output C file**
The output C file should be syntactically correct, but not expected to be semantically correct.

**Python Keywords Coverage**
There will be several keywords that will not be implemented. They are:

1. Global
2. Yield
3. And
4. As
5. Assert
6. Class
7. Del
8. Except
9. Exec
10. Finally
11. From
12. Import
13. Is
14. Lambda
15. Not
16. Or

17. Pass
18. Raise
19. Try
20. With

There will be several operators that will not be implemented. They are:

1. **
2. //
3. @
4. :=

There will be several delimiters that will not be implemented. They are:

1. {
2. }
3. //
4. @=
5. **=

**Python Data Types**

The input Python file will only have these data types:

1. String
2. Integer
3. Float
4. Lists of a single data type (No type mixing and no nested list)

**Python Function Return Data Types**

The input Python file can only return literals (excluding lists) but not identifiers (e.g. return “asd”)

**Python Programming Patterns**

The input Python file will not implement any Object Oriented Programming. The file will also not contain list comprehensions.

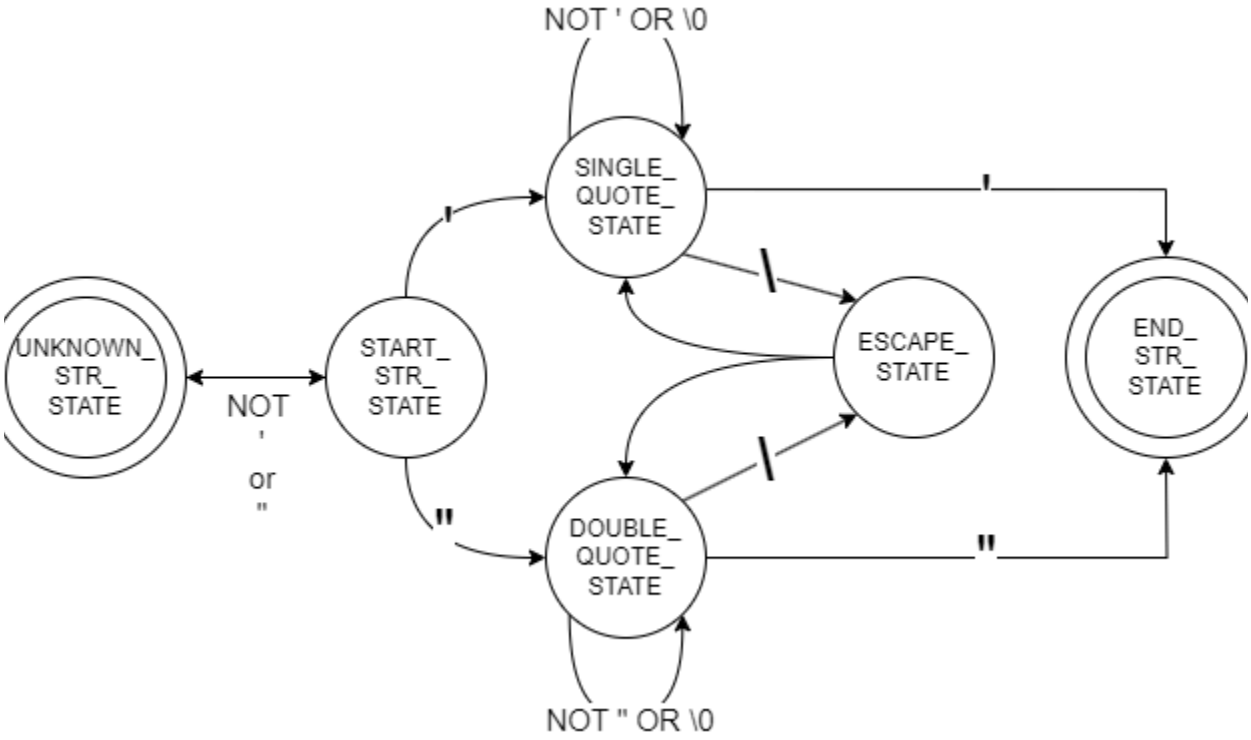
# 3.Implementation

## Lexer

The lexer reads the input python code and breaks it down into small, meaningful chunks called tokens. The lexer has to have the ability to differentiate between literals, python keywords, identifiers, operators, and comments from the given python source code.

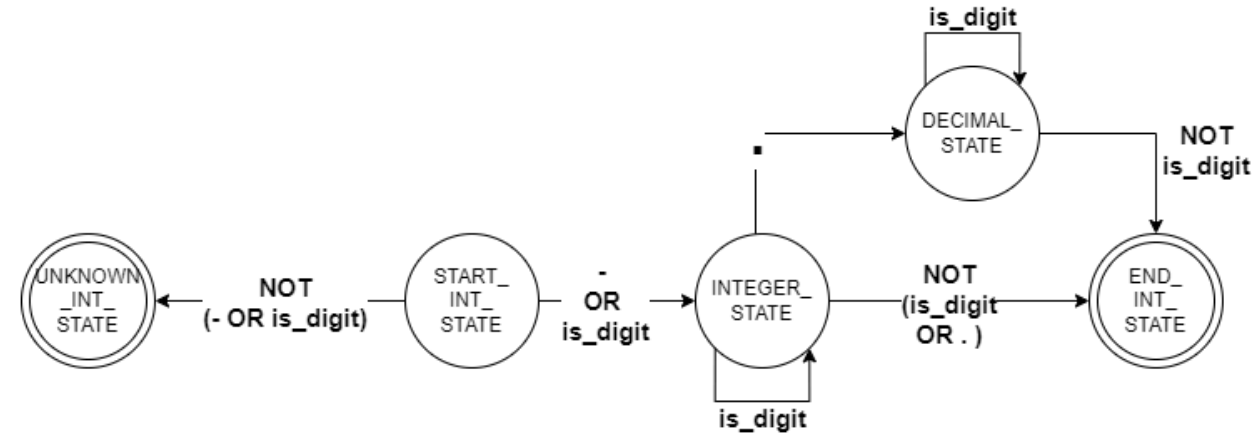
## Literal FSM

For the checking of literals, we implemented Finite State Machines (FSMs) to detect whether it is a string literal, integer/float literal and list literals (list_int , list_float, list_string). The FSM diagrams corresponding to the PythonTokenType function are illustrated in the figures below.

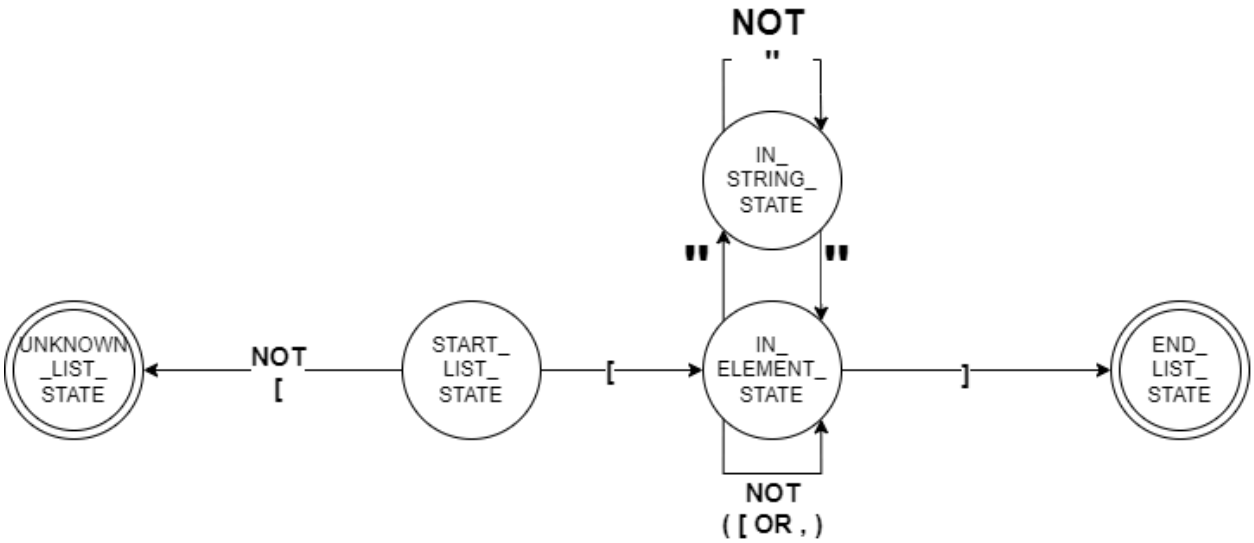



*is_python_string* FSM Diagram

```
int is_digit(char c) { return c >= '0' && c <= '9'; }
```




*is_python_numeric* FSM Diagram




*is_python_list* FSM Diagram

## Parser

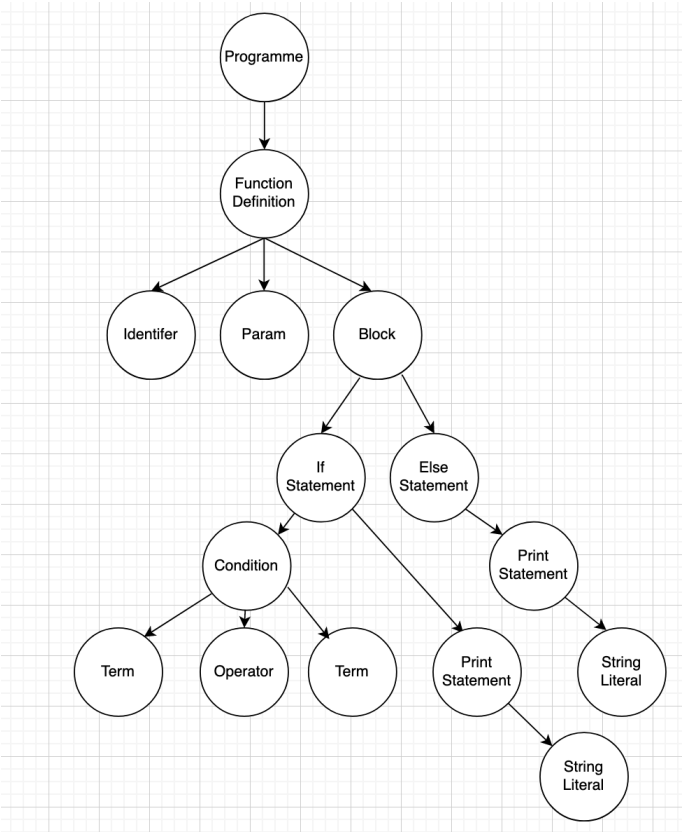
The parser then takes in a stream of python tokens generated by the lexer and builds an Abstract Syntax Tree (AST) based on the type of token. An AST is used in this step to abstract out the syntax of the python code (parenthesis, colons, commas, semi-colons etc), ensuring a smoother code conversion process.

The parser creates different tree nodes of type struct TreeNode depending on the type of python token. The type of node then determines how the writer should process and write the corresponding C code to the output file.

The TreeNode Struct contains the following variables:

```
 6   typedef struct TreeNode
 7   {
 8       char *label;
 9       struct TreeNode *parent;
10       struct TreeNode **children;
11       size_t num_children;
12       Token *token;
13   } TreeNode;
```

The children of a Node consist of the essential components required to . For example, the children of a ‘IfStatement’ node would consist of the ‘Condition’ node, representing the condition of the if statement.




Sample structure of the AST

## C-Writer

The main function of the C Writer (contained in c_writer.c file) is to convert the AST into C code. The AST is traversed recursively, per the pseudocode below. There are thus three main components, which are traverse tree, setting state and processing a node.

**Traverse Tree**

To write the Python code in C, we traversed the AST recursively and performed operations in each traversal, such as setting state and processing the node. This ensures that the C code will respect the scope and order of the input Python code.

```
traverse_tree(TreeNode * root, State *state, NodeQueue * queue, …):
        // save nodes in global scope to process later
        If node in global scope:
                Enqueue_node( )
                Return
```

```
// update state based on current node
set_state(root, state, …)

// process the node and write to file
process_node(root, state, …)

// traverse and process children recursively
For child in root.children:
        traverse_tree(child, state, queue, …)

// on recursion exit, close scopes
If node is block:
        Write_to_file( " } ")

// on root node of entire tree, write the main function
if node is root:
        write_main( root, state, queue, …)
```

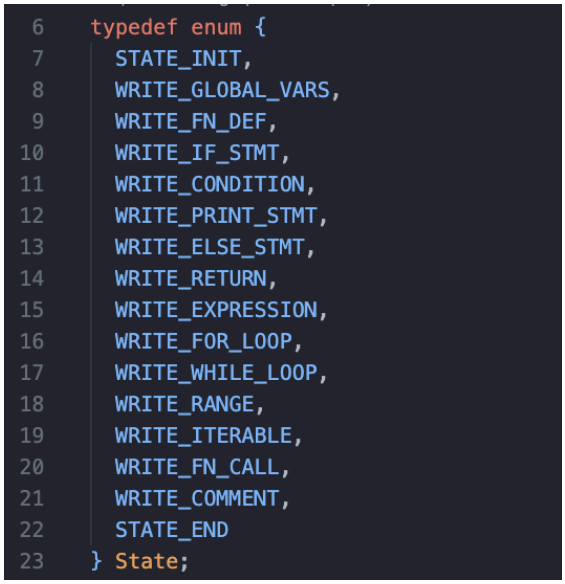
**Setting State**

Firstly, due to the reuse of various TreeNode labels (Identifiers, Parameters, EOL etc..), a finite state machine (FSM) is required to keep track of what kind of TreeNode is currently being processed, and thus write the respective syntax in C.

For example we would want to write a function definition in C. From the AST, given a parent TreeNode with label "FunctionDefinition" with respective child nodes with labels "Identifier" and "Parameter", the way to write the code in C is to declare the return type of the function and the subsequent return type of the parameters.

However, given a similar case where we want to write a function call, represented by a parent node with label "FunctionCall" and child nodes "Identifier" and "Parameter", it would be erroneous to declare the return type of the function call (and its parameters) in C.

Therefore, the use of an FSM is required to keep track of what kind of Node is currently being processed. The input for each state is the current state and the current node. In the subsequent process_node function, the states will determine what action will be taken (what C code will be written).



```
typedef enum {
  STATE_INIT,
  WRITE_GLOBAL_VARS,
  WRITE_FN_DEF,
  WRITE_IF_STMT,
  WRITE_CONDITION,
  WRITE_PRINT_STMT,
  WRITE_ELSE_STMT,
  WRITE_RETURN,
  WRITE_EXPRESSION,
  WRITE_FOR_LOOP,
  WRITE_WHILE_LOOP,
  WRITE_RANGE,
  WRITE_ITERABLE,
  WRITE_FN_CALL,
  WRITE_COMMENT,
  STATE_END
} State;
```

*Various States of the c_writer FSM*

**Process Node**
In the `process_node` function, C code will be written depending on what the current state and current node is.

# 4.Challenges

Through the implementation of this project, we encountered several challenges, some of which we overcame and some that can be improved on. Below are the challenges and the approach we took in the attempt to overcome them.

## Closing blocks in C

Python uses indentation to determine which lines of codes belong to the same block, which is a different approach to C that uses curly braces. This then posed a challenge to us on how to close blocks in C as we would not have information on when the block closes when we are traversing the tree.

## Identifiers and Return types

In order to print variables in C, we require the type to be declared, which is not required in Python. Hence, we implemented a lookahead to determine the identifier type. This is also done for function definition identifiers, in order to determine the return type.

## Printing For Loops

In Python, elements of a List can be iterated over easily, such as in the example: `for i in List`. To convert this into C, we will require not only the identifier *i* but also the Iterable *List.* To achieve this, we initiate temporary nodes which store *i* and *List* to be accessed in the Print function later.

### Main Function

Python does not require the use of a main function to execute code existing in the global scope (zero indented code). Thus, this presents a challenge, especially when there may be instances of global code mixed with function declarations in varying orders. In our attempt to solve this, we created a NodeQueue that will enqueue all nodes in the global scope (where parent node is “Program”) and delay processing until the recursion arrives at the root node of the tree at the final traversal.

Here, the nodes in NodeQueue are once again traversed recursively and processed in the same fashion as before to write the code into a main function.

## 5.Future Improvements

In the future, we aim to enhance our capability to accommodate additional Python data types like dictionaries. Additionally, we aspire to extend our parsing capabilities to encompass more intricate Python functionalities, including list comprehensions, ternary operators, and string formatting.

# Appendix

## A.Contributions

Goh Yu Fan: *Wrote parts of the lexer, and most of the c_writer.*
Cheh Kang Ming: *Wrote parseTree which creates the AST.*
Nicholas Gandhi: *Wrote the FSMs for parsing literals in lexer and most of lexer and helped in c_writer.*

## B.List of Labels Implemented for Abstract Syntax Tree (In no order)

- **FunctionDefinition**
- **FunctionCall**
- **EOL**
- **ForStatement**
- **Identifier**
- **Iterable**
- **Range**
- **Start**
- **Stop**
- **Step**
- **Parameter**
- **PrintStatement**
- **IfStatement**
- **Condition**
- **Term**
- **Operator**
- **ElseStatement**
- **ReturnStatement**
- **Expression**
- **Block**
- **WhileStatement**
- **Char - under Iterable node**
- **String - under Iterable node**
- **Comment**
- **ElifStatement**
- **ListFloat - for both under Iterable and normal list token**
- **ListInt - for both under Iterable and normal list token**
- **ListString - for both under Iterable and normal list token**

## C. SAMPLE INPUT AND OUTPUT

input > helloworld.py > ...

```python
# Integer Literal
a = 1
# Float Literal
b = 2.2

# String Literal
c = "abc"
d = 'abc'
# Char Literal
e = "a"
f = 'a'
# Empty string
g = ""
h = ''

# List Int Literal
i = [1, 2, 3]
# List Float Literal
j = [1.1, 2.5, 3.7]
# List String Literal
k = ["abc", "def", "ghi"]
```

output.c > main()

```c
int main() {
    /* # Integer Literal */
    int a = 1;
    /* # Float Literal */
    float b = 2.2;
    /* # String Literal */
    char c[] = "abc";
    char d[] = "abc";
    /* # Char Literal */
    char e = 'a';
    char f = 'a';
    /* # Empty string */
    char g = "";
    char h = "";
    /* # List Int Literal */
    int i[] = {1, 2, 3};
    /* # List Float Literal */
    float j[] = {1.1, 2.5, 3.7};
    /* # List String Literal */
    char* k[] = {"abc", "def", "ghi"};
return 0;
}
```

*Input of different literal types and its respective output*

input > helloworld.py > ...

```python
def print_function(x, y):
    list = [x, y]
    for i in list:
        print(i)
    for n in range(1, 2, 3):
        print(n)


for i in range(1, 2):
    print(i)
print_function(1, 2)

x = 3
z = "hel"
print(z)
```

output.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include "output.h"

void print_function(int x, int y)
{
    int list[] = {x, y};
    for (int i= 0; i < 1; i++)    {
        printf("%d", list[i]);
    }
    for (int n = 1; n < 2; n += 3)
    {
        printf("%d", n);
    }
}
int main() {
    for (int i = 1; i < 2; i += 1)
{
    printf("%d", i);
}
    print_function(1, 2);
    int x = 3;
    char z[] = "hel";
    printf(z);
return 0;
}
```

*Input of different type of for loop*

input > helloworld.py > ...

```python
def func1():
    return 1
def func2():
    return "abc"
def func3():
    return 5.5
def func4():
    return 'a'

func1()
func2()
func3()
func4()
```

output.c > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include "output.h"

int func1()
{
    return 1;
}
char*  func2()
{
    return "abc";
}
float func3()
{
    return 5.5;
}
char func4()
{
    return 'a';
}
int main() {
    func1();
    func2();
    func3();
    func4();
return 0;
}

```

*Different return type*