



SRM INSTITUTE OF SCIENCE AND
TECHNOLOGY



SCHOOL OF COMPUTING

DEPARTMENT OF DATASCIENCE AND

BUSINESS SYSTEMS

18CSC304J Compiler Design

MINI PROJECT REPORT

Title The Super Tiny Compiler

NAME: AVINASH REDDY VASIPALLI

REGISTER NUMBER: RA1911027010007

MAIL ID: AV4443@SRMIST.EDU.IN

DEPARTMENT: B.TECH CSE

SPECIALIZATION: BIG DATA ANALYTICS

SEMESTER: VI

Team Members

- | | |
|----------------|-----------------|
| ➤ G.Sravan sai | RA1911027010010 |
| ➤ Sagnik Roy | RA1911027010013 |

CONTENT PAGE

1. Introduction

2. LISP

3. Phases

 a. Phase 1

 i. Parsing

 ii. Tokens Generation

 b. Phase 2

 i. Transformation

 ii. Traversal

 iii. Visitors

 c. Phase 3

 i. Code Generation

4. Screenshots

 a. Input

 b. Output

 c. Github

5. Conclusion

6. References

7. Appendix A – Source Code

8. Appendix B – GitHub Profile and Link for the Project

INTRODUCTION

This mini project is a super tiny compiler written in python language. This Compiler is so small that it is just around 200 lines without any comments and works effective in converting LISP functions call to C function call. As mentioned, it is not a total compiler for the conversion but it has major components a compiler need starting from lexical analyser to code generator.

LISP FUNCTION TO C FUNCTION

LISP is a speech impairment in which a person misarticulates sibilants. LISP is speech lang. LISP is family of programming language with long history of distinctive, fully parenthesized prefix notation.

As we know for a compiler to work tokens are the basic step and a language with parenthesis and words and number is what we use to tokenize. It is also second Oldest high level programming language which is still in use.

Examples for a LISP Input:

1. (add 2 3)
2. (subtract 4 2)
3. (add 2 (subtract 4 2))

Considering the above input, we make a compiler that converts it to C like function. C is the basic yet most powerful language in modern era. It is the basic language one can learn. C is developed keeping in mind all the oldest languages in mind. The syntax is C is precise and well compiled.

Examples of C Like Function cell

1. subtract(4, 2)
2. add(2, subtract(4, 2)) etc...

PHASES

a. Phase 1

i. Parsing

Parsing is typically analysing a input in to logical syntactic components parsing has many components in it but the major 2 that we used in this project are *Lexical analysis* and *Syntactic analysis*.

Lexical Analysis takes the raw code or input and splits it apart into tokens. This process of assigning tokens to the raw code is called tokenization or lexer.

ii. Token Generation

Tokens are an array of tiny objects that describe an isolated piece of the syntax. These pieces can be numbers, labels, punctuations, operators etc

For the following Syntax: *(add 2 (subtract 4 2))*

Tokens that are generated are –

```
[
  { type: 'paren', value: '('      },
  { type: 'name',  value: 'add'    },
  { type: 'number', value: '2'     },
  { type: 'paren', value: '('      },
  { type: 'name',  value: 'subtract' },
  { type: 'number', value: '4'     },
  { type: 'number', value: '2'     },
  { type: 'paren', value: ')'      },
  { type: 'paren', value: ')'      },
]
```

Syntactic Analysis takes the tokens we converted before and reformats them into representation that will describe each part of the syntax and their relation to one another. This process is known as intermediate representation or Abstract Syntax Tree.

Abstract Syntax Tree, or AST is deeply nested object that represent code in a way that is both easy to work and tells us information.

For the above syntax in tokenizer the AST is –

```
{
  type: 'Program',
  body: [{
    type: 'CallExpression',
    name: 'add',
    params: [{
      type: 'NumberLiteral',
      value: '2',
    }, {
      type: 'CallExpression',
      name: 'subtract',
      params: [{
        type: 'NumberLiteral',
        value: '4',
      }, {
        type: 'NumberLiteral',
        value: '2',
      }]
    }]
  }]
}
```

b. Phase 2

i. Transformation

Transformation takes this abstract representation and manipulate it to do what every compiler wants it to. It just takes AST from the last step and makes changes.

It can manipulate AST in the same language or it can translate it into an entirely new language.

In the above AST we can notice that elements within it are very similar. These are those objects with type property each of these are known as AST nodes. These nodes have defined properties on them that describe one isolated part of the tree.

We can have a node for NumberLiteral or CallExpression like shown below:

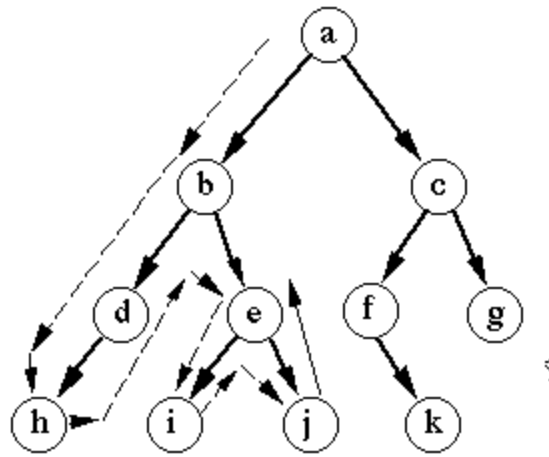
```
{
  type: 'NumberLiteral',
  value: '2'
}
and
{
  type: 'CallExpression',
  name: 'subtract',
  params: [...nested nodes go here...],
}
```

When transforming the AST, we manipulate nodes by adding, removing or replacing properties. We can leave AST alone and create an entirely new one. As we are targeting new language, we are focusing on creating entirely new AST that specific to target language.

ii. Traversal

In order to navigate through all these nodes, we need should be able to transvers through them. This transversal process goes to each node in the AST.

Depth First Search is the algorithm used for traversing or searching the tree. This algo starts at the root node and explores as far as the alone possible branch and then backtracks to the previous node



Depth-first search

The above graph represents the Depth first search where the nodes start at a and goes to the extreme end of a branch and backtracks to the start node. In this process it ensures that all the nodes are parsed.

For the above syntax the AST is-

```
{
  type: 'Program',
  body: [{
    type: 'CallExpression',
    name: 'add',
    params: [{
      type: 'NumberLiteral',
      value: '2'
    }, {
      type: 'CallExpression',
      name: 'subtract',
      params: [{
        type: 'NumberLiteral',
        value: '4'
      }, {
        type: 'NumberLiteral',
        value: '2'
      }]
    }]
  }]
}
```

Then the transversal is

1. Program - Starting at the top level of the AST
2. CallExpression (add) - Moving to the first element of the Program's body
3. NumberLiteral (2) - Moving to the first element of CallExpression's params
4. CallExpression (subtract) - Moving to the second element of CallExpression's params
5. NumberLiteral (4) - Moving to the first element of CallExpression's params
6. NumberLiteral (2) - Moving to the second element of CallExpression's params

If we manipulate this AST directly, instead of treating a separate AST, we would like to introduce all sort of abstractions. But just visiting each node in the tree is enough for this project. The reason we used visiting is because there is pattern of how to represent operations on elements of an object structure.

iii. Visitors

The basic idea of visitors is to go and create a visitor object that has methods that will accept different node types. These visitors help in making a method that will accept any and every type of node types.

```
var visitor = {  
  NumberLiteral() {},  
  CallExpression() {},  
};
```

When we traverse our AST, we will call the methods on this visitor whenever we enter a node of matching type. In order to make this useful we will also pass the node and a reference to parent node.

```
var visitor = {  
  NumberLiteral(node, parent) {},  
  CallExpression(node, parent) {},  
};
```

Then exist the possibility of calling things on EXIT. Imagine our tree structure from before in the list form:

- Program
- CallExpression
- NumberLiteral
- CallExpression
- NumberLiteral
- NumberLiteral

As we traverser down, we are going to reach branches worth dead ends. As we finish each branch of the tree, we Exit and going down the tree we enter each node, and going back up we exit. This process of enter and exit continues until the tree is traversed totally and all the nodes are parsed.,

```
var visitor = {  
  NumberLiteral: {  
    enter(node, parent) {},  
    exit(node, parent) {},  
  }  
};
```

c. Phase 3

i. Code Generation

The final phase of a compiler is code generation. Sometimes compilers will do things that overlap with transformation, but for the most part code generation just means take our ASI and stringify code back out.

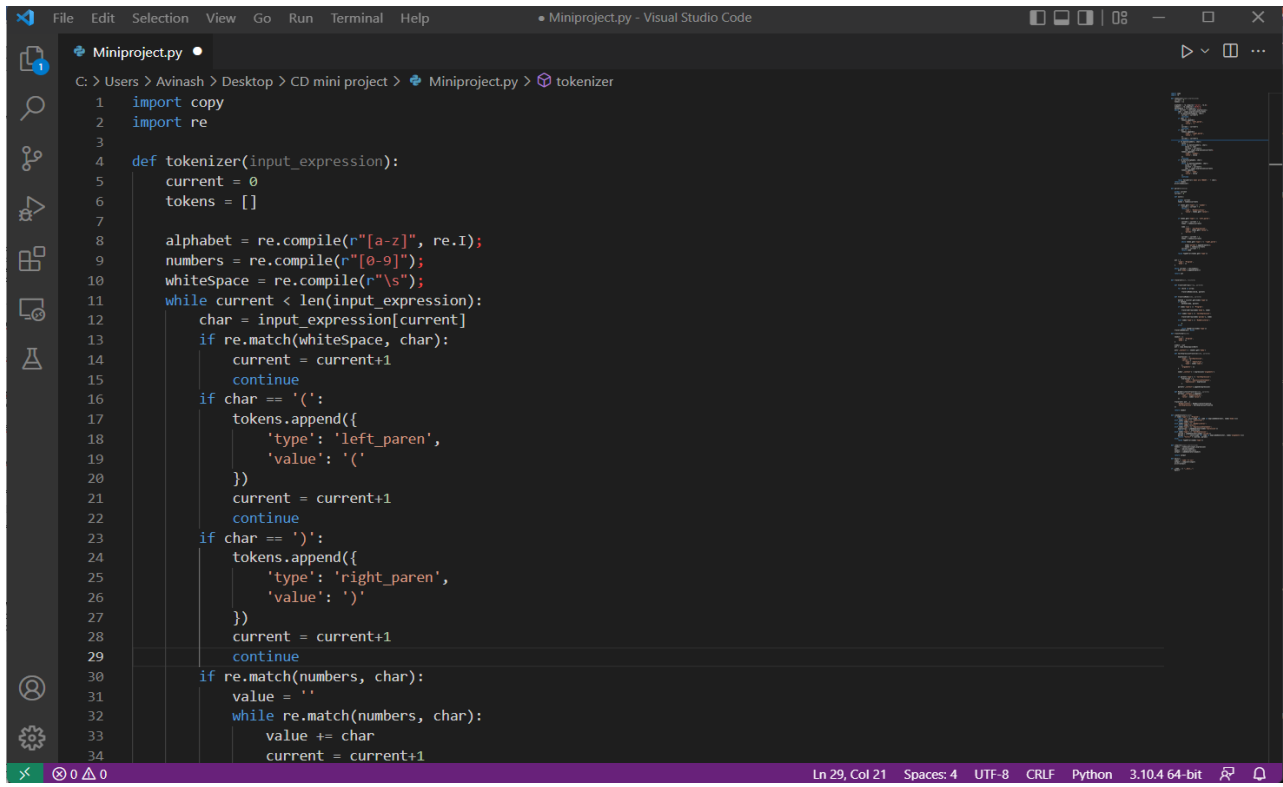
Code generation works in different ways some compiler will resume the tokens that are generated earlier whereas others will create a separate representation of the code so that they can print nodes linearly. But most methods use same AST we just created which is better.

The code generator we used effectively knows how to print all the different nodes and types of ASI. This will recursively call itself to print nested nodes until everything is printed into one long string of code.

These are different pieces of a compiler and this is not how every compiler works as they have different purposes, and they may need more steps. But now that we know general high-level idea on compiler you can understand how a compiler works in theory.

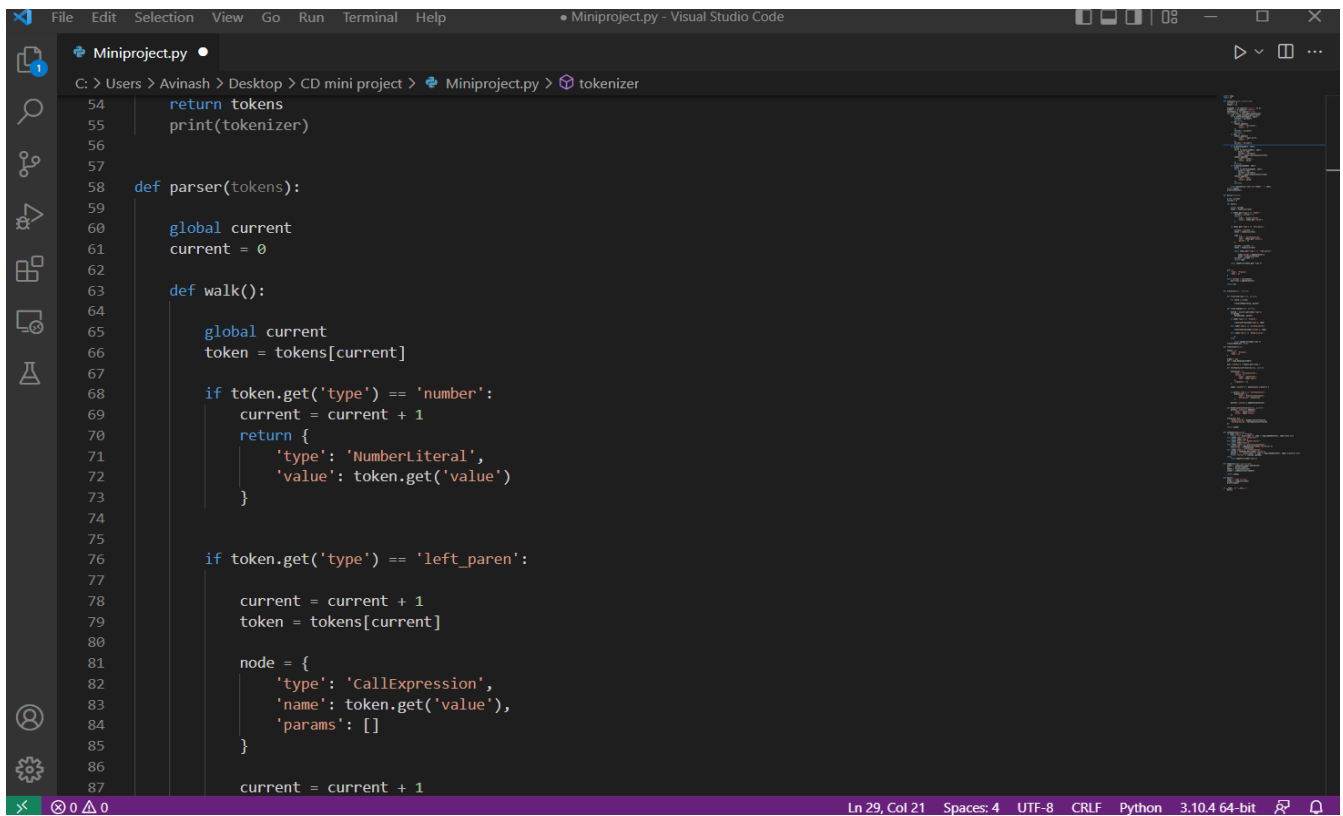
SCREENSHOTS

i. Input



The screenshot shows the Visual Studio Code editor with the file `Miniproject.py` open. The editor is displaying the `tokenizer` function, which is responsible for tokenizing an input expression. The function uses regular expressions to identify whitespace, parentheses, and numbers. The code is as follows:

```
1 import copy
2 import re
3
4 def tokenizer(input_expression):
5     current = 0
6     tokens = []
7
8     alphabet = re.compile(r"[a-z]", re.I);
9     numbers = re.compile(r"[0-9]");
10    whitespace = re.compile(r"\s");
11    while current < len(input_expression):
12        char = input_expression[current]
13        if re.match(whitespace, char):
14            current = current+1
15            continue
16        if char == '(':
17            tokens.append({
18                'type': 'left_paren',
19                'value': '('
20            })
21            current = current+1
22            continue
23        if char == ')':
24            tokens.append({
25                'type': 'right_paren',
26                'value': ')'
27            })
28            current = current+1
29            continue
30        if re.match(numbers, char):
31            value = ''
32            while re.match(numbers, char):
33                value += char
34                current = current+1
```

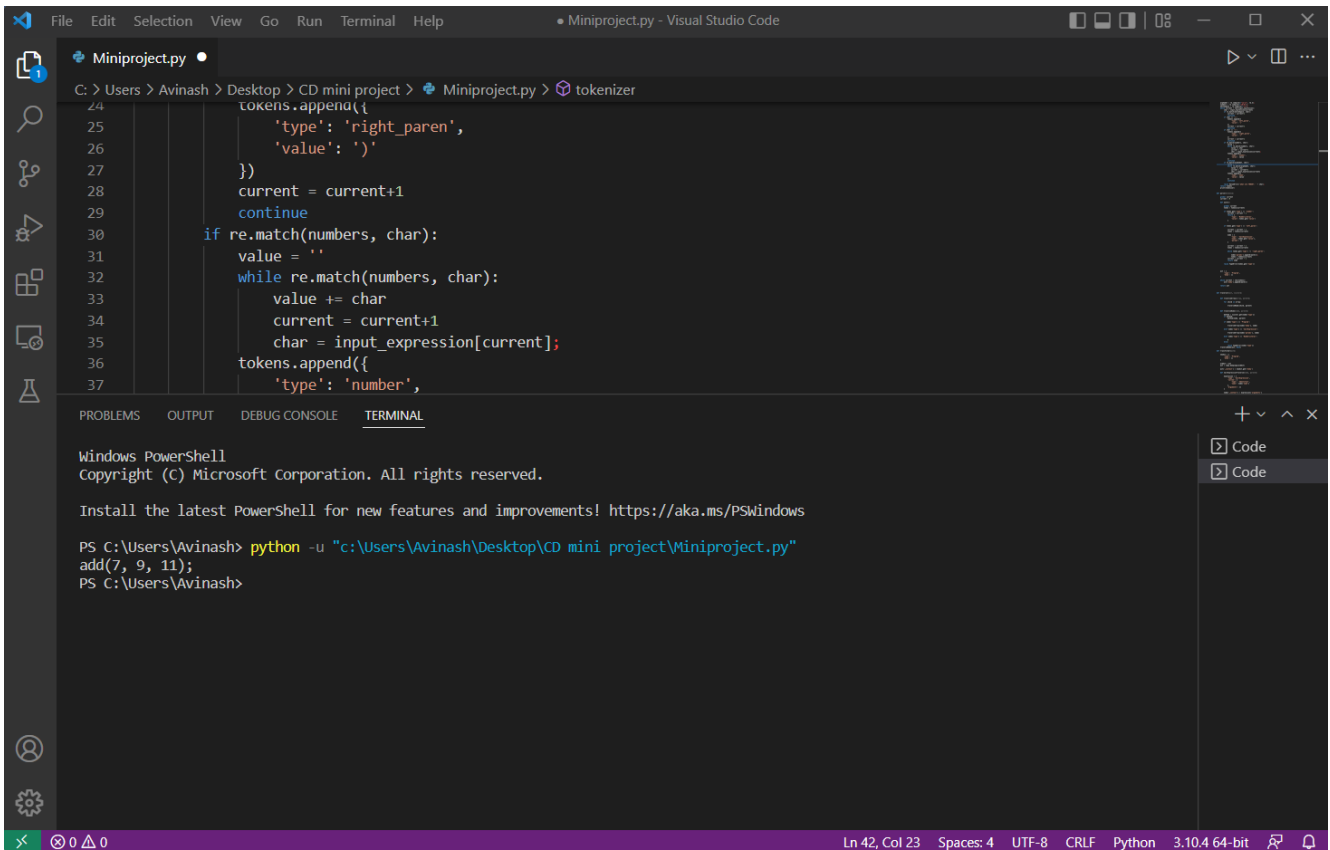


The screenshot shows the Visual Studio Code editor with the file `Miniproject.py` open. The editor is displaying the `parser` function, which is responsible for parsing the tokens generated by the `tokenizer` function. The function uses a global `current` variable to track the current position in the tokens list. The code is as follows:

```
54 return tokens
55 print(tokenizer)
56
57
58 def parser(tokens):
59     global current
60     current = 0
61
62     def walk():
63         global current
64         token = tokens[current]
65
66         if token.get('type') == 'number':
67             current = current + 1
68             return {
69                 'type': 'NumberLiteral',
70                 'value': token.get('value')
71             }
72
73         if token.get('type') == 'left_paren':
74
75             current = current + 1
76             token = tokens[current]
77
78             node = {
79                 'type': 'CallExpression',
80                 'name': token.get('value'),
81                 'params': []
82             }
83
84             current = current + 1
```

ii. Output

✓ Add(7 9 11)



```
File Edit Selection View Go Run Terminal Help • Miniproject.py - Visual Studio Code
Miniproject.py
C: > Users > Avinash > Desktop > CD mini project > Miniproject.py > tokenizer
24 tokens.append({
25     'type': 'right_paren',
26     'value': ')'
27 })
28 current = current+1
29 continue
30 if re.match(numbers, char):
31     value = ''
32     while re.match(numbers, char):
33         value += char
34         current = current+1
35         char = input_expression[current];
36     tokens.append({
37         'type': 'number',
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

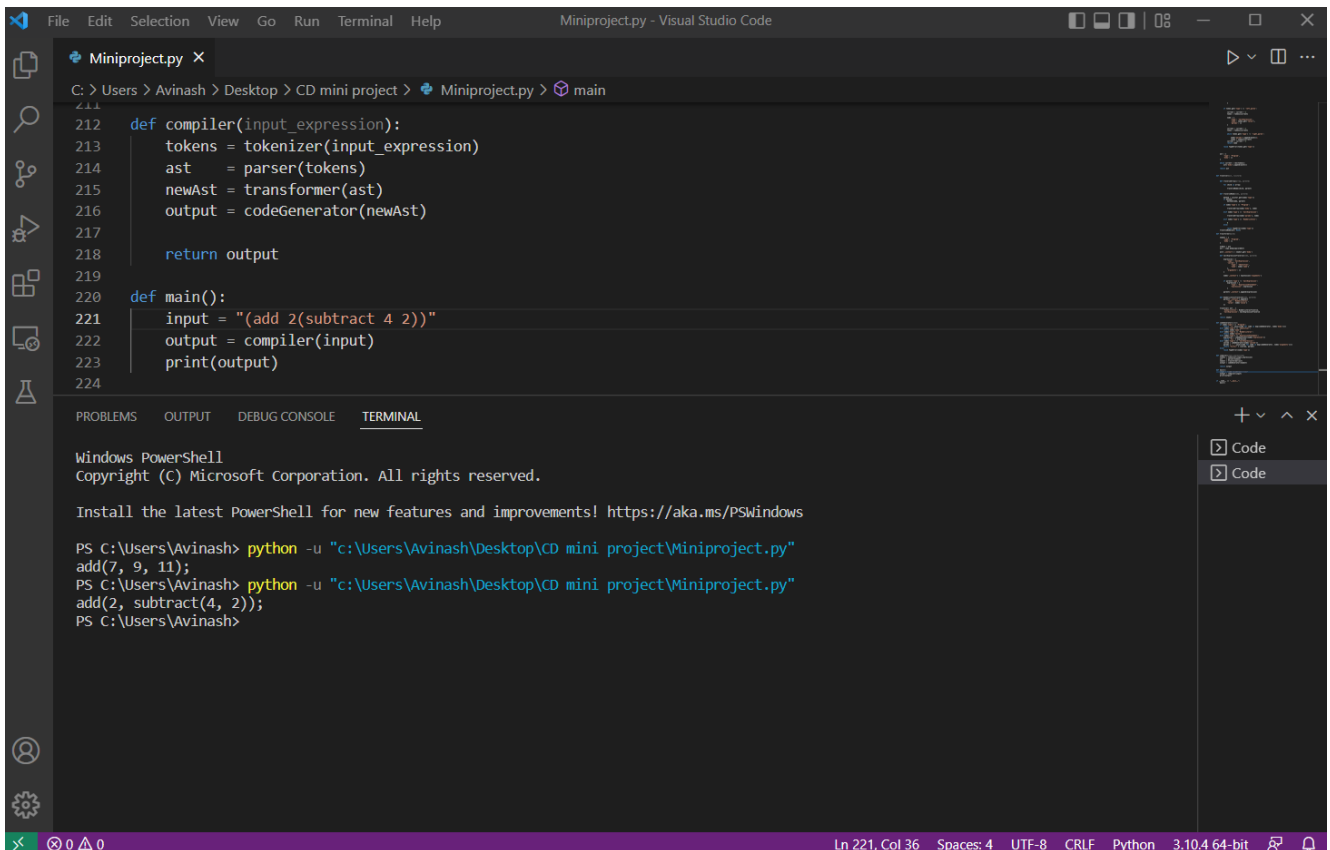
Windows PowerShell
Copyright (c) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! <https://aka.ms/PSWindows>

PS C:\Users\Avinash> python -u "c:\Users\Avinash\Desktop\CD mini project\Miniproject.py"
add(7, 9, 11);
PS C:\Users\Avinash>

Ln 42, Col 23 Spaces: 4 UTF-8 CRLF Python 3.10.4 64-bit

✓ (add 2(subtract 4 2))



```
File Edit Selection View Go Run Terminal Help Miniproject.py - Visual Studio Code
Miniproject.py X
C: > Users > Avinash > Desktop > CD mini project > Miniproject.py > main
212 def compiler(input_expression):
213     tokens = tokenizer(input_expression)
214     ast = parser(tokens)
215     newAst = transformer(ast)
216     output = codeGenerator(newAst)
217
218     return output
219
220 def main():
221     input = "(add 2(subtract 4 2))"
222     output = compiler(input)
223     print(output)
224
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

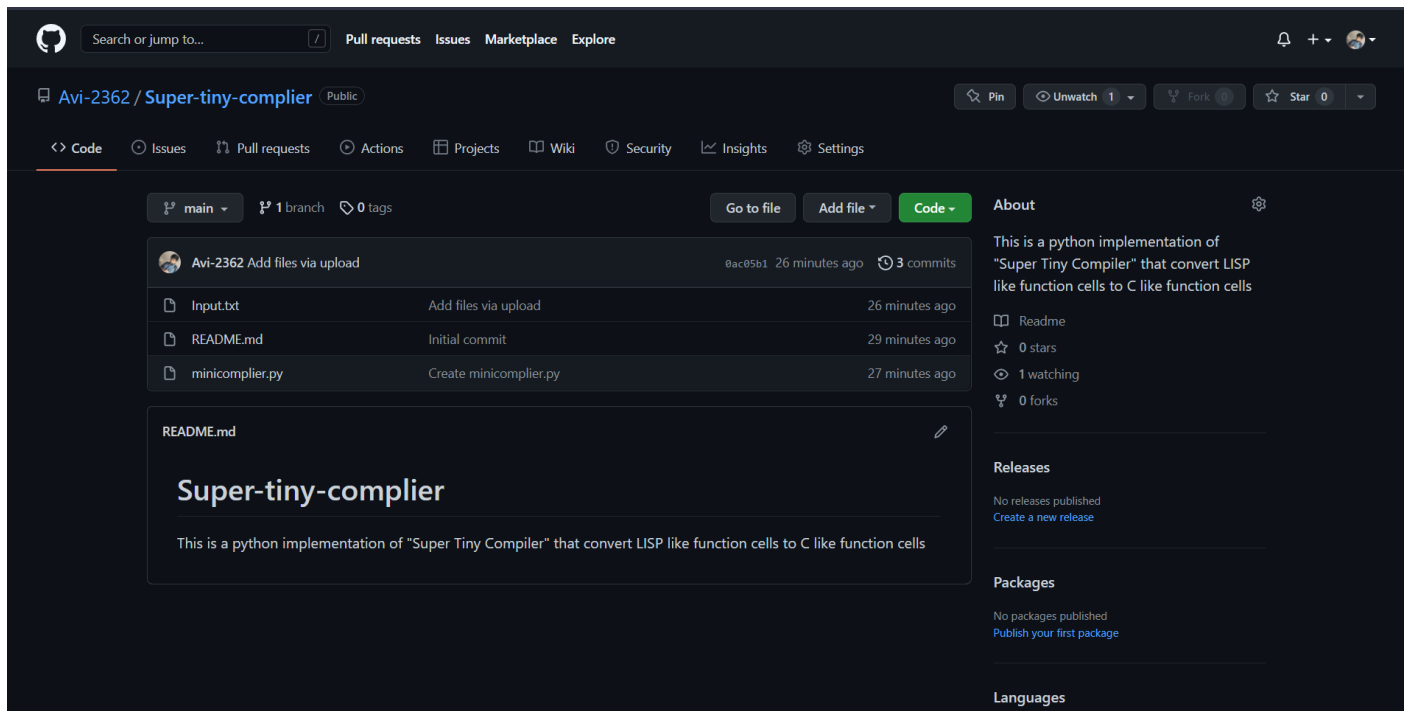
Windows PowerShell
Copyright (c) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! <https://aka.ms/PSWindows>

PS C:\Users\Avinash> python -u "c:\Users\Avinash\Desktop\CD mini project\Miniproject.py"
add(7, 9, 11);
PS C:\Users\Avinash> python -u "c:\Users\Avinash\Desktop\CD mini project\Miniproject.py"
add(2, subtract(4, 2));
PS C:\Users\Avinash>

Ln 221, Col 36 Spaces: 4 UTF-8 CRLF Python 3.10.4 64-bit

iii. Github



CONCLUSION

As shown in about the source code has all the components need for a compiler and the implementation is done which takes the input format in LISP and gives out C like function cell in which the input is tokenized, parsed, traversed and output is printed. Every step in the process is considered from lexer to DFS. The output is printed which is in the format c function.

REFERENCES

- Stackoverflow for the components of a compiler
- GeeksforGeeks for the theory part of Compiler
- GitHub of JAMIEBUILDS
 - <https://github.com/jamiebuilds/the-super-tiny-compiler>
- Textbook references of compiler design

APPENDIX A – SOURCE CODE

```
import copy
import re

def tokenizer(input_expression):
    current = 0
    tokens = []

    alphabet = re.compile(r"[a-z]", re.I);
    numbers = re.compile(r"[0-9]");
    whiteSpace = re.compile(r"\s");
    while current < len(input_expression):
        char = input_expression[current]
        if re.match(whiteSpace, char):
            current = current+1
            continue
        if char == '(':
            tokens.append({
                'type': 'left_paren',
                'value': '('
            })
            current = current+1
            continue
        if char == ')':
            tokens.append({
                'type': 'right_paren',
                'value': ')'
            })
            current = current+1
            continue
        if re.match(numbers, char):
            value = ""
            while re.match(numbers, char):
                value += char
                current = current+1
            char = input_expression[current];
            tokens.append({
                'type': 'number',
                'value': value
            })
    })
```

```

    continue
if re.match(alphabet, char):
    value = ""
    while re.match(alphabet, char):
        value += char
        current = current+1
        char = input_expression[current]
    tokens.append({
        'type': 'name',
        'value': value
    })
    continue

    raise ValueError('what are THOSE?: ' + char);
return tokens
print(tokenizer)

```

```
def parser(tokens):
```

```

    global current
    current = 0

```

```
def walk():
```

```

    global current
    token = tokens[current]

```

```

if token.get('type') == 'number':
    current = current + 1
    return {
        'type': 'NumberLiteral',
        'value': token.get('value')
    }

```

```
if token.get('type') == 'left_paren':
```

```

    current = current + 1
    token = tokens[current]

```

```

    node = {
        'type': 'CallExpression',

```

```
    'name': token.get('value'),  
    'params': []  
}
```

```
current = current + 1  
token = tokens[current]
```

```
while token.get('type') != 'right_paren':
```

```
    node['params'].append(walk());  
    token = tokens[current]  
current = current + 1  
return node
```

```
raise TypeError(token.get('type'))
```

```
ast = {  
    'type': 'Program',  
    'body': []  
}
```

```
while current < len(tokens):  
    ast['body'].append(walk())
```

```
return ast
```

```
def traverser(ast, visitor):
```

```
    def traverseArray(array, parent):
```

```
        for child in array:
```

```
            traverseNode(child, parent)
```

```
    def traverseNode(node, parent):
```

```
        method = visitor.get(node['type'])
```

```

if method:
    method(node, parent)

if node['type'] == 'Program':

    traverseArray(node['body'], node)

elif node['type'] == 'CallExpression':

    traverseArray(node['params'], node)

elif node['type'] == 'NumberLiteral':

    0
else:

    raise TypeError(node['type'])
traverseNode(ast, None)

def transformer(ast):

    newAst = {
        'type': 'Program',
        'body': []
    }

    oldAst = ast
    ast = copy.deepcopy(oldAst)

    ast['_context'] = newAst.get('body')

    def CallExpressionTraverse(node, parent):

        expression = {
            'type': 'CallExpression',
            'callee': {
                'type': 'Identifier',
                'name': node['name']
            },
            'arguments': []
        }

        node['_context'] = expression['arguments']

```



```

if parent['type'] != 'CallExpression':
    expression = {
        'type': 'ExpressionStatement',
        'expression': expression
    }

parent['_context'].append(expression)

def NumberLiteralTraverse(node, parent):
    parent['_context'].append({
        'type': 'NumberLiteral',
        'value': node['value']
    })

traverser( ast , {
    'NumberLiteral': NumberLiteralTraverse,
    'CallExpression': CallExpressionTraverse
})

return newAst

def codeGenerator(node):
    if node['type'] == 'Program':
        return '\n'.join([code for code in map(codeGenerator, node['body'])])
    elif node['type'] == 'Identifier':
        return node['name']
    elif node['type'] == 'NumberLiteral':
        return node['value']
    elif node['type'] == 'ExpressionStatement':
        expression = codeGenerator(node['expression'])
        return '%s;' % expression
    elif node['type'] == 'CallExpression':
        callee = codeGenerator(node['callee'])
        params = ', '.join([code for code in map(codeGenerator, node['arguments'])])
        return "%s(%s)" % (callee, params)
    else:
        raise TypeError(node['type'])

```

```
def compiler(input_expression):
    tokens = tokenizer(input_expression)
    ast = parser(tokens)
    newAst = transformer(ast)
    output = codeGenerator(newAst)

    return output

def main():
    input = "(add 2(subtract 4 2))"
    output = compiler(input)
    print(output)

if __name__ == "__main__":
    main()
```

APPENDIX B – GITHUB PROFILE AND LINK FOR THE PROJECT

GitHub Profile - <https://github.com/Avi-2362>

Project link - <https://github.com/Avi-2362/Super-tiny-complier>