CSE 439 Design of Compilers

PROJECT DOCUMENTATION

Submitted to:

Prof. Dr. Eslam Amer

Eng. Hazem / Eng. Ahmed

Submitted by:

Youssef George Fouad	19P9824
Mostafa Nasrat Metwally	19P4619
Kerollos Wageeh Youssef	19P3468
Anthony Amgad Fayek	19P9880

0. Introduction

The program is written in Python using PyQt5, Qt Web Engine, PyQt5 Tools and Pyvis libraries.

The project source code can also be found on GitHub - youssefg7/CSE439Team3Spring22

Before testing the source code please make sure you have python installed as well as the required dependencies.

Here's a link to install python Welcome to Python.org

Here's a list of cmd commands to install all required dependencies after installing python:

- `pip install PyQt5`
- `pip install PyQtWebEngine`
- `pip install pyqt5-tools`
- `pip install pyvis`

Table of Contents

1. Quick Explanation

Our project Team was Assigned Case 3:

- If statement
- Used tokens (if,then,end,else,ID,NUM,:=, ;)
- Example

```
If 1 then
        x := 5;
        y := x;
end
```

- Note the condition part is only a number (0 or 1 or 2 …. etc)
- Be aware of nested IFs , to clear ambiguity each else is connected to it’s closest if
- Required (RegExp and DFA) for the if statement and token list
- SLR(1) parser

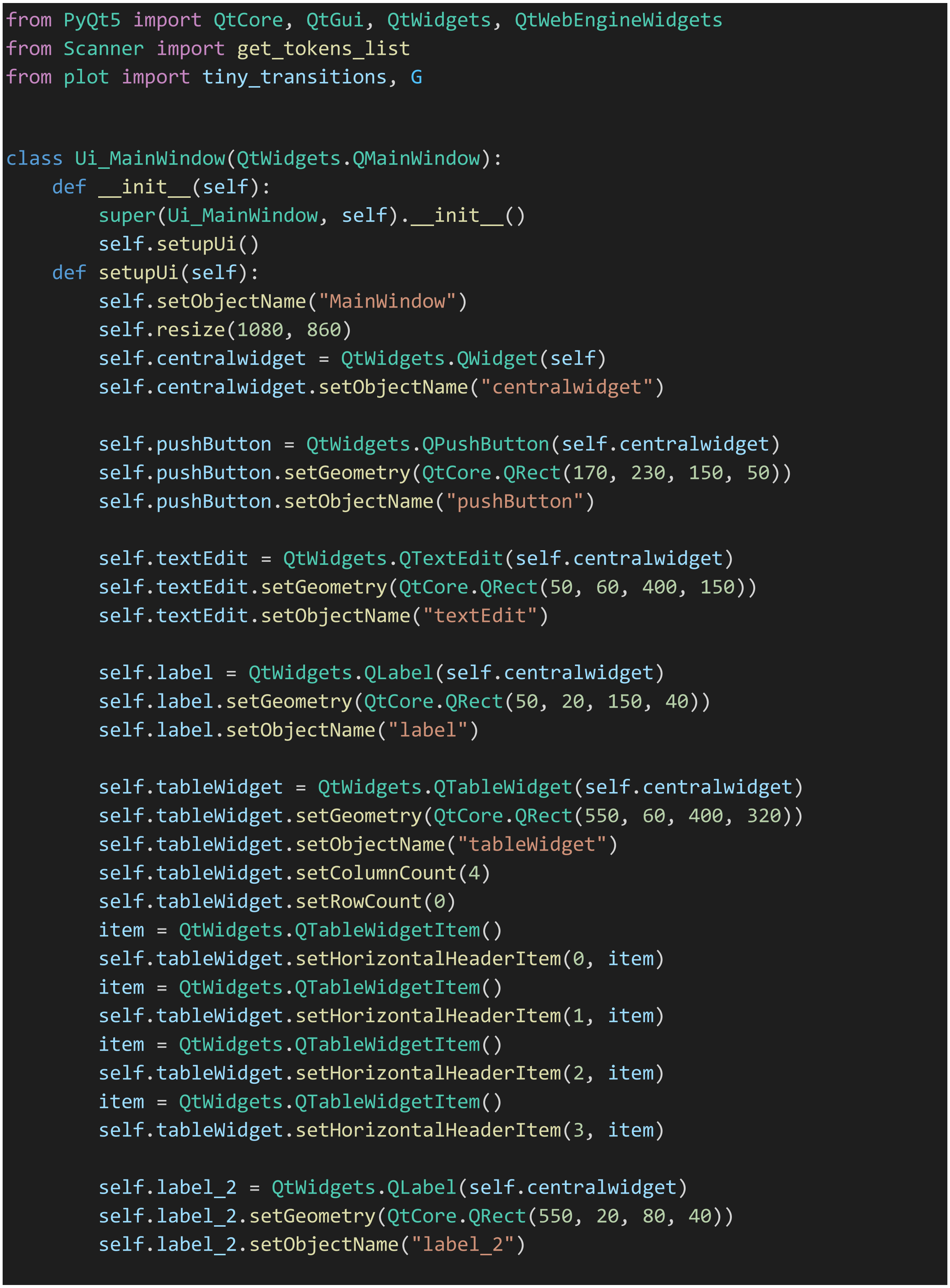
2. Code

2.1. MainUI.py

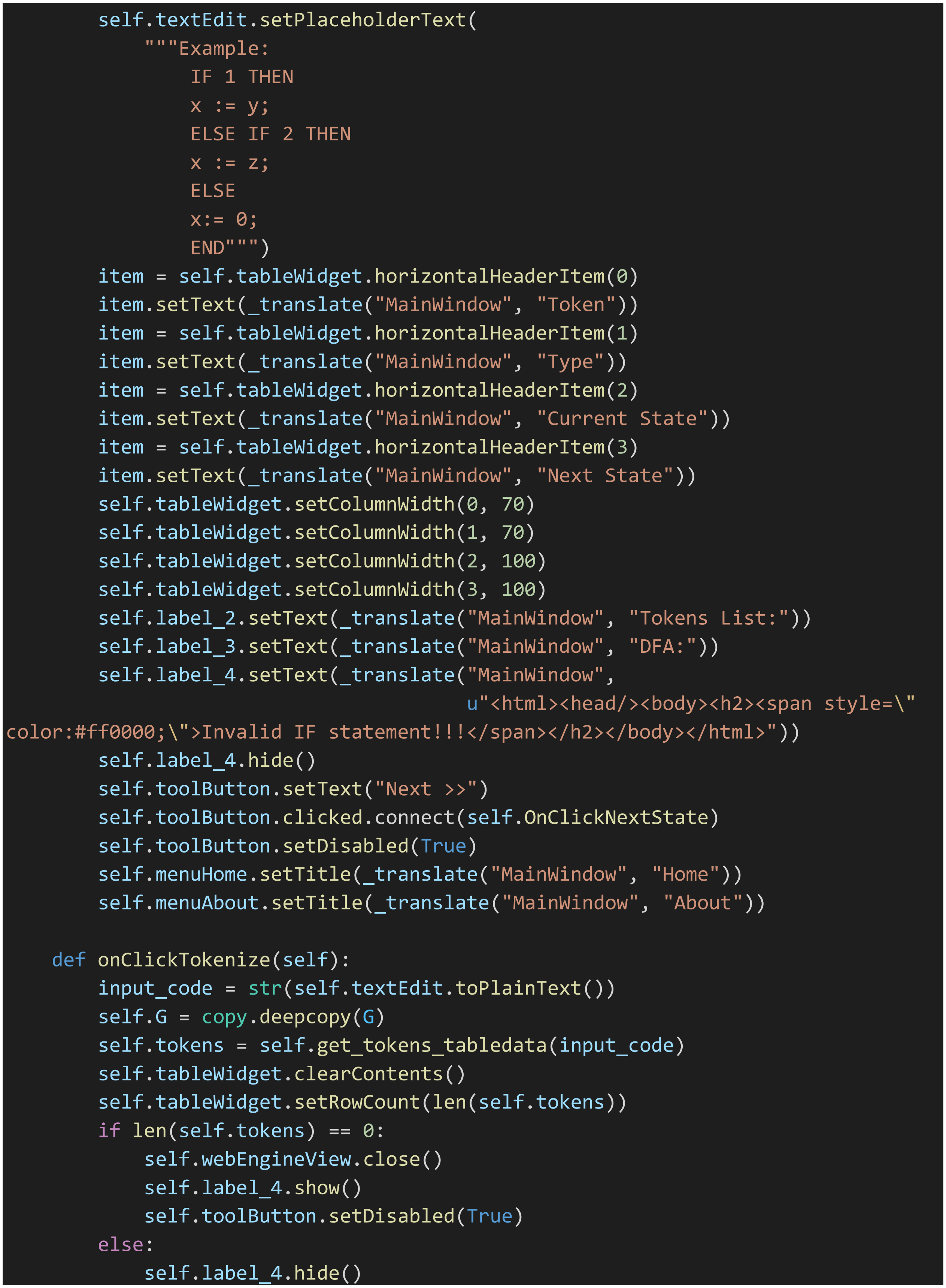
```
from PyQt5 import QtWidgets, uic, QtGui, QtCore
import sys
from AnimatedGUI import Ui_MainWindow
from ParserUI import ParserUi

class MainUi(QtWidgets.QMainWindow):
    def __init__(self):
        super(MainUi,self).__init__()
        uic.loadUi('ui\MainGUI.ui',self)
        #self.setFixedSize(804, 156)

        self.parserPushButton.clicked.connect(self.onClickParser)
        self.scannerPushButton.clicked.connect(self.onClickScanner)
        #self.show()

    def onClickParser(self):
        parserUi = ParserUi()
        stackedWidget.addWidget(parserUi)
        stackedWidget.setGeometry(QtCore.QRect(500,50,1000,1000))
        stackedWidget.setCurrentIndex(stackedWidget.currentIndex() + 1)

    def onClickScanner(self):
        scannerUi = Ui_MainWindow()
        stackedWidget.addWidget(scannerUi)
        stackedWidget.setGeometry(QtCore.QRect(500,50,1080,1000))
        stackedWidget.setCurrentIndex(stackedWidget.currentIndex() + 1)
if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    mainUi = MainUi()
    stackedWidget = QtWidgets.QStackedWidget()
    stackedWidget.addWidget(mainUi)
    stackedWidget.setGeometry(QtCore.QRect(500,200,820,620))
    stackedWidget.show()
    app.exec_()
```

2.2. AnimatedGUI.py

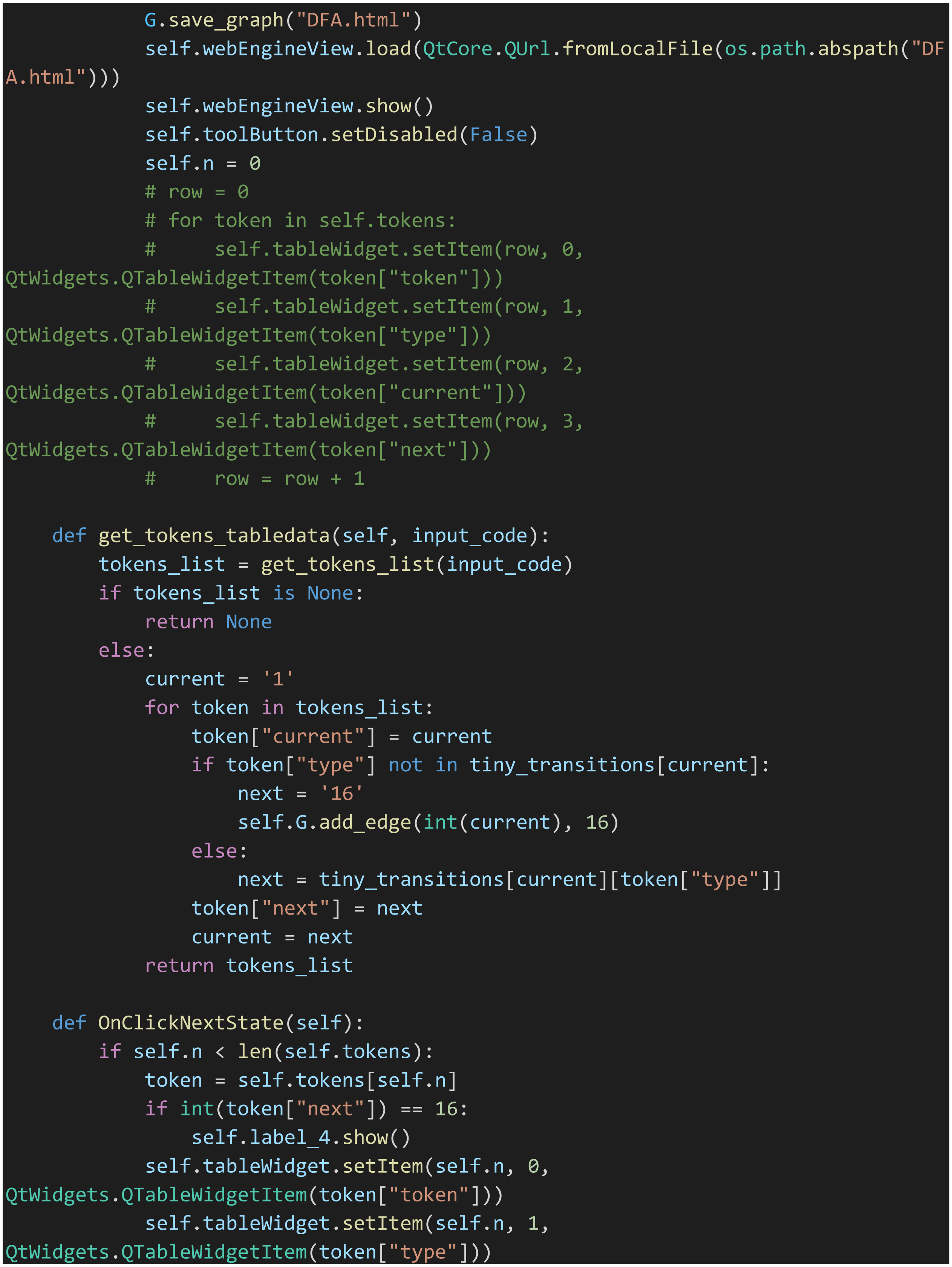
```
import copy
import os.path
```



```python
from PyQt5 import QtCore, QtGui, QtWidgets, QtWebEngineWidgets
from Scanner import get_tokens_list
from plot import tiny_transitions, G


class Ui_MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super(Ui_MainWindow, self).__init__()
        self.setupUi()
    def setupUi(self):
        self.setObjectName("MainWindow")
        self.resize(1080, 860)
        self.centralwidget = QtWidgets.QWidget(self)
        self.centralwidget.setObjectName("centralwidget")

        self.pushButton = QtWidgets.QPushButton(self.centralwidget)
        self.pushButton.setGeometry(QtCore.QRect(170, 230, 150, 50))
        self.pushButton.setObjectName("pushButton")

        self.textEdit = QtWidgets.QTextEdit(self.centralwidget)
        self.textEdit.setGeometry(QtCore.QRect(50, 60, 400, 150))
        self.textEdit.setObjectName("textEdit")

        self.label = QtWidgets.QLabel(self.centralwidget)
        self.label.setGeometry(QtCore.QRect(50, 20, 150, 40))
        self.label.setObjectName("label")

        self.tableWidget = QtWidgets.QTableWidget(self.centralwidget)
        self.tableWidget.setGeometry(QtCore.QRect(550, 60, 400, 320))
        self.tableWidget.setObjectName("tableWidget")
        self.tableWidget.setColumnCount(4)
        self.tableWidget.setRowCount(0)
        item = QtWidgets.QTableWidgetItem()
        self.tableWidget.setHorizontalHeaderItem(0, item)
        item = QtWidgets.QTableWidgetItem()
        self.tableWidget.setHorizontalHeaderItem(1, item)
        item = QtWidgets.QTableWidgetItem()
        self.tableWidget.setHorizontalHeaderItem(2, item)
        item = QtWidgets.QTableWidgetItem()
        self.tableWidget.setHorizontalHeaderItem(3, item)

        self.label_2 = QtWidgets.QLabel(self.centralwidget)
        self.label_2.setGeometry(QtCore.QRect(550, 20, 80, 40))
        self.label_2.setObjectName("label_2")
        self.label_3 = QtWidgets.QLabel(self.centralwidget)
        self.label_3.setGeometry(QtCore.QRect(40, 360, 60, 40))
        self.label_3.setObjectName("label_3")

        self.label_4 = QtWidgets.QLabel(self.centralwidget)
        self.label_4.setGeometry(QtCore.QRect(140, 300, 260, 40))
        self.label_4.setObjectName("label_4")
        self.webEngineView =
QtWebEngineWidgets.QWebEngineView(self.centralwidget)

        self.webEngineView.setGeometry(QtCore.QRect(30, 400, 1000, 400))
        self.webEngineView.setObjectName("webEngineView")

        self.toolButton = QtWidgets.QToolButton(self.centralwidget)
        self.toolButton.setGeometry(40, 410, 80, 40)
        self.toolButton.setObjectName("toolButton")

        self.setCentralWidget(self.centralwidget)

        self.menubar = QtWidgets.QMenuBar(self)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 1080, 26))
        self.menubar.setObjectName("menubar")
        self.menuHome = QtWidgets.QMenu(self.menubar)
        self.menuHome.setObjectName("menuHome")
        self.menuAbout = QtWidgets.QMenu(self.menubar)
        self.menuAbout.setObjectName("menuAbout")
        self.setMenuBar(self.menubar)

        self.statusbar = QtWidgets.QStatusBar(self)
        self.statusbar.setObjectName("statusbar")
        self.setStatusBar(self.statusbar)
        self.menubar.addAction(self.menuHome.menuAction())
        self.menubar.addAction(self.menuAbout.menuAction())

        self.retranslateUi(self)
        QtCore.QMetaObject.connectSlotsByName(self)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "TINY Language
Compiler"))
        self.pushButton.setText(_translate("MainWindow", "Tokenize Code"))
        self.pushButton.clicked.connect(self.onClickTokenize)
        self.label.setText(_translate("MainWindow", "Insert your code here:"))
```



```
        self.textEdit.setPlaceholderText(
            """Example:
                IF 1 THEN
                x := y;
                ELSE IF 2 THEN
                x := z;
                ELSE
                x:= 0;
                END""")
        item = self.tableWidget.horizontalHeaderItem(0)
        item.setText(_translate("MainWindow", "Token"))
        item = self.tableWidget.horizontalHeaderItem(1)
        item.setText(_translate("MainWindow", "Type"))
        item = self.tableWidget.horizontalHeaderItem(2)
        item.setText(_translate("MainWindow", "Current State"))
        item = self.tableWidget.horizontalHeaderItem(3)
        item.setText(_translate("MainWindow", "Next State"))
        self.tableWidget.setColumnWidth(0, 70)
        self.tableWidget.setColumnWidth(1, 70)
        self.tableWidget.setColumnWidth(2, 100)
        self.tableWidget.setColumnWidth(3, 100)
        self.label_2.setText(_translate("MainWindow", "Tokens List:"))
        self.label_3.setText(_translate("MainWindow", "DFA:"))
        self.label_4.setText(_translate("MainWindow",
                                        u"<html><head/><body><h2><span style=\"
color:#ff0000;\">Invalid IF statement!!!</span></h2></body></html>"))
        self.label_4.hide()
        self.toolButton.setText("Next >>")
        self.toolButton.clicked.connect(self.OnClickNextState)
        self.toolButton.setDisabled(True)
        self.menuHome.setTitle(_translate("MainWindow", "Home"))
        self.menuAbout.setTitle(_translate("MainWindow", "About"))

    def onClickTokenize(self):
        input_code = str(self.textEdit.toPlainText())
        self.G = copy.deepcopy(G)
        self.tokens = self.get_tokens_tabledata(input_code)
        self.tableWidget.clearContents()
        self.tableWidget.setRowCount(len(self.tokens))
        if len(self.tokens) == 0:
            self.webEngineView.close()
            self.label_4.show()
            self.toolButton.setDisabled(True)
        else:
            self.label_4.hide()
```

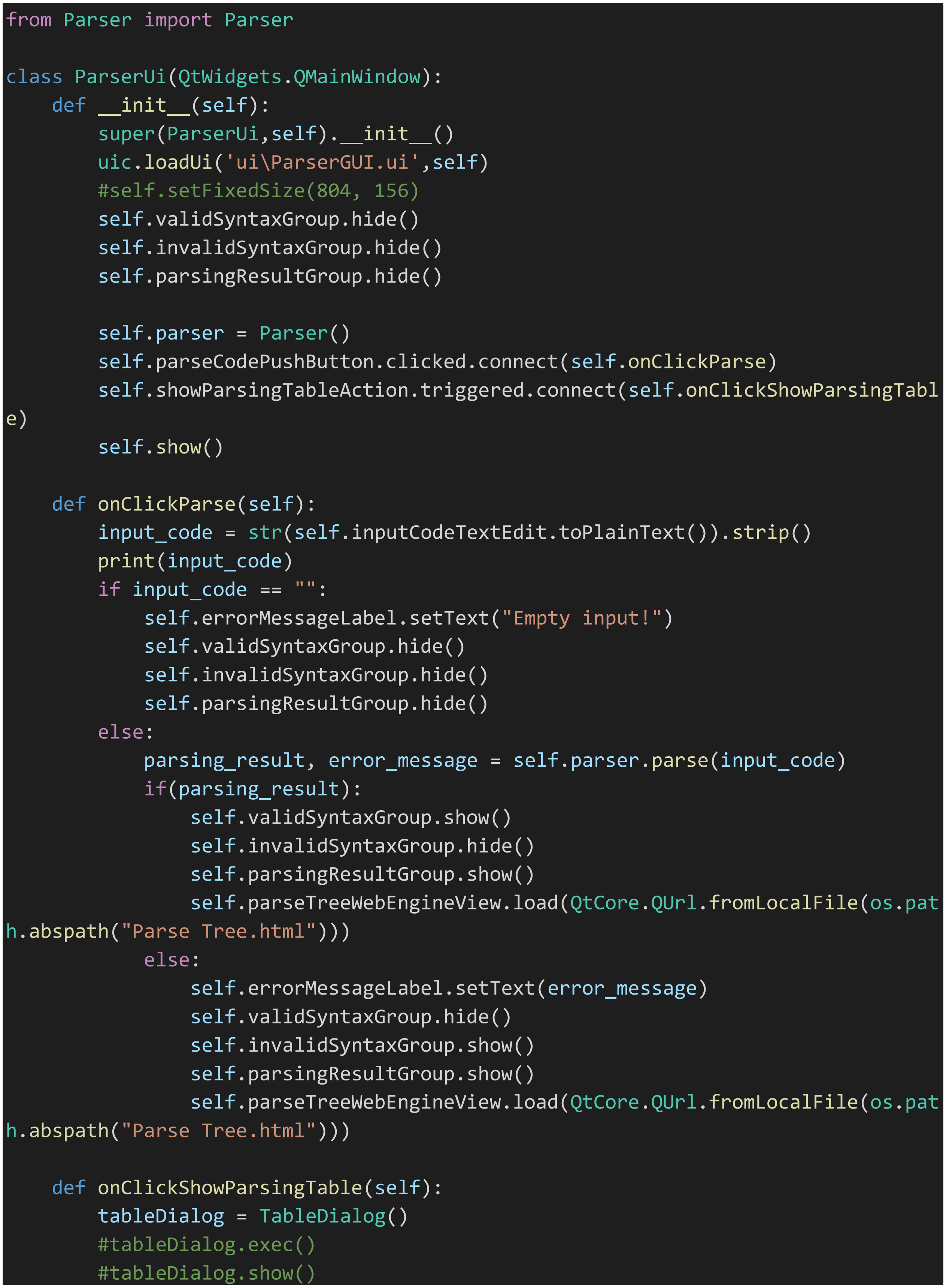


```
            G.save_graph("DFA.html")
            self.webEngineView.load(QtCore.QUrl.fromLocalFile(os.path.abspath("DF
A.html")))
            self.webEngineView.show()
            self.toolButton.setDisabled(False)
            self.n = 0
            # row = 0
            # for token in self.tokens:
            #     self.tableWidget.setItem(row, 0,
QtWidgets.QTableWidgetItem(token["token"]))
            #     self.tableWidget.setItem(row, 1,
QtWidgets.QTableWidgetItem(token["type"]))
            #     self.tableWidget.setItem(row, 2,
QtWidgets.QTableWidgetItem(token["current"]))
            #     self.tableWidget.setItem(row, 3,
QtWidgets.QTableWidgetItem(token["next"]))
            #     row = row + 1

    def get_tokens_tabledata(self, input_code):
        tokens_list = get_tokens_list(input_code)
        if tokens_list is None:
            return None
        else:
            current = '1'
            for token in tokens_list:
                token["current"] = current
                if token["type"] not in tiny_transitions[current]:
                    next = '16'
                    self.G.add_edge(int(current), 16)
                else:
                    next = tiny_transitions[current][token["type"]]
                token["next"] = next
                current = next
            return tokens_list

    def OnClickNextState(self):
        if self.n < len(self.tokens):
            token = self.tokens[self.n]
            if int(token["next"]) == 16:
                self.label_4.show()
            self.tableWidget.setItem(self.n, 0,
QtWidgets.QTableWidgetItem(token["token"]))
            self.tableWidget.setItem(self.n, 1,
QtWidgets.QTableWidgetItem(token["type"]))
```

```
            self.tableWidget.setItem(self.n, 2, 
QtWidgets.QTableWidgetItem(token["current"]))
            self.tableWidget.setItem(self.n, 3, 
QtWidgets.QTableWidgetItem(token["next"]))
            self.G.nodes[int(token["current"]) - 1]["color"] = 'lime'
            self.G.nodes[int(token["next"]) - 1]["color"] = {"background": 
'yellow', "border": 'red'}
            self.G.save_graph("DFA.html")
            self.redisplayDFA()
        else:
            self.G.nodes[int(self.tokens[-1]["next"]) - 1]["color"] = 
{"background": 'lime', "border": 'blue'}
            self.G.save_graph("DFA.html")
            self.redisplayDFA()
            self.toolButton.setDisabled(True)
            #self.toolButton.setText("Repeat?")
            #self.n = 0
        self.n = self.n + 1

    def redisplayDFA(self):
        self.webEngineView.close()
        self.webEngineView.load(QtCore.QUrl.fromLocalFile(os.path.abspath("DFA.ht
ml")))
        self.webEngineView.show()


if __name__ == "__main__":
    import sys

    app = QtWidgets.QApplication(sys.argv)
    #MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.show()
    #ui.setupUi(MainWindow)
    #MainWindow.show()
    sys.exit(app.exec_())
```

2.3. ParserUI.py

```
from PyQt5 import QtWidgets, uic, QtGui, QtWebEngineWidgets, QtCore
import sys
import os.path
```

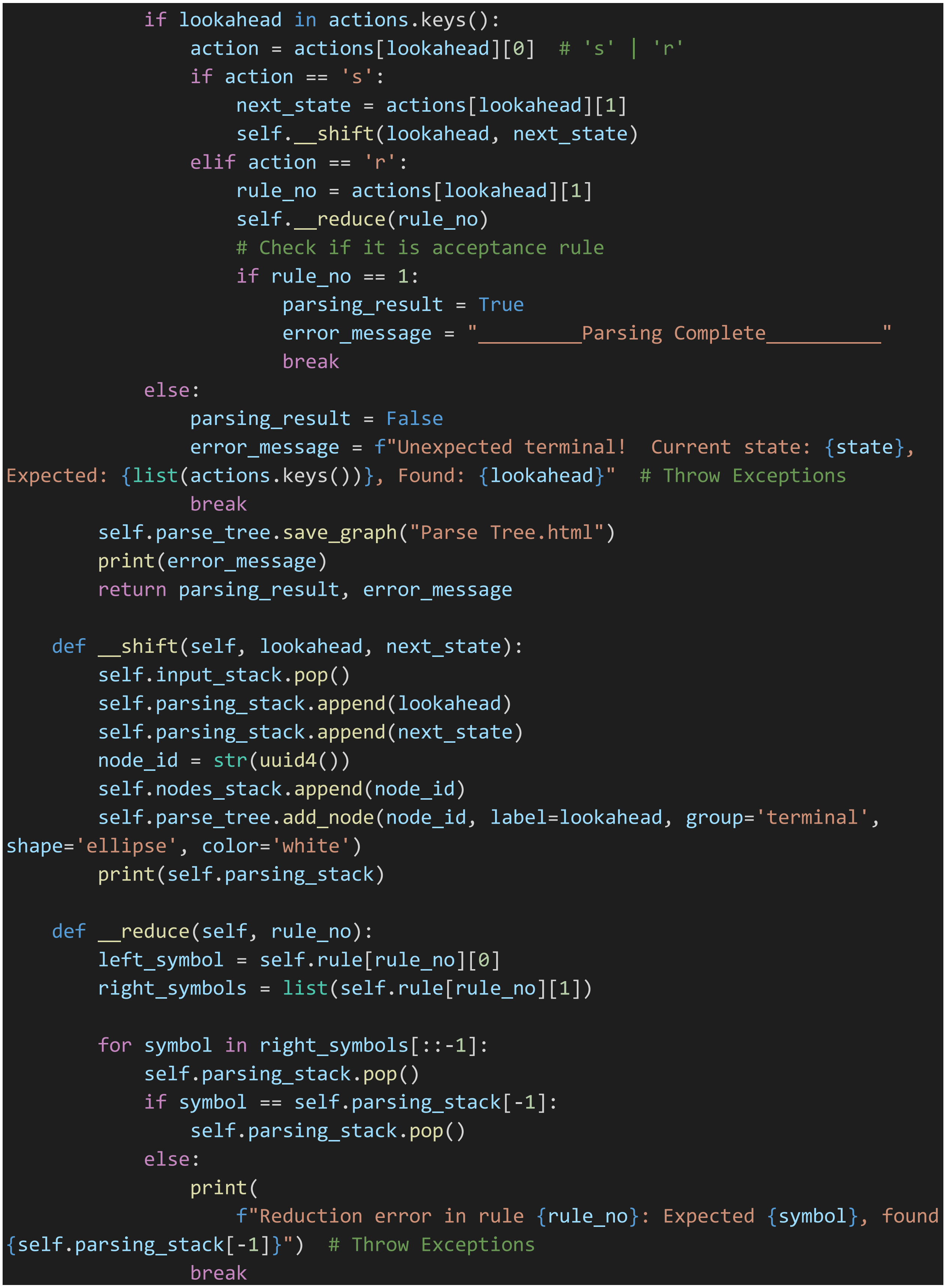


```
from Parser import Parser

class ParserUi(QtWidgets.QMainWindow):
    def __init__(self):
        super(ParserUi,self).__init__()
        uic.loadUi('ui\ParserGUI.ui',self)
        #self.setFixedSize(804, 156)
        self.validSyntaxGroup.hide()
        self.invalidSyntaxGroup.hide()
        self.parsingResultGroup.hide()

        self.parser = Parser()
        self.parseCodePushButton.clicked.connect(self.onClickParse)
        self.showParsingTableAction.triggered.connect(self.onClickShowParsingTabl
e)
        self.show()

    def onClickParse(self):
        input_code = str(self.inputCodeTextEdit.toPlainText()).strip()
        print(input_code)
        if input_code == "":
            self.errorMessageLabel.setText("Empty input!")
            self.validSyntaxGroup.hide()
            self.invalidSyntaxGroup.hide()
            self.parsingResultGroup.hide()
        else:
            parsing_result, error_message = self.parser.parse(input_code)
            if(parsing_result):
                self.validSyntaxGroup.show()
                self.invalidSyntaxGroup.hide()
                self.parsingResultGroup.show()
                self.parseTreeWebEngineView.load(QtCore.QUrl.fromLocalFile(os.pat
h.abspath("Parse Tree.html")))
            else:
                self.errorMessageLabel.setText(error_message)
                self.validSyntaxGroup.hide()
                self.invalidSyntaxGroup.show()
                self.parsingResultGroup.show()
                self.parseTreeWebEngineView.load(QtCore.QUrl.fromLocalFile(os.pat
h.abspath("Parse Tree.html")))

    def onClickShowParsingTable(self):
        tableDialog = TableDialog()
        #tableDialog.exec()
        #tableDialog.show()
```

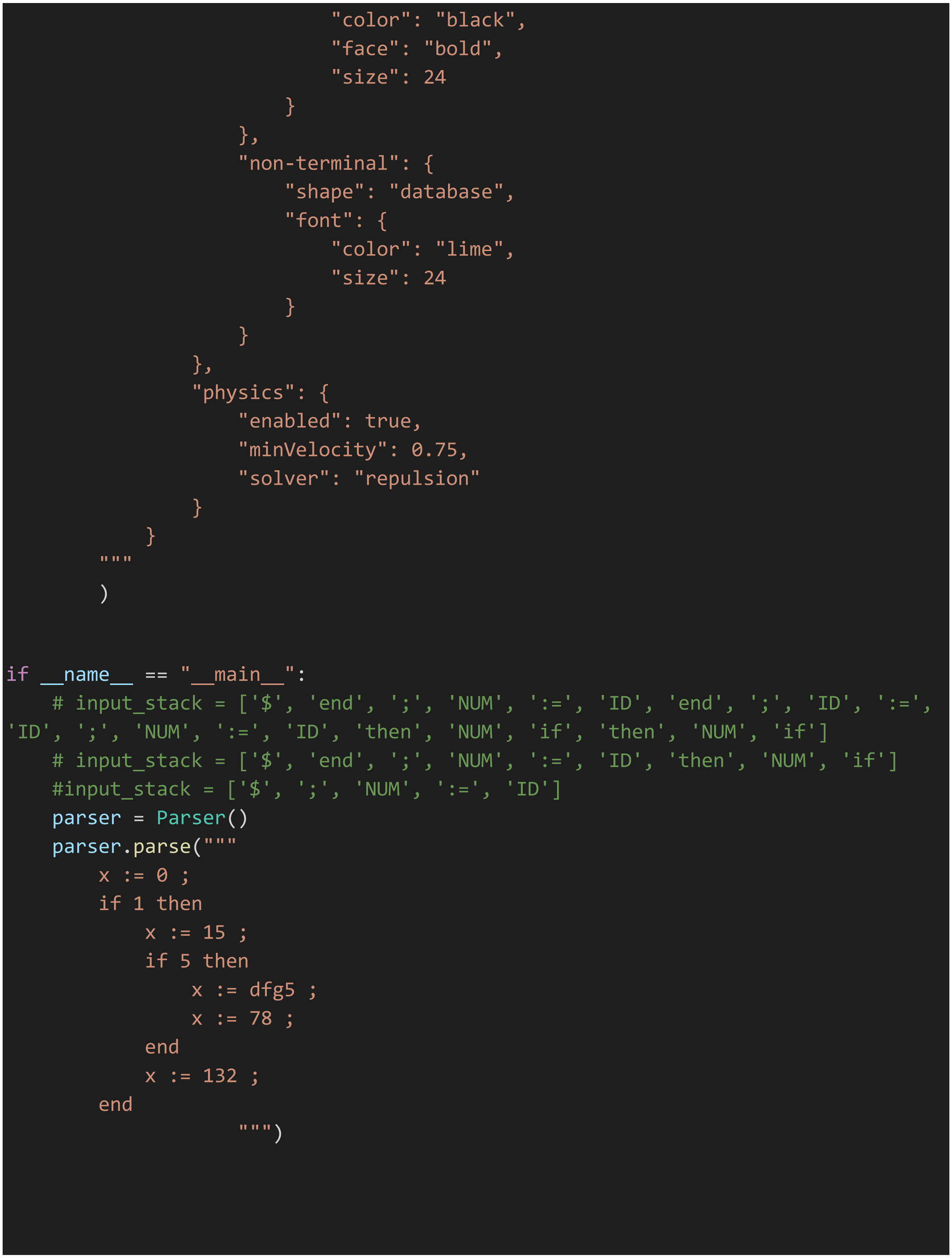
```
class TableDialog(QtWidgets.QDialog):
    def __init__(self):
        super().__init__()
        uic.loadUi('ui\TableDialog.ui',self)
        #self.setFixedSize(self.width, self.height)
        self.okPushButton.clicked.connect(self.hide)
        self.exec()



if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    window = ParserUi()
    app.exec_()
```

2.4. Parser.py

```
from ParsingTable import parsing_table, production_rules
from pyvis.network import Network
from uuid import uuid4
from Scanner import Scanner


class Parser:
    def __init__(self):
        self.table = parsing_table
        self.rule = production_rules
        self.parsing_stack = ['$', 0]
        self.input_stack = ['$']
        self.nodes_stack = []
        self.parse_tree = Network(height='100%', width='100%', directed=True)
        self.__set_parse_tree_options()

    def parse(self, input_code):
        parsing_result = False
        error_message = "Parsing error"
        self.__init__()
        tokens = Scanner().get_tokens(input_code)
        [self.input_stack.append(token) for token in tokens[::-1]]
        while len(self.input_stack) != 0:
            lookahead = self.input_stack[-1]
            state = self.parsing_stack[-1]
            actions = self.table[state]['actions']
```

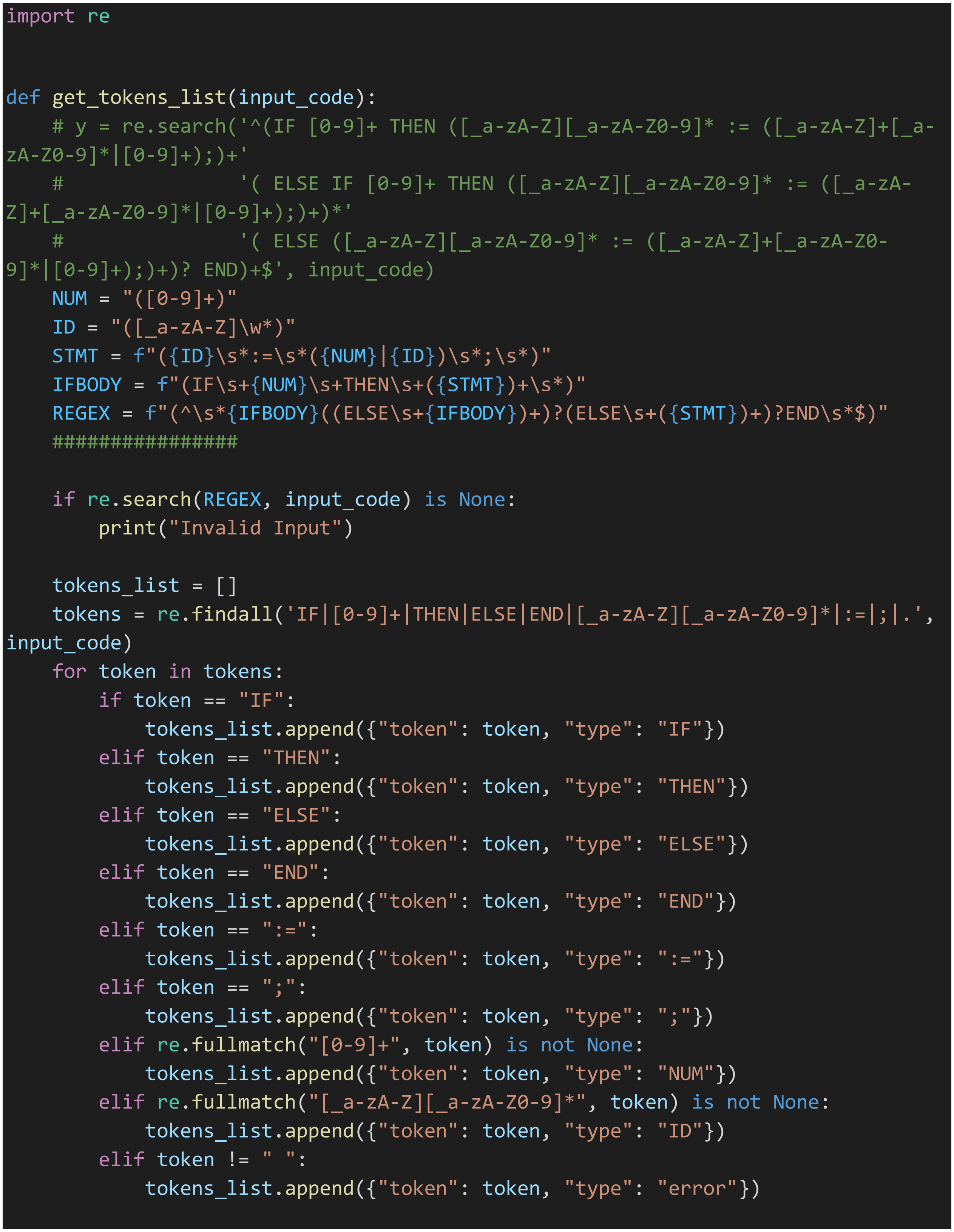


```
            if lookahead in actions.keys():
                action = actions[lookahead][0]  # 's' | 'r'
                if action == 's':
                    next_state = actions[lookahead][1]
                    self.__shift(lookahead, next_state)
                elif action == 'r':
                    rule_no = actions[lookahead][1]
                    self.__reduce(rule_no)
                    # Check if it is acceptance rule
                    if rule_no == 1:
                        parsing_result = True
                        error_message = "_________Parsing Complete__________"
                        break
            else:
                parsing_result = False
                error_message = f"Unexpected terminal!  Current state: {state}, 
Expected: {list(actions.keys())}, Found: {lookahead}"  # Throw Exceptions
                break
        self.parse_tree.save_graph("Parse Tree.html")
        print(error_message)
        return parsing_result, error_message

    def __shift(self, lookahead, next_state):
        self.input_stack.pop()
        self.parsing_stack.append(lookahead)
        self.parsing_stack.append(next_state)
        node_id = str(uuid4())
        self.nodes_stack.append(node_id)
        self.parse_tree.add_node(node_id, label=lookahead, group='terminal', 
shape='ellipse', color='white')
        print(self.parsing_stack)

    def __reduce(self, rule_no):
        left_symbol = self.rule[rule_no][0]
        right_symbols = list(self.rule[rule_no][1])

        for symbol in right_symbols[::-1]:
            self.parsing_stack.pop()
            if symbol == self.parsing_stack[-1]:
                self.parsing_stack.pop()
            else:
                print(
                    f"Reduction error in rule {rule_no}: Expected {symbol}, found 
{self.parsing_stack[-1]}")  # Throw Exceptions
                break
        old_state = self.parsing_stack[-1]
        if left_symbol in self.table[old_state]['goto'].keys():
            next_state = self.table[old_state]['goto'][left_symbol]
            self.parsing_stack.append(left_symbol)
            self.parsing_stack.append(next_state)
            print(self.parsing_stack)
        # else:
        #     print("error3")  # Throw Exceptions
        #     break

        n = len(right_symbols)
        child_nodes_ids = self.nodes_stack[-n:]
        parent_id = str(uuid4())
        self.parse_tree.add_node(parent_id, label=left_symbol, group='non-terminal', shape='ellipse')
        for child_id in child_nodes_ids:
            self.parse_tree.add_edge(parent_id, child_id)
            self.nodes_stack.pop()
        self.nodes_stack.append(parent_id)

    def __set_parse_tree_options(self):
        self.parse_tree.set_options(
        """
            {
                "nodes": {
                    "color": {
                        "border": "#03dac8",
                        "background": "white"
                    },
                    "shape": "text"
                },
                "interaction": {
                    "keyboard": {
                        "enabled": true
                    },
                    "navigationButtons": true
                },
                "layout": {
                    "hierarchical": {
                        "sortMethod": "directed"
                    }
                },
                "groups": {
                    "terminal": {
                        "font": {
```

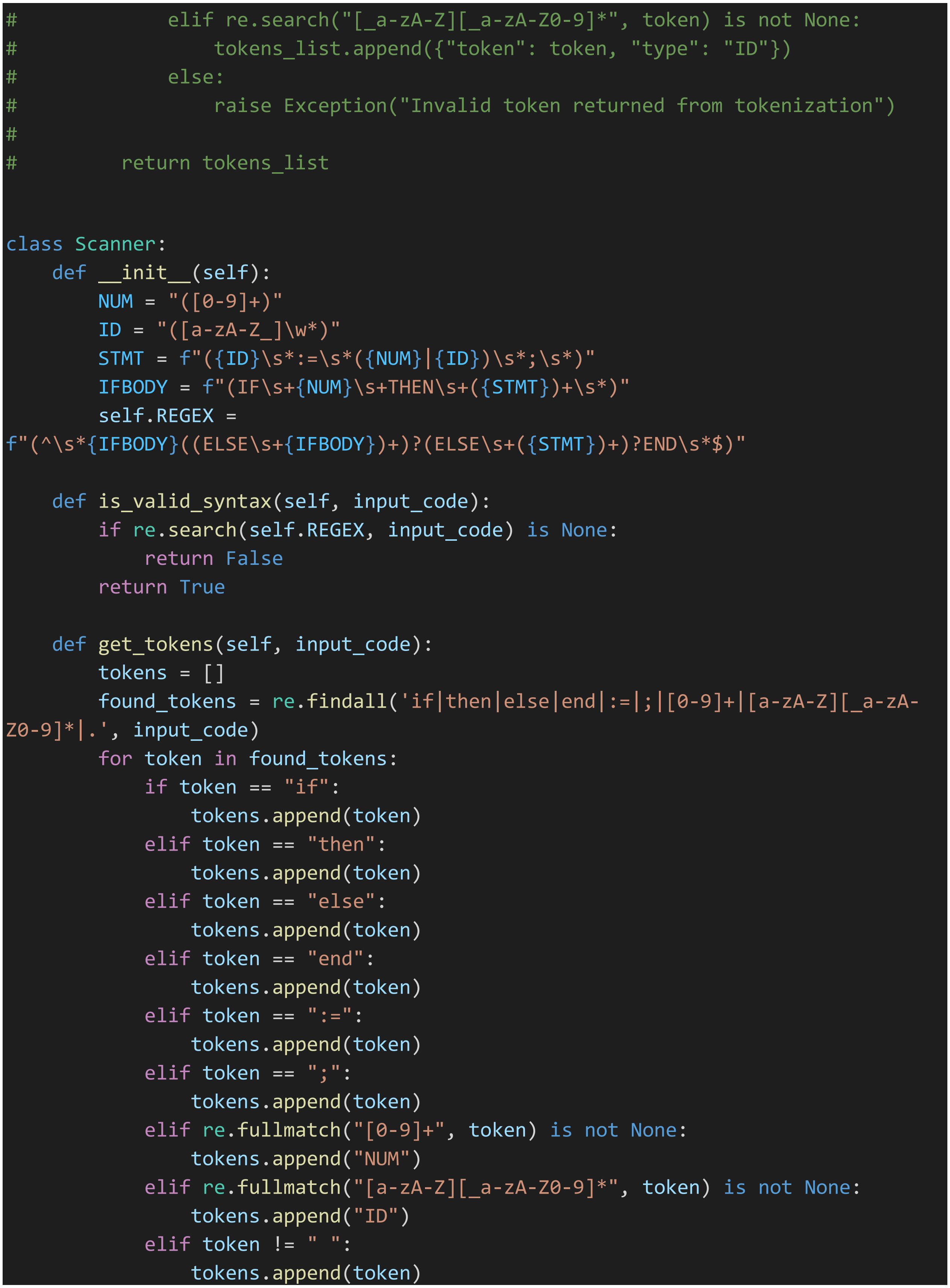


```
                            "color": "black",
                            "face": "bold",
                            "size": 24
                        }
                    },
                    "non-terminal": {
                        "shape": "database",
                        "font": {
                            "color": "lime",
                            "size": 24
                        }
                    }
                },
                "physics": {
                    "enabled": true,
                    "minVelocity": 0.75,
                    "solver": "repulsion"
                }
            }
        """
        )


if __name__ == "__main__":
    # input_stack = ['$', 'end', ';', 'NUM', ':=', 'ID', 'end', ';', 'ID', ':=', 'ID', ';', 'NUM', ':=', 'ID', 'then', 'NUM', 'if', 'then', 'NUM', 'if']
    # input_stack = ['$', 'end', ';', 'NUM', ':=', 'ID', 'then', 'NUM', 'if']
    #input_stack = ['$', ';', 'NUM', ':=', 'ID']
    parser = Parser()
    parser.parse("""
        x := 0 ;
        if 1 then
            x := 15 ;
            if 5 then
                x := dfg5 ;
                x := 78 ;
            end
            x := 132 ;
        end
                    """)
```

2.5. Scanner.py



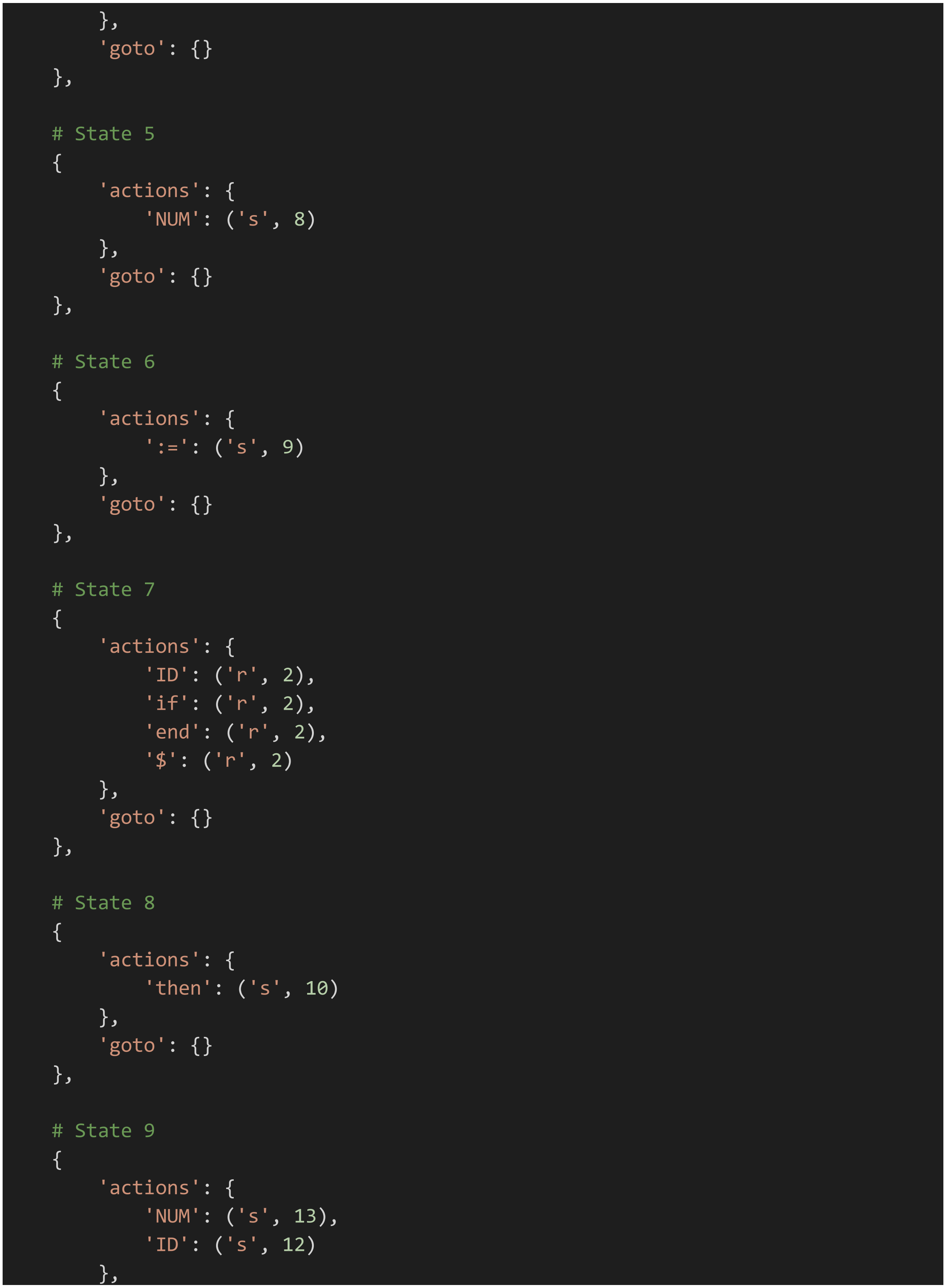
```
import re


def get_tokens_list(input_code):
    # y = re.search('^(IF [0-9]+ THEN ([_a-zA-Z][_a-zA-Z0-9]* := ([_a-zA-Z]+[_a-
zA-Z0-9]*|[0-9]+);)+'
    #                '( ELSE IF [0-9]+ THEN ([_a-zA-Z][_a-zA-Z0-9]* := ([_a-zA-
Z]+[_a-zA-Z0-9]*|[0-9]+);)+)*'
    #                '( ELSE ([_a-zA-Z][_a-zA-Z0-9]* := ([_a-zA-Z]+[_a-zA-Z0-
9]*|[0-9]+);)+)? END)+$', input_code)
    NUM = "([0-9]+)"
    ID = "([_a-zA-Z]\w*)"
    STMT = f"({ID}\s*:=\s*({NUM}|{ID})\s*;\s*)"
    IFBODY = f"(IF\s+{NUM}\s+THEN\s+({STMT})+\s*)"
    REGEX = f"(^\s*{IFBODY}((ELSE\s+{IFBODY})+)?(ELSE\s+({STMT})+)?END\s*$)"
    ################

    if re.search(REGEX, input_code) is None:
        print("Invalid Input")

    tokens_list = []
    tokens = re.findall('IF|[0-9]+|THEN|ELSE|END|[_a-zA-Z][_a-zA-Z0-9]*|:=|;|.',
input_code)
    for token in tokens:
        if token == "IF":
            tokens_list.append({"token": token, "type": "IF"})
        elif token == "THEN":
            tokens_list.append({"token": token, "type": "THEN"})
        elif token == "ELSE":
            tokens_list.append({"token": token, "type": "ELSE"})
        elif token == "END":
            tokens_list.append({"token": token, "type": "END"})
        elif token == ":=":
            tokens_list.append({"token": token, "type": ":="})
        elif token == ";":
            tokens_list.append({"token": token, "type": ";"})
        elif re.fullmatch("[0-9]+", token) is not None:
            tokens_list.append({"token": token, "type": "NUM"})
        elif re.fullmatch("[_a-zA-Z][_a-zA-Z0-9]*", token) is not None:
            tokens_list.append({"token": token, "type": "ID"})
        elif token != " ":
            tokens_list.append({"token": token, "type": "error"})
    return tokens_list



# import re
#
#
# def get_tokens_list(input_code):
#     # y = re.search('^(IF [0-9]+ THEN ([_a-zA-Z][_a-zA-Z0-9]* := ([_a-zA-
Z]+[_a-zA-Z0-9]*|[0-9]+);)+'
#     #               '( ELSE IF [0-9]+ THEN ([_a-zA-Z][_a-zA-Z0-9]* := ([_a-zA-
Z]+[_a-zA-Z0-9]*|[0-9]+);)+)*'
#     #               '( ELSE ([_a-zA-Z][_a-zA-Z0-9]* := ([_a-zA-Z]+[_a-zA-Z0-
9]*|[0-9]+);)+)? END)+$', input_code)
#     NUM = "([0-9]+)"
#     ID = "([_a-zA-Z]\w*)"
#     STMT = f"({ID}\s*:=\s*({NUM}|{ID})\s*;\s*)"
#     IFBODY = f"(IF\s+{NUM}\s+THEN\s+({STMT})+\s*)"
#     REGEX = f"(\s*{IFBODY}((ELSE\s+{IFBODY})+)?(ELSE\s+({STMT})+)?END\s*)"
#     ################
#     y = re.fullmatch(REGEX, input_code)
#
#     if y is None:
#         print("Invalid IF statement")
#         return None
#     else:
#         tokens_list = []
#         tokens = re.findall('IF|[0-9]+|THEN|ELSE|END|[_a-zA-Z][_a-zA-Z0-
9]*|:=|;', y.string)
#         for token in tokens:
#             if token == "IF":
#                 tokens_list.append({"token": token, "type": "IF"})
#             elif token == "THEN":
#                 tokens_list.append({"token": token, "type": "THEN"})
#             elif token == "ELSE":
#                 tokens_list.append({"token": token, "type": "ELSE"})
#             elif token == "END":
#                 tokens_list.append({"token": token, "type": "END"})
#             elif token == ":=":
#                 tokens_list.append({"token": token, "type": ":="})
#             elif token == ";":
#                 tokens_list.append({"token": token, "type": ";"})
#             elif re.search("[0-9]+", token) is not None:
#                 tokens_list.append({"token": token, "type": "NUM"})
```



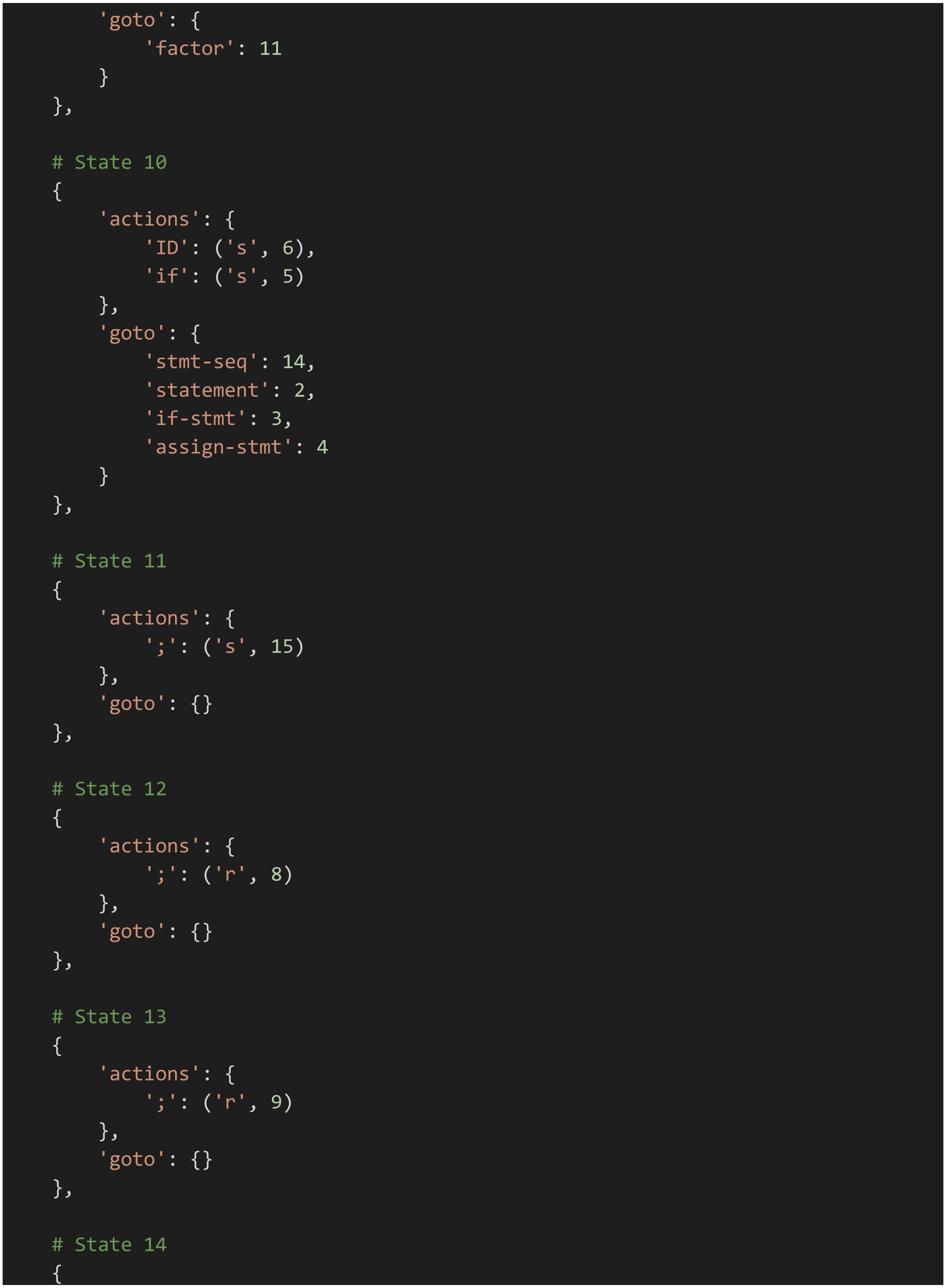
```
#             elif re.search("[_a-zA-Z][_a-zA-Z0-9]*", token) is not None:
#                 tokens_list.append({"token": token, "type": "ID"})
#             else:
#                 raise Exception("Invalid token returned from tokenization")
#
#         return tokens_list


class Scanner:
    def __init__(self):
        NUM = "([0-9]+)"
        ID = "([a-zA-Z_]\w*)"
        STMT = f"({ID}\s*:=\s*({NUM}|{ID})\s*;\s*)"
        IFBODY = f"(IF\s+{NUM}\s+THEN\s+({STMT})+\s*)"
        self.REGEX =
f"(^\s*{IFBODY}((ELSE\s+{IFBODY})+)?(ELSE\s+({STMT})+)?END\s*$)"

    def is_valid_syntax(self, input_code):
        if re.search(self.REGEX, input_code) is None:
            return False
        return True

    def get_tokens(self, input_code):
        tokens = []
        found_tokens = re.findall('if|then|else|end|:=|;|[0-9]+|[a-zA-Z][_a-zA-
Z0-9]*|.', input_code)
        for token in found_tokens:
            if token == "if":
                tokens.append(token)
            elif token == "then":
                tokens.append(token)
            elif token == "else":
                tokens.append(token)
            elif token == "end":
                tokens.append(token)
            elif token == ":=":
                tokens.append(token)
            elif token == ";":
                tokens.append(token)
            elif re.fullmatch("[0-9]+", token) is not None:
                tokens.append("NUM")
            elif re.fullmatch("[a-zA-Z][_a-zA-Z0-9]*", token) is not None:
                tokens.append("ID")
            elif token != " ":
                tokens.append(token)
        return tokens


    def get_tokens_list(self, input_code):
        tokens_list = []
        tokens = re.findall('if|then|else|end|:=|;|[0-9]+|[a-zA-Z][_a-zA-Z0-
9]*|.', input_code)
        for token in tokens:
            if token == "if":
                tokens_list.append({"token": token, "type": "IF"})
            elif token == "then":
                tokens_list.append({"token": token, "type": "THEN"})
            elif token == "else":
                tokens_list.append({"token": token, "type": "ELSE"})
            elif token == "end":
                tokens_list.append({"token": token, "type": "END"})
            elif token == ":=":
                tokens_list.append({"token": token, "type": ":="})
            elif token == ";":
                tokens_list.append({"token": token, "type": ";"})
            elif re.fullmatch("[0-9]+", token) is not None:
                tokens_list.append({"token": token, "type": "NUM"})
            elif re.fullmatch("[a-zA-Z][_a-zA-Z0-9]*", token) is not None:
                tokens_list.append({"token": token, "type": "ID"})
            elif token != " ":
                tokens_list.append({"token": token, "type": "UNDEFINED"})
        return tokens_list
```

2.6. ParsingTable.py

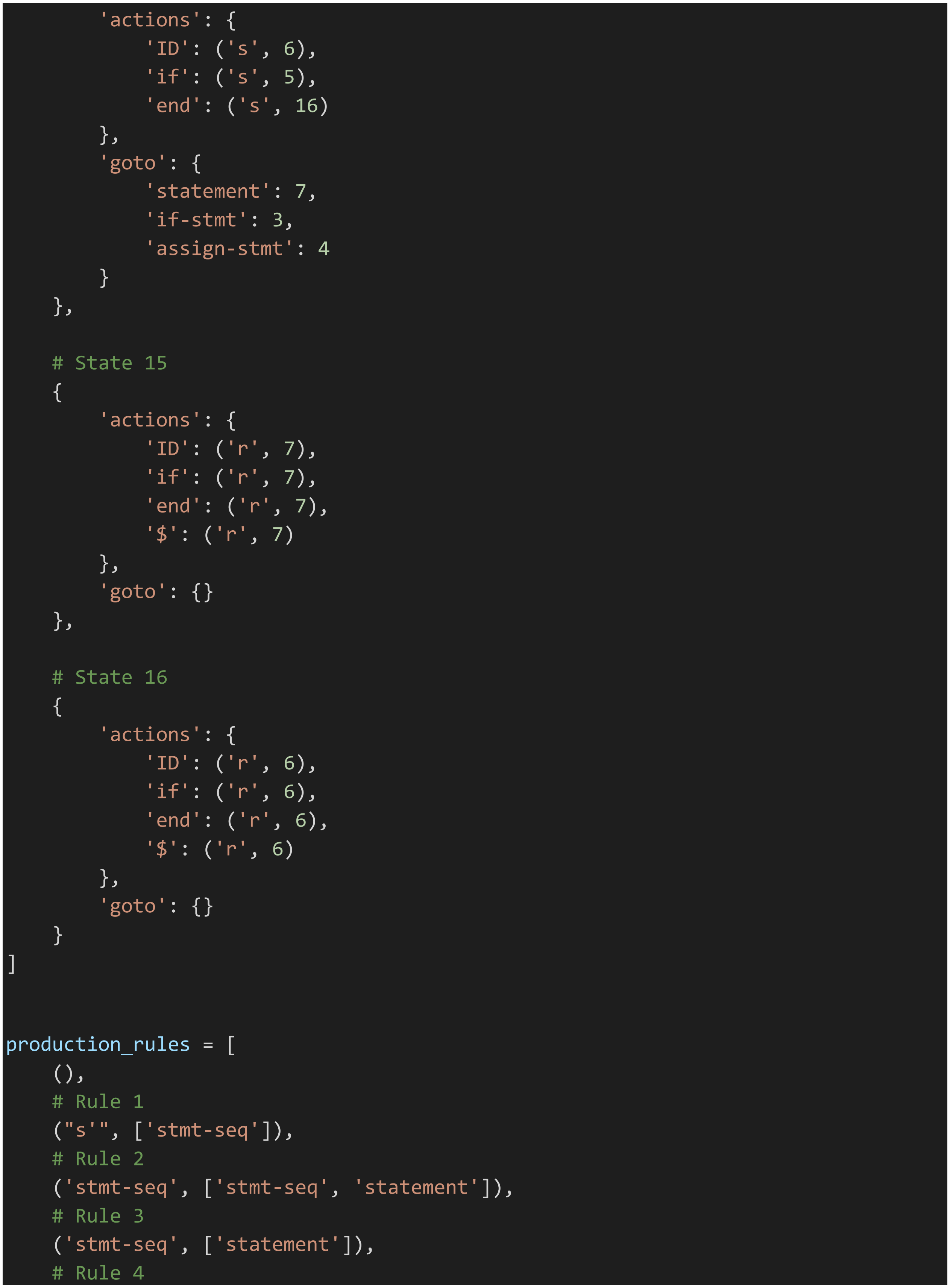
```
parsing_table = [
    # State 0
    {
        'actions': {
            'ID': ('s', 6),
            'if': ('s', 5)
        },
        'goto': {
            'stmt-seq': 1,
            'statement': 2,
            'if-stmt': 3,
            'assign-stmt': 4,
        }
    },

    # State 1
    {
        'actions': {
            'ID': ('s', 6),
            'if': ('s', 5),
            '$': ('r', 1)       # acceptance
        },
        'goto': {
            'statement': 7,
            'if-stmt': 3,
            'assign-stmt': 4
        }
    },

    # State 2
    {
        'actions': {
            'ID': ('r', 3),
            'if': ('r', 3),
            'end': ('r', 3),
            '$': ('r', 3)
        },
        'goto': {}
    },

    # State 3
    {
        'actions': {
            'ID': ('r', 4),
            'if': ('r', 4),
            'end': ('r', 4),
            '$': ('r', 4)
        },
        'goto': {}
    },

    # State 4
    {
        'actions': {
            'ID': ('r', 5),
            'if': ('r', 5),
            'end': ('r', 5),
            '$': ('r', 5)
```



```
        },
        'goto': {}
    },

    # State 5
    {
        'actions': {
            'NUM': ('s', 8)
        },
        'goto': {}
    },

    # State 6
    {
        'actions': {
            ':=': ('s', 9)
        },
        'goto': {}
    },

    # State 7
    {
        'actions': {
            'ID': ('r', 2),
            'if': ('r', 2),
            'end': ('r', 2),
            '$': ('r', 2)
        },
        'goto': {}
    },

    # State 8
    {
        'actions': {
            'then': ('s', 10)
        },
        'goto': {}
    },

    # State 9
    {
        'actions': {
            'NUM': ('s', 13),
            'ID': ('s', 12)
        },
```



```
        'goto': {
            'factor': 11
        }
    },

    # State 10
    {
        'actions': {
            'ID': ('s', 6),
            'if': ('s', 5)
        },
        'goto': {
            'stmt-seq': 14,
            'statement': 2,
            'if-stmt': 3,
            'assign-stmt': 4
        }
    },

    # State 11
    {
        'actions': {
            ';': ('s', 15)
        },
        'goto': {}
    },

    # State 12
    {
        'actions': {
            ';': ('r', 8)
        },
        'goto': {}
    },

    # State 13
    {
        'actions': {
            ';': ('r', 9)
        },
        'goto': {}
    },

    # State 14
    {
```



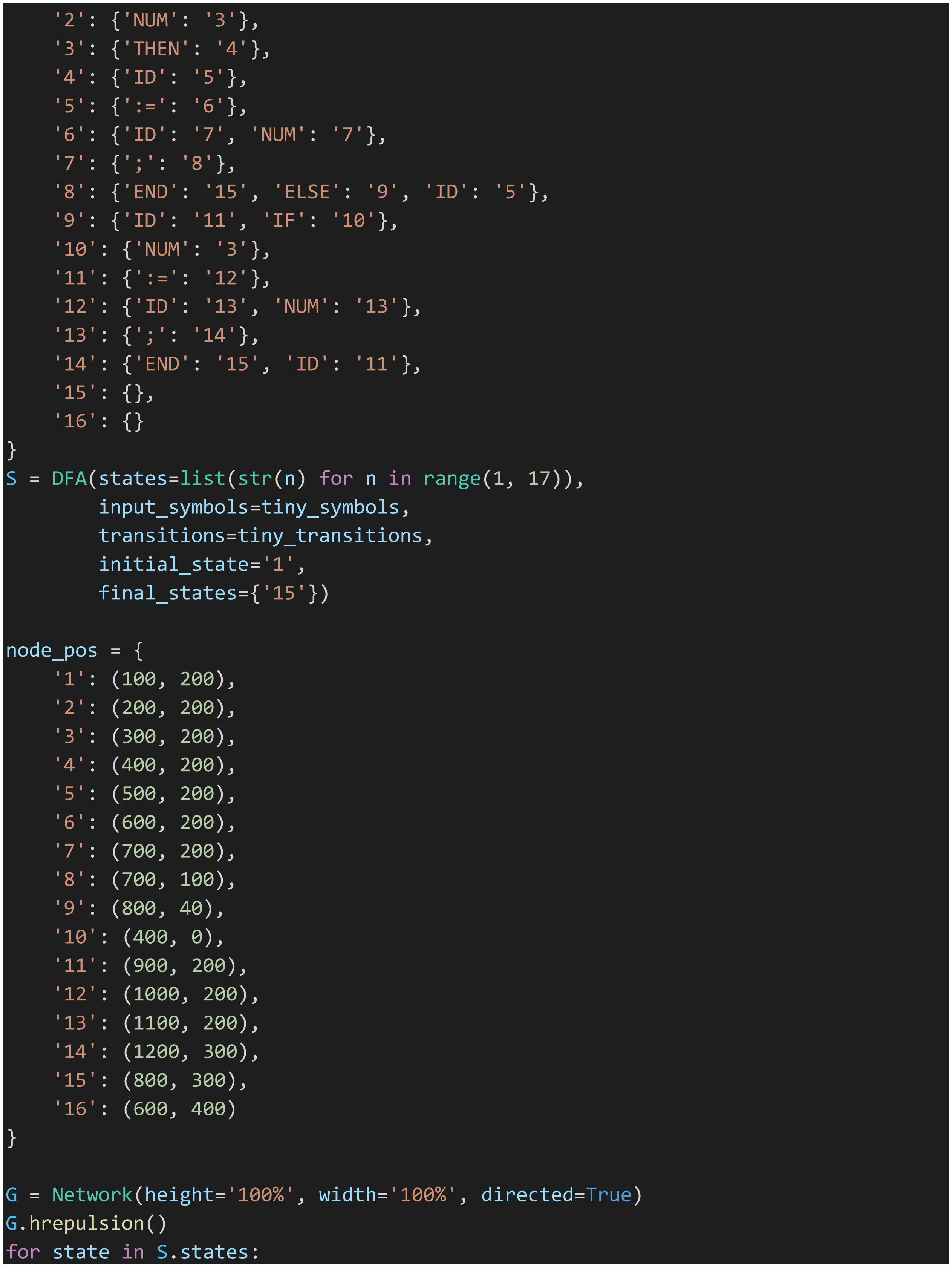
```
        'actions': {
            'ID': ('s', 6),
            'if': ('s', 5),
            'end': ('s', 16)
        },
        'goto': {
            'statement': 7,
            'if-stmt': 3,
            'assign-stmt': 4
        }
    },

    # State 15
    {
        'actions': {
            'ID': ('r', 7),
            'if': ('r', 7),
            'end': ('r', 7),
            '$': ('r', 7)
        },
        'goto': {}
    },

    # State 16
    {
        'actions': {
            'ID': ('r', 6),
            'if': ('r', 6),
            'end': ('r', 6),
            '$': ('r', 6)
        },
        'goto': {}
    }
]


production_rules = [
    (),
    # Rule 1
    ("s'", ['stmt-seq']),
    # Rule 2
    ('stmt-seq', ['stmt-seq', 'statement']),
    # Rule 3
    ('stmt-seq', ['statement']),
    # Rule 4
    ('statement', ['if-stmt']),
    # Rule 5
    ('statement', ['assign-stmt']),
    # Rule 6
    ('if-stmt', ['if', 'NUM', 'then', 'stmt-seq', 'end']),
    # Rule 7
    ('assign-stmt', ['ID', ':=', 'factor', ';']),
    # Rule 8
    ('factor', ['ID']),
    # Rule 9
    ('factor', ['NUM']),
]
```

2.7. dfa.py

```
#!/usr/bin/env python3
"""Classes and methods for working with deterministic finite automata."""
import copy
from collections import defaultdict


class DFA:
    """A deterministic finite automaton."""

    def __init__(self, *, states, input_symbols, transitions,
                 initial_state, final_states, allow_partial=False):
        """Initialize a complete DFA."""
        self.states = states.copy()
        self.input_symbols = input_symbols.copy()
        self.transitions = copy.deepcopy(transitions)
        self.initial_state = initial_state
        self.final_states = final_states.copy()
        self.allow_partial = allow_partial

    def __lt__(self, other):
        """Return True if this DFA is a strict subset of another DFA."""
        if isinstance(other, DFA):
            return self <= other and self != other
        else:
            raise NotImplementedError

    def __gt__(self, other):
        """Return True if this DFA is a strict superset of another DFA."""
        if isinstance(other, DFA):
            return self >= other and self != other
        else:
            raise NotImplementedError

    def _get_next_current_state(self, current_state, input_symbol):
        """
        Follow the transition for the given input symbol on the current state.

        Raise an error if the transition does not exist.
        """
        if input_symbol in self.transitions[current_state]:
            return self.transitions[current_state][input_symbol]

    def _make_graph(self):
        """
        Returns a simple graph representation of the DFA.
        """
        G = defaultdict(set)
        for k, v in self.transitions.items():
            for c, u in v.items():
                G[k].add(u)
        return G

    def _reachable_nodes(self, G, v, vis):
        """
        Computes the set of reachable nodes
        in the graph G starting at vertex v.
        """
        if v not in vis:
            vis.add(v)
            for u in G[v]:
                self._reachable_nodes(G, u, vis)
```

2.8. plot.py

```
import networkx as nx
import matplotlib.pyplot as plt
from pyvis.network import Network
from dfa import DFA

tiny_symbols = {'IF', 'NUM', 'THEN', 'ID', 'ELSE', 'END', ';', ':='}
tiny_transitions = {
    '1': {'IF': '2'},
```



```
    '2': {'NUM': '3'},
    '3': {'THEN': '4'},
    '4': {'ID': '5'},
    '5': {':=': '6'},
    '6': {'ID': '7', 'NUM': '7'},
    '7': {';': '8'},
    '8': {'END': '15', 'ELSE': '9', 'ID': '5'},
    '9': {'ID': '11', 'IF': '10'},
    '10': {'NUM': '3'},
    '11': {':=': '12'},
    '12': {'ID': '13', 'NUM': '13'},
    '13': {';': '14'},
    '14': {'END': '15', 'ID': '11'},
    '15': {},
    '16': {}
}
S = DFA(states=list(str(n) for n in range(1, 17)),
        input_symbols=tiny_symbols,
        transitions=tiny_transitions,
        initial_state='1',
        final_states={'15'})

node_pos = {
    '1': (100, 200),
    '2': (200, 200),
    '3': (300, 200),
    '4': (400, 200),
    '5': (500, 200),
    '6': (600, 200),
    '7': (700, 200),
    '8': (700, 100),
    '9': (800, 40),
    '10': (400, 0),
    '11': (900, 200),
    '12': (1000, 200),
    '13': (1100, 200),
    '14': (1200, 300),
    '15': (800, 300),
    '16': (600, 400)
}

G = Network(height='100%', width='100%', directed=True)
G.hrepulsion()
for state in S.states:
    G.add_node(int(state), shape="ellipse", physics=False, x=node_pos[state][0], 
y=node_pos[state][1])

G.nodes[0]["color"] = 'yellow'
G.nodes[0]["title"] = "Initial State"

G.nodes[14]["color"] = {"background": 'DodgerBlue', "border": 'blue'}
G.nodes[14]["borderWidth"] = 5
G.nodes[14]["borderWidthSelected"] = 5
G.nodes[14]["title"] = "Goal State"

G.nodes[15]["label"] = 'D'
G.nodes[15]["color"] = 'DarkGray'
G.nodes[15]["title"] = "Dead State"

for k in S.states:
    for kk in S.transitions[k]:
        G.add_edge(int(k), int(S.transitions[k][kk]), label=kk)

G.set_options("""
var options = {
                  "edges": {
                    "smooth": {
                        "enabled" : true
                    },
                    "color": {
                        "inherit" : false
                    }
                  },
                  "interaction": {
                    "hover": true,
                    "keyboard": {
                      "enabled": true
                    },
                    "multiselect": true,
                    "navigationButtons": true
                  }
                }
""")
G.save_graph("DFA.html")
```