Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS 4308-W02: Concepts of Programming Languages

Deliverable 3 - Interpreter

Jesse Schultheis [redacted]

Katlin Scott [redacted]

William Stigall [redacted]

Hailey Walker [redacted]

November 13, 2023

Initial Problem Statement

The problem is to develop an executer program for a subset of the SCL language, working in conjunction with previously developed scanner and parser programs, resulting in a complete interpreter/translator to intermediate code and an abstract machine that includes the scanner, parser, and executer.

Summary/Purpose

The purpose of this deliverable is to create an interpreter for a subset of the SCL language using Python or another language not utilized in prior phases, encompassing a scanner, parser, and executer to process input SCL programs, generate tokens, construct a parse tree, execute program actions, manage memory, and demonstrate successful execution of identified statements from varied input files.

Detailed Description/Work Done

For the scanner, Python code defines the keywords ("DISPLAY", "IF", "THEN", "ENDIF", "FUNCTION", "IS", "ENDFUN", "PARAMETERS", "INTEGER", "FLOAT", "CHAR", "NOT") and token types (“IDENTIFIER”, “UNSIGNICON”, “SIGNICON”, “PLUS”, “MINUS”, “STAR”, “DIVOP”, “EQUOP”, “RELOP”, “LB”, “RB”, “LP”, “RP”, “COMMA”, “STRING_LITERAL”), takes in six SCL files and reads them. It then tokenizes all of the files and puts each token with its correct token type in the output.json file.

For the parser, the Python code takes in a .JSON file, and it walks through each token in the file. As long as it is not "ENDIF", "ELSE", "ENDFUN", or "EOF", the parser continues and identifies parts of the grammar such as assignments, function calls, if-else statements, lists, and algebraic expressions. If at any point the tokens do not follow the established grammar, an error is thrown, and the user is told that the information in the file does not follow the subset of the SCL grammar.

The executer reads each line of code and decides if it is an assignment statement, if statement, etc. Once it figures out what kind of code the line is, it follows the logic of that statement. I'd like to specifically highlight that if the executer reads an if statement, it will find the condition, determine its veracity, then it will either execute or skip everything between the THEN and the ENDIF. This was made possible by using a line skip variable that acts as an on/off switch for whether the inside of the if statement should be skipped.

Input Files

arduino_ex1.scl

arrayex1b.scl

bitops1.scl

datablistp.scl

linkedg.scl

welcome.scl

Test.scl	→ Used in the screenshots in this report
Test2.scl	→ Used in the screenshots in this report
Test3.scl	→ Used in the screenshots in this report

Output Files

output.json	
tokens.json	→ Used in the screenshots in this report

Grammar

<program> ::= <statements>

<statements> ::= <statement> | <statement> <statements>

<statement> ::= <var_declaration> | <expression> | <print_statement> | <if_statement> | <function_declaration> | <function_call>

<var_declaration> ::= IDENTIFIER EQUOP <expression> | IDENTIFIER <array_def>

<array_def> ::= LB <expression> RB

<expression> ::= <term> | <term> PLUS <term> | <term> MINUS <term> | <term> STAR <term> | <term> DIVOP <term>

<term> ::= IDENTIFIER | UNSIGNICON | SIGNICON | <expression> | <function_call>

<print_statement> ::= DISPLAY <expression_list>

<expression_list> ::= <expression> | <expression> COMMA <expression_list>

<if_statement> ::= IF <condition> THEN <statements> ENDIF

<condition> ::= <expression> RELOP <expression> | NOT <condition>

<function_declaration> ::= FUNCTION IDENTIFIER parameters IS <statements> ENDFUN IDENTIFIER

<parameters> ::= PARAMETERS param_list | ε

<param_list> ::= IDENTIFIER OF <data_type> | IDENTIFIER OF <data_type> COMMA <param_list>

<data_type> ::= INTEGER | FLOAT | CHAR

<function_call> ::= IDENTIFIER parguments

<parguments> ::= LP <expression_list> RP

Limitations of the above specification and design of the system

One thing that could be improved in this system is its ability to assign a long arithmetic expression. At the moment, it can only use one operator in between two numbers (e.g., 2+3). Most code that is written uses many operators with many different numbers (e.g., 1+2-3).

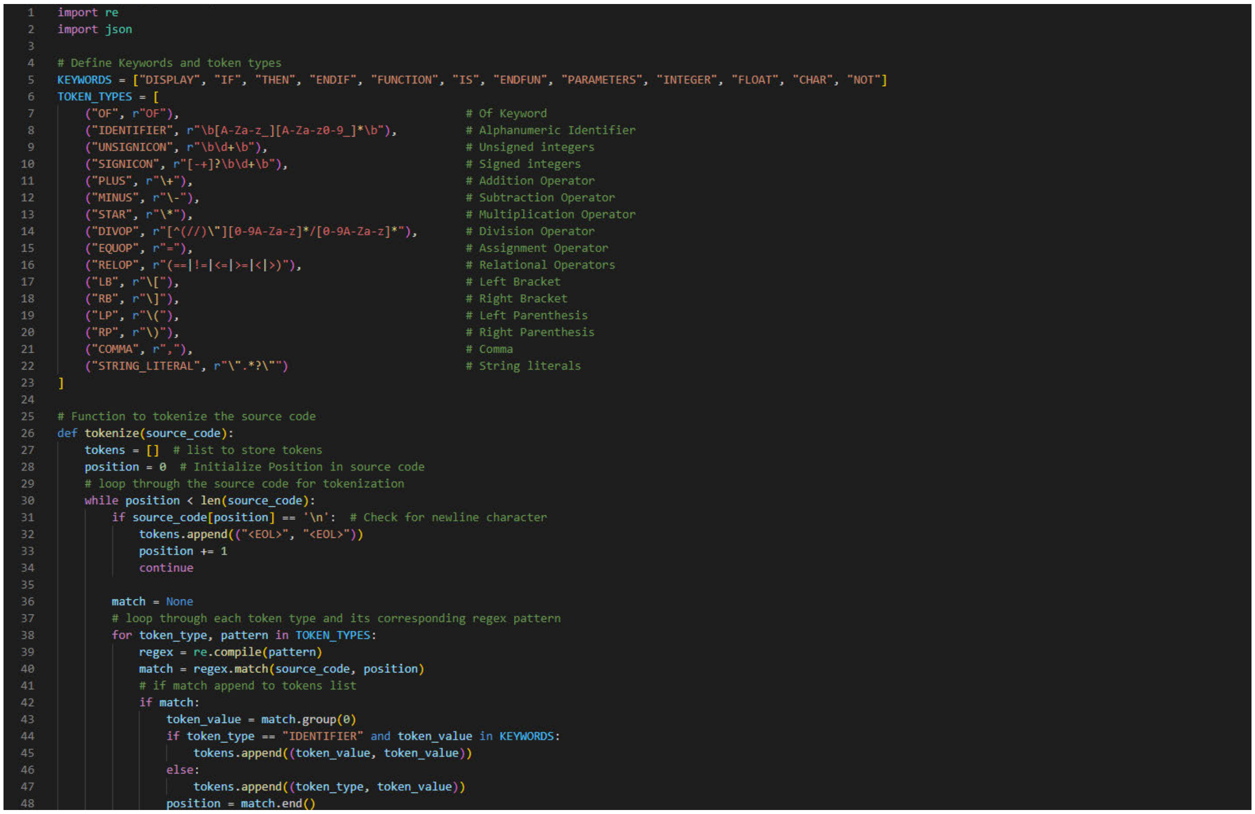
Discussion of how the solution can be improved and extended

One way we could improve is by taking more possible errors into account. We could consider any errors that may occur during tokenization. There could also be more error handling when the file is being read. With the previous deliverables, we could have included more of the SCL grammar as part of our subset.

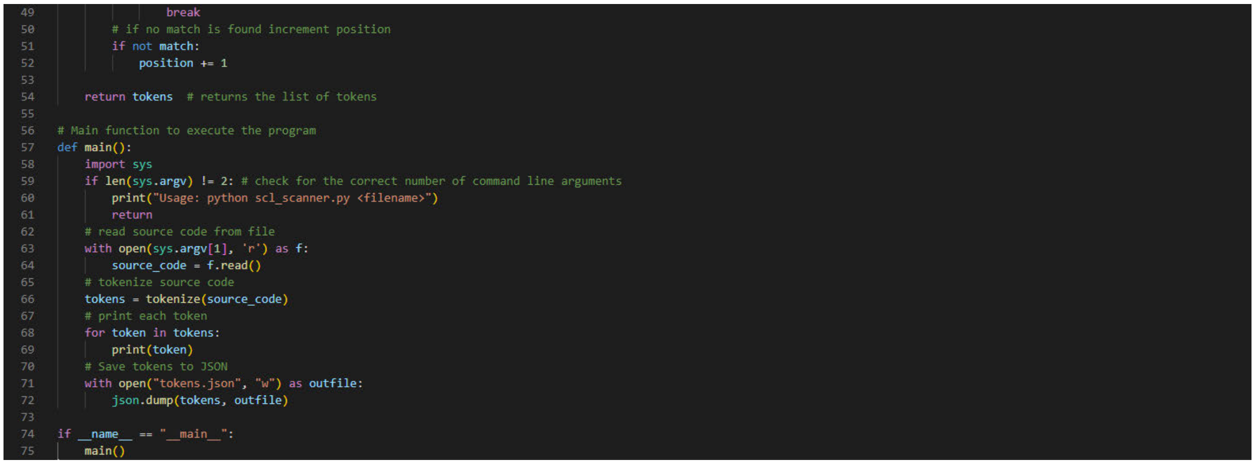
The analyze() function is long, and – while improving modularity might improve code organization and maintenance – it also would introduce a layer of complexity for quick and effective debugging. The interpreter could also be extended by including more of the SCL grammar present to create a larger subset than the one we used.

Source Code Screenshots

scl_scanner.py Source Code:

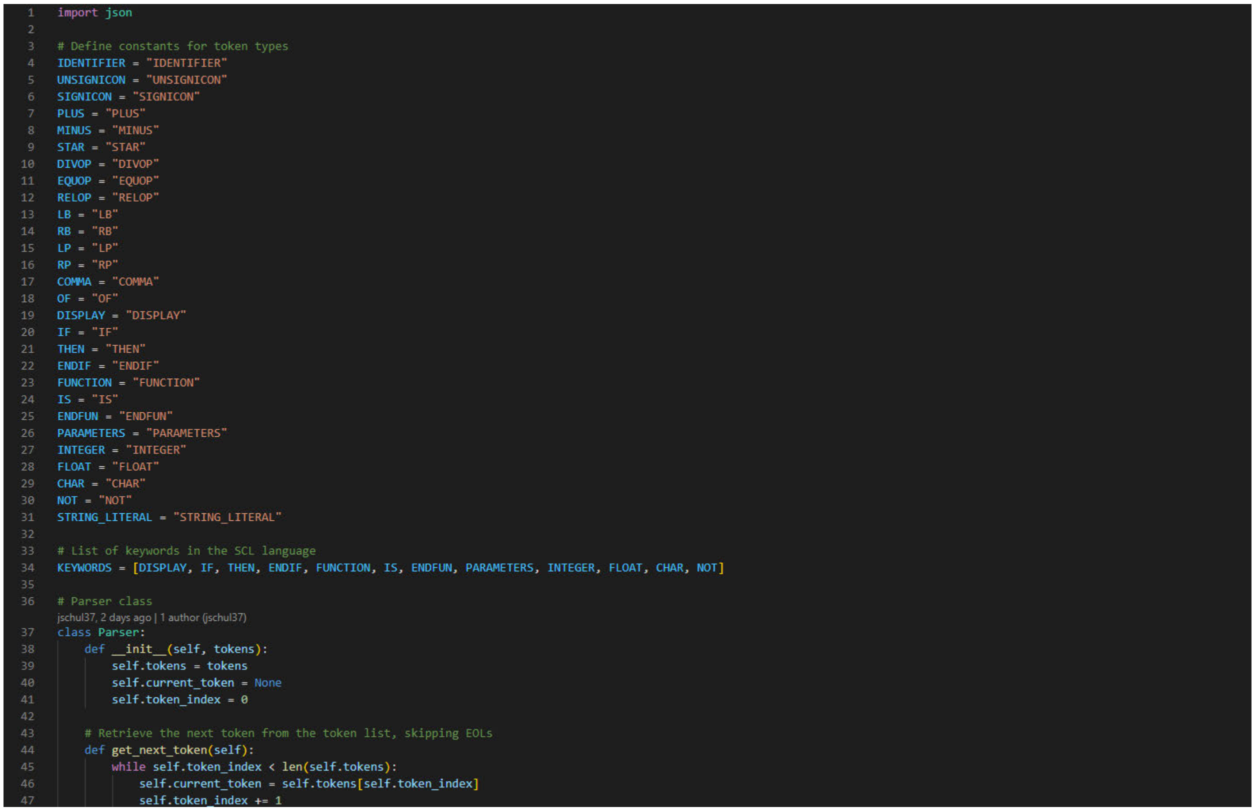


```python
import re
import json

# Define Keywords and token types
KEYWORDS = ["DISPLAY", "IF", "THEN", "ENDIF", "FUNCTION", "IS", "ENDFUN", "PARAMETERS", "INTEGER", "FLOAT", "CHAR", "NOT"]
TOKEN_TYPES = [
    ("OF", r"OF"),                                          # Of Keyword
    ("IDENTIFIER", r"\b[A-Za-z_][A-Za-z0-9_]*\b"),          # Alphanumeric Identifier
    ("UNSIGNICON", r"\b\d+\b"),                             # Unsigned integers
    ("SIGNICON", r"[-+]?\b\d+\b"),                          # Signed integers
    ("PLUS", r"\+"),                                        # Addition Operator
    ("MINUS", r"\-"),                                       # Subtraction Operator
    ("STAR", r"\*"),                                        # Multiplication Operator
    ("DIVOP", r"[^(//)\"][0-9A-Za-z]*/[0-9A-Za-z]*"),        # Division Operator
    ("EQUOP", r"="),                                        # Assignment Operator
    ("RELOP", r"(==|!=|<=|>=|<|>)"),                        # Relational Operators
    ("LB", r"\["),                                          # Left Bracket
    ("RB", r"\]"),                                          # Right Bracket
    ("LP", r"\("),                                          # Left Parenthesis
    ("RP", r"\)"),                                          # Right Parenthesis
    ("COMMA", r","),                                        # Comma
    ("STRING_LITERAL", r"\".*?\"")                          # String literals
]

# Function to tokenize the source code
def tokenize(source_code):
    tokens = []  # list to store tokens
    position = 0  # Initialize Position in source code
    # loop through the source code for tokenization
    while position < len(source_code):
        if source_code[position] == '\n':  # Check for newline character
            tokens.append(("<EOL>", "<EOL>"))
            position += 1
            continue

        match = None
        # loop through each token type and its corresponding regex pattern
        for token_type, pattern in TOKEN_TYPES:
            regex = re.compile(pattern)
            match = regex.match(source_code, position)
            # if match append to tokens list
            if match:
                token_value = match.group(0)
                if token_type == "IDENTIFIER" and token_value in KEYWORDS:
                    tokens.append((token_value, token_value))
                else:
                    tokens.append((token_type, token_value))
                position = match.end()
```

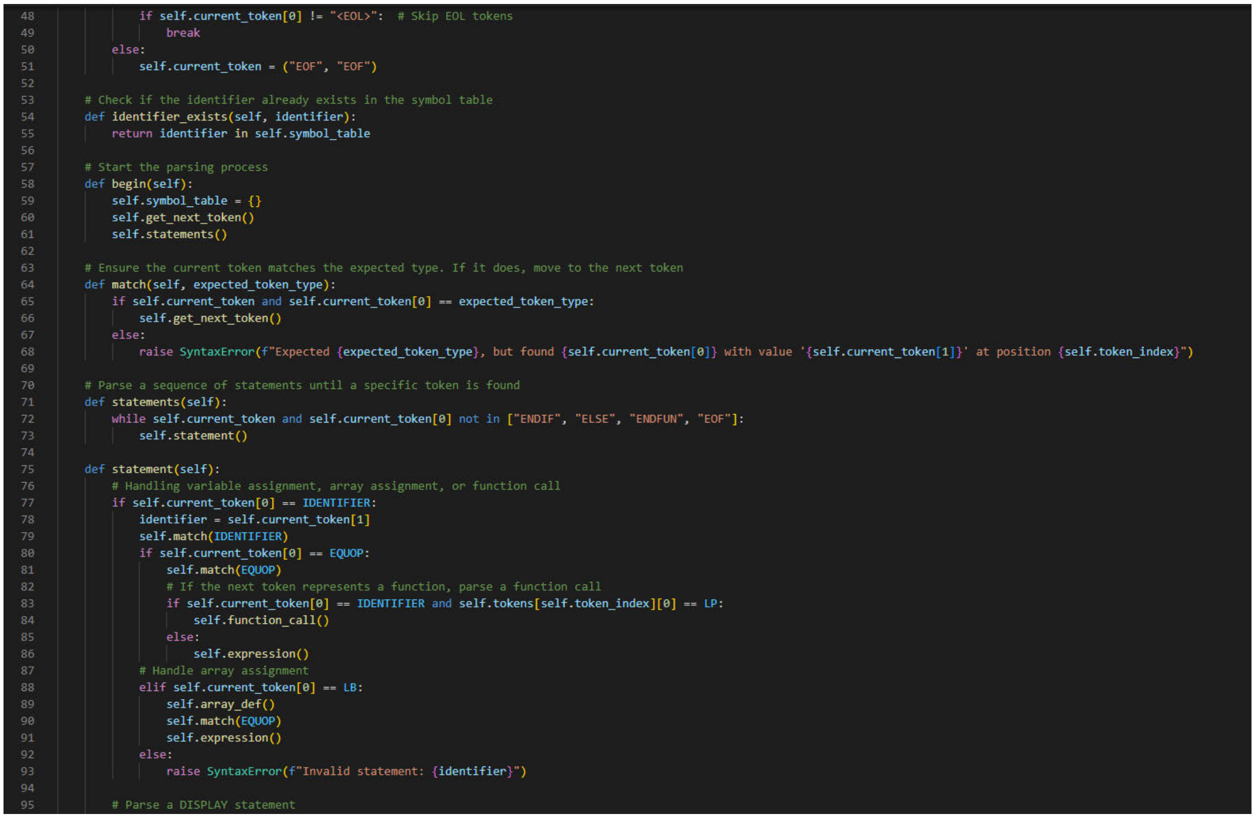


```python
                break
        # if no match is found increment position
        if not match:
            position += 1

    return tokens  # returns the list of tokens

# Main function to execute the program
def main():
    import sys
    if len(sys.argv) != 2: # check for the correct number of command line arguments
        print("Usage: python scl_scanner.py <filename>")
        return
    # read source code from file
    with open(sys.argv[1], 'r') as f:
        source_code = f.read()
    # tokenize source code
    tokens = tokenize(source_code)
    # print each token
    for token in tokens:
        print(token)
    # Save tokens to JSON
    with open("tokens.json", "w") as outfile:
        json.dump(tokens, outfile)

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

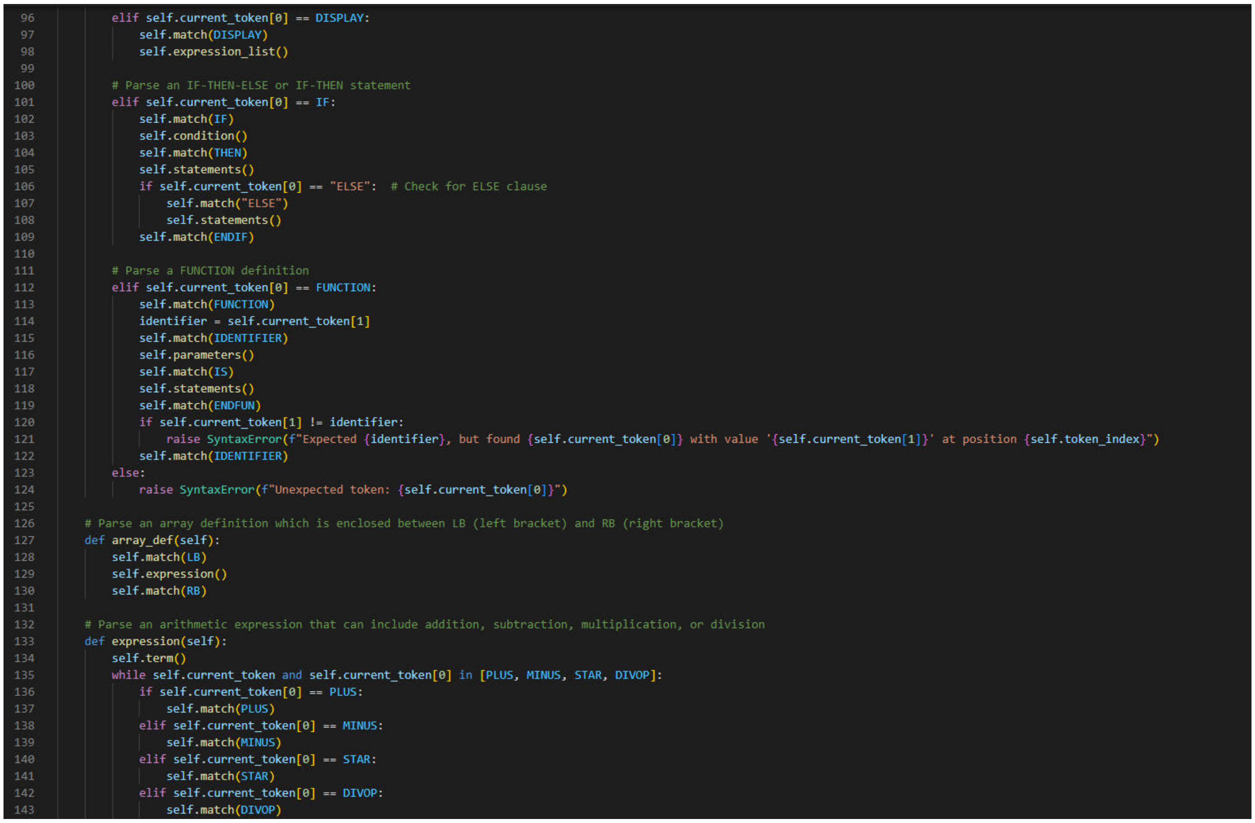
scl_parser.py Source Code:



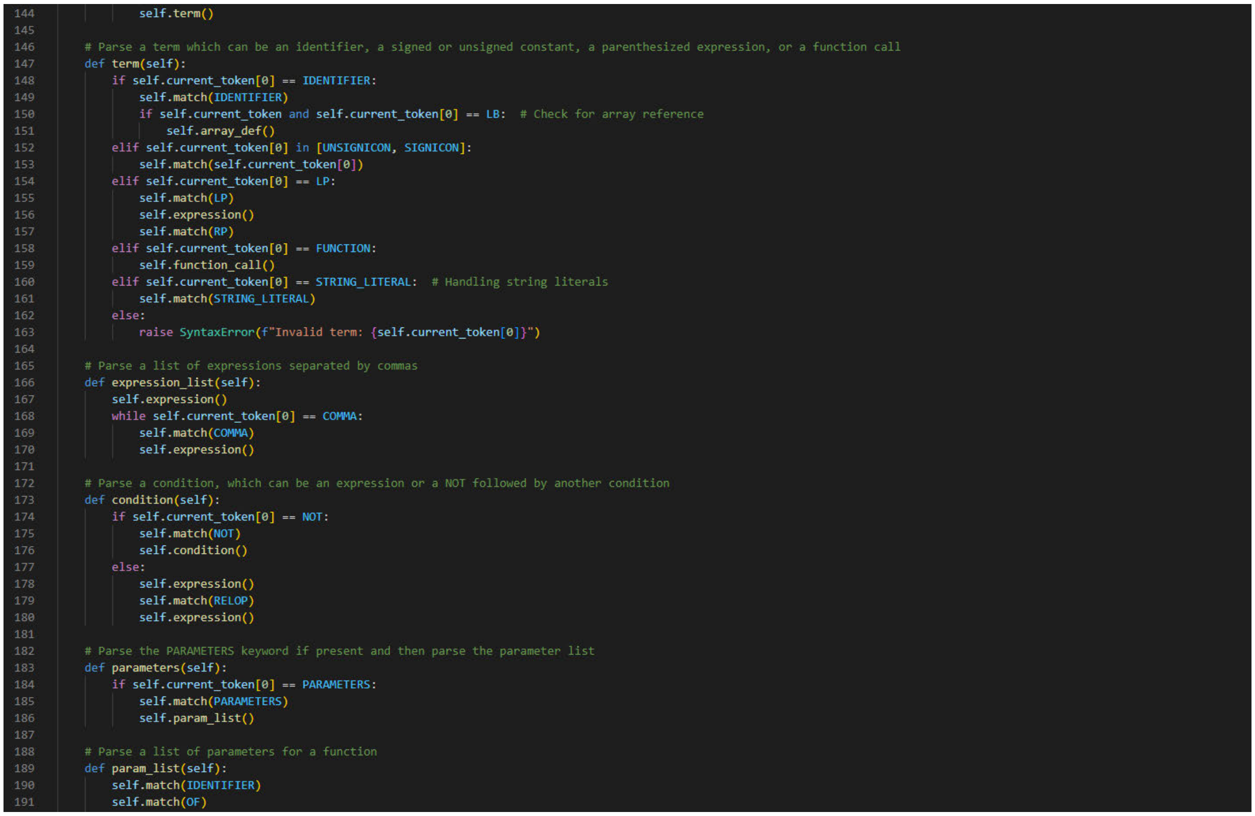
```python
import json

# Define constants for token types
IDENTIFIER = "IDENTIFIER"
UNSIGNICON = "UNSIGNICON"
SIGNICON = "SIGNICON"
PLUS = "PLUS"
MINUS = "MINUS"
STAR = "STAR"
DIVOP = "DIVOP"
EQUOP = "EQUOP"
RELOP = "RELOP"
LB = "LB"
RB = "RB"
LP = "LP"
RP = "RP"
COMMA = "COMMA"
OF = "OF"
DISPLAY = "DISPLAY"
IF = "IF"
THEN = "THEN"
ENDIF = "ENDIF"
FUNCTION = "FUNCTION"
IS = "IS"
ENDFUN = "ENDFUN"
PARAMETERS = "PARAMETERS"
INTEGER = "INTEGER"
FLOAT = "FLOAT"
CHAR = "CHAR"
NOT = "NOT"
STRING_LITERAL = "STRING_LITERAL"

# List of keywords in the SCL language
KEYWORDS = [DISPLAY, IF, THEN, ENDIF, FUNCTION, IS, ENDFUN, PARAMETERS, INTEGER, FLOAT, CHAR, NOT]

# Parser class
jschul37, 2 days ago | 1 author (jschul37)
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.current_token = None
        self.token_index = 0

    # Retrieve the next token from the token list, skipping EOLs
    def get_next_token(self):
        while self.token_index < len(self.tokens):
            self.current_token = self.tokens[self.token_index]
            self.token_index += 1
```



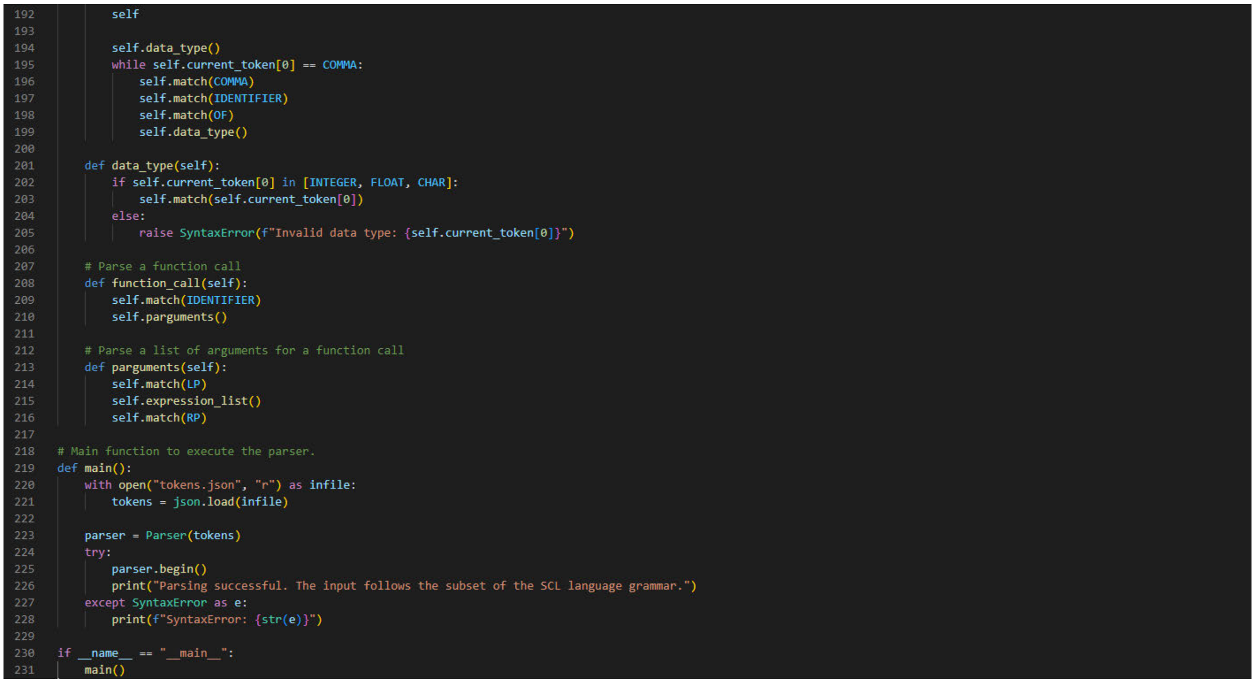
```python
            if self.current_token[0] != "<EOL>":  # Skip EOL tokens
                break
        else:
            self.current_token = ("EOF", "EOF")

    # Check if the identifier already exists in the symbol table
    def identifier_exists(self, identifier):
        return identifier in self.symbol_table

    # Start the parsing process
    def begin(self):
        self.symbol_table = {}
        self.get_next_token()
        self.statements()

    # Ensure the current token matches the expected type. If it does, move to the next token
    def match(self, expected_token_type):
        if self.current_token and self.current_token[0] == expected_token_type:
            self.get_next_token()
        else:
            raise SyntaxError(f"Expected {expected_token_type}, but found {self.current_token[0]} with value '{self.current_token[1]}' at position {self.token_index}")

    # Parse a sequence of statements until a specific token is found
    def statements(self):
        while self.current_token and self.current_token[0] not in ["ENDIF", "ELSE", "ENDFUN", "EOF"]:
            self.statement()

    def statement(self):
        # Handling variable assignment, array assignment, or function call
        if self.current_token[0] == IDENTIFIER:
            identifier = self.current_token[1]
            self.match(IDENTIFIER)
            if self.current_token[0] == EQUOP:
                self.match(EQUOP)
                # If the next token represents a function, parse a function call
                if self.current_token[0] == IDENTIFIER and self.tokens[self.token_index][0] == LP:
                    self.function_call()
                else:
                    self.expression()
            # Handle array assignment
            elif self.current_token[0] == LB:
                self.array_def()
                self.match(EQUOP)
                self.expression()
            else:
                raise SyntaxError(f"Invalid statement: {identifier}")

        # Parse a DISPLAY statement
```



```python
        elif self.current_token[0] == DISPLAY:
            self.match(DISPLAY)
            self.expression_list()

        # Parse an IF-THEN-ELSE or IF-THEN statement
        elif self.current_token[0] == IF:
            self.match(IF)
            self.condition()
            self.match(THEN)
            self.statements()
            if self.current_token[0] == "ELSE":  # Check for ELSE clause
                self.match("ELSE")
                self.statements()
            self.match(ENDIF)

        # Parse a FUNCTION definition
        elif self.current_token[0] == FUNCTION:
            self.match(FUNCTION)
            identifier = self.current_token[1]
            self.match(IDENTIFIER)
            self.parameters()
            self.match(IS)
            self.statements()
            self.match(ENDFUN)
            if self.current_token[1] != identifier:
                raise SyntaxError(f"Expected {identifier}, but found {self.current_token[0]} with value '{self.current_token[1]}' at position {self.token_index}")
            self.match(IDENTIFIER)
        else:
            raise SyntaxError(f"Unexpected token: {self.current_token[0]}")

    # Parse an array definition which is enclosed between LB (left bracket) and RB (right bracket)
    def array_def(self):
        self.match(LB)
        self.expression()
        self.match(RB)

    # Parse an arithmetic expression that can include addition, subtraction, multiplication, or division
    def expression(self):
        self.term()
        while self.current_token and self.current_token[0] in [PLUS, MINUS, STAR, DIVOP]:
            if self.current_token[0] == PLUS:
                self.match(PLUS)
            elif self.current_token[0] == MINUS:
                self.match(MINUS)
            elif self.current_token[0] == STAR:
                self.match(STAR)
            elif self.current_token[0] == DIVOP:
                self.match(DIVOP)
```

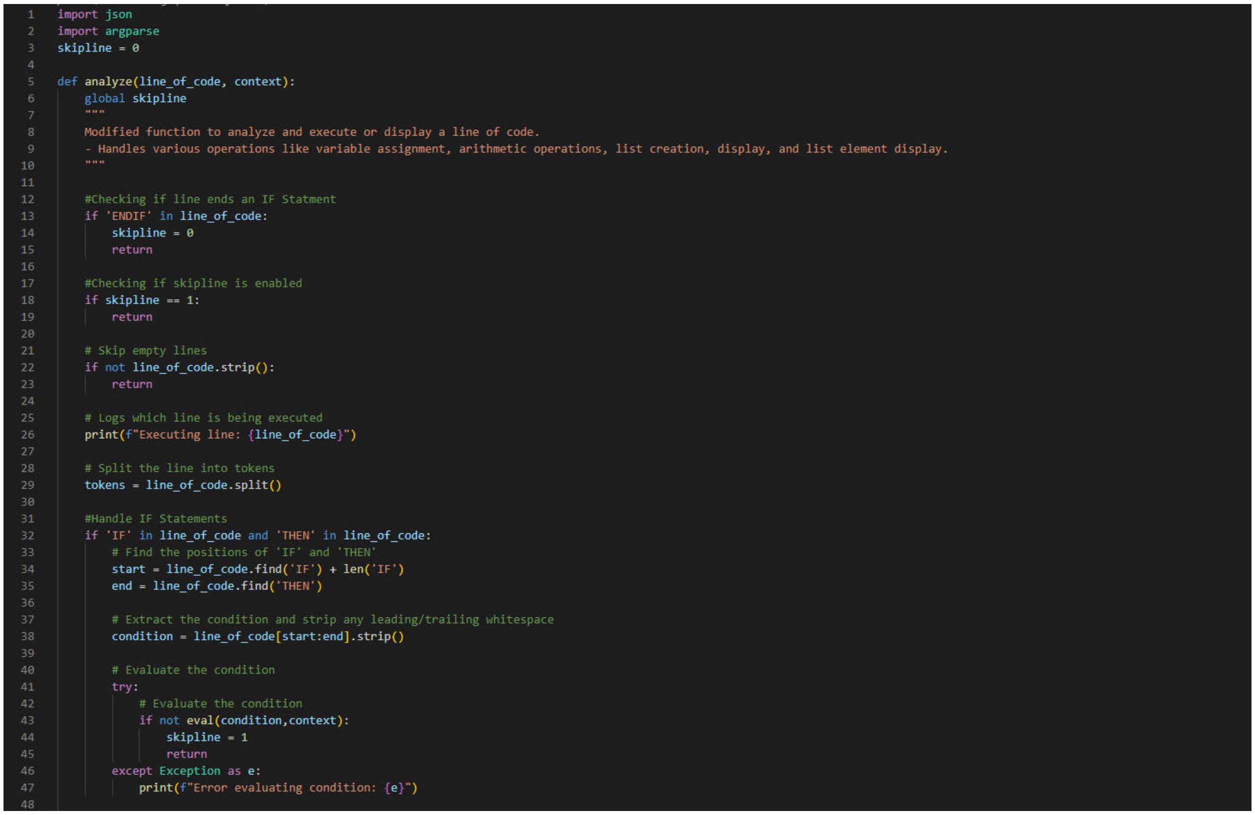


```python
            self.term()

    # Parse a term which can be an identifier, a signed or unsigned constant, a parenthesized expression, or a function call
    def term(self):
        if self.current_token[0] == IDENTIFIER:
            self.match(IDENTIFIER)
            if self.current_token and self.current_token[0] == LB:  # Check for array reference
                self.array_def()
        elif self.current_token[0] in [UNSIGNICON, SIGNICON]:
            self.match(self.current_token[0])
        elif self.current_token[0] == LP:
            self.match(LP)
            self.expression()
            self.match(RP)
        elif self.current_token[0] == FUNCTION:
            self.function_call()
        elif self.current_token[0] == STRING_LITERAL:  # Handling string literals
            self.match(STRING_LITERAL)
        else:
            raise SyntaxError(f"Invalid term: {self.current_token[0]}")

    # Parse a list of expressions separated by commas
    def expression_list(self):
        self.expression()
        while self.current_token[0] == COMMA:
            self.match(COMMA)
            self.expression()

    # Parse a condition, which can be an expression or a NOT followed by another condition
    def condition(self):
        if self.current_token[0] == NOT:
            self.match(NOT)
            self.condition()
        else:
            self.expression()
            self.match(RELOP)
            self.expression()

    # Parse the PARAMETERS keyword if present and then parse the parameter list
    def parameters(self):
        if self.current_token[0] == PARAMETERS:
            self.match(PARAMETERS)
            self.param_list()

    # Parse a list of parameters for a function
    def param_list(self):
        self.match(IDENTIFIER)
        self.match(OF)
```

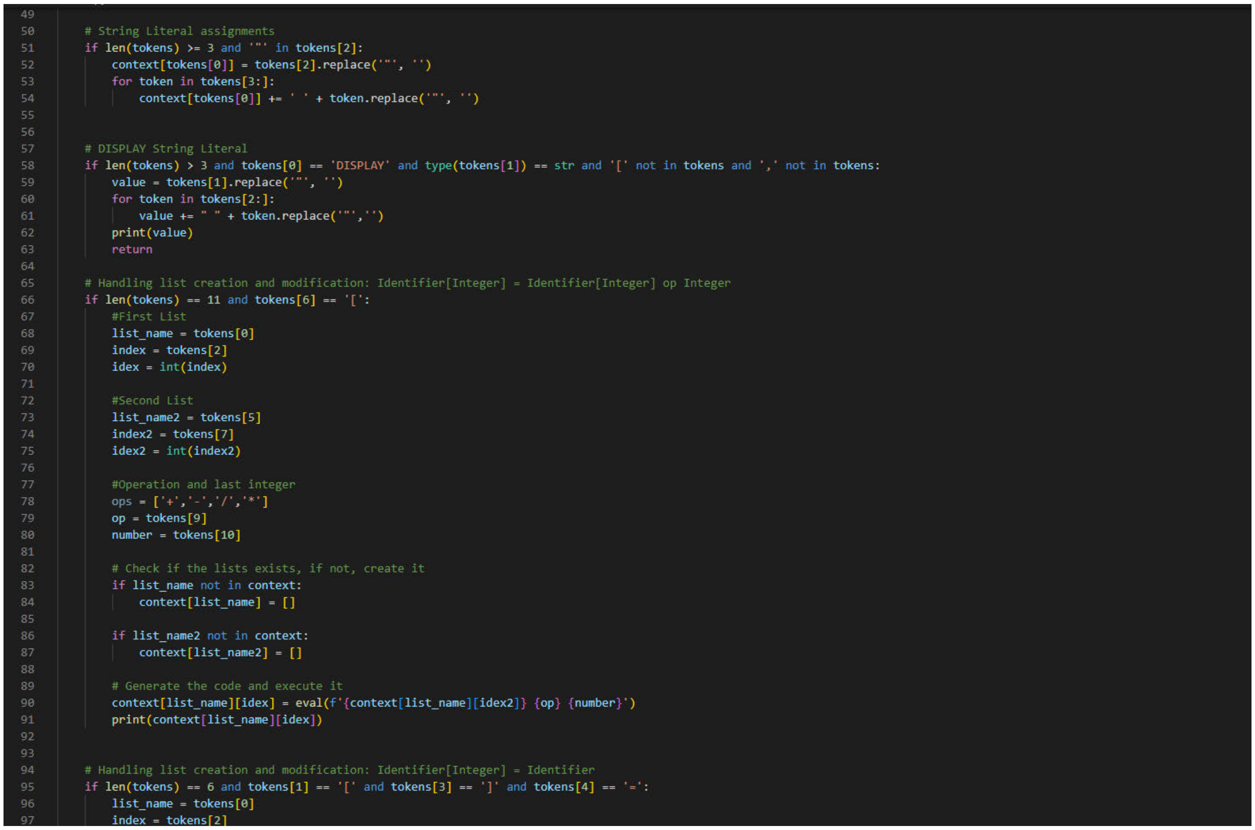


```python
        self

        self.data_type()
        while self.current_token[0] == COMMA:
            self.match(COMMA)
            self.match(IDENTIFIER)
            self.match(OF)
            self.data_type()

    def data_type(self):
        if self.current_token[0] in [INTEGER, FLOAT, CHAR]:
            self.match(self.current_token[0])
        else:
            raise SyntaxError(f"Invalid data type: {self.current_token[0]}")

    # Parse a function call
    def function_call(self):
        self.match(IDENTIFIER)
        self.parguments()

    # Parse a list of arguments for a function call
    def parguments(self):
        self.match(LP)
        self.expression_list()
        self.match(RP)

# Main function to execute the parser.
def main():
    with open("tokens.json", "r") as infile:
        tokens = json.load(infile)

    parser = Parser(tokens)
    try:
        parser.begin()
        print("Parsing successful. The input follows the subset of the SCL language grammar.")
    except SyntaxError as e:
        print(f"SyntaxError: {str(e)}")

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

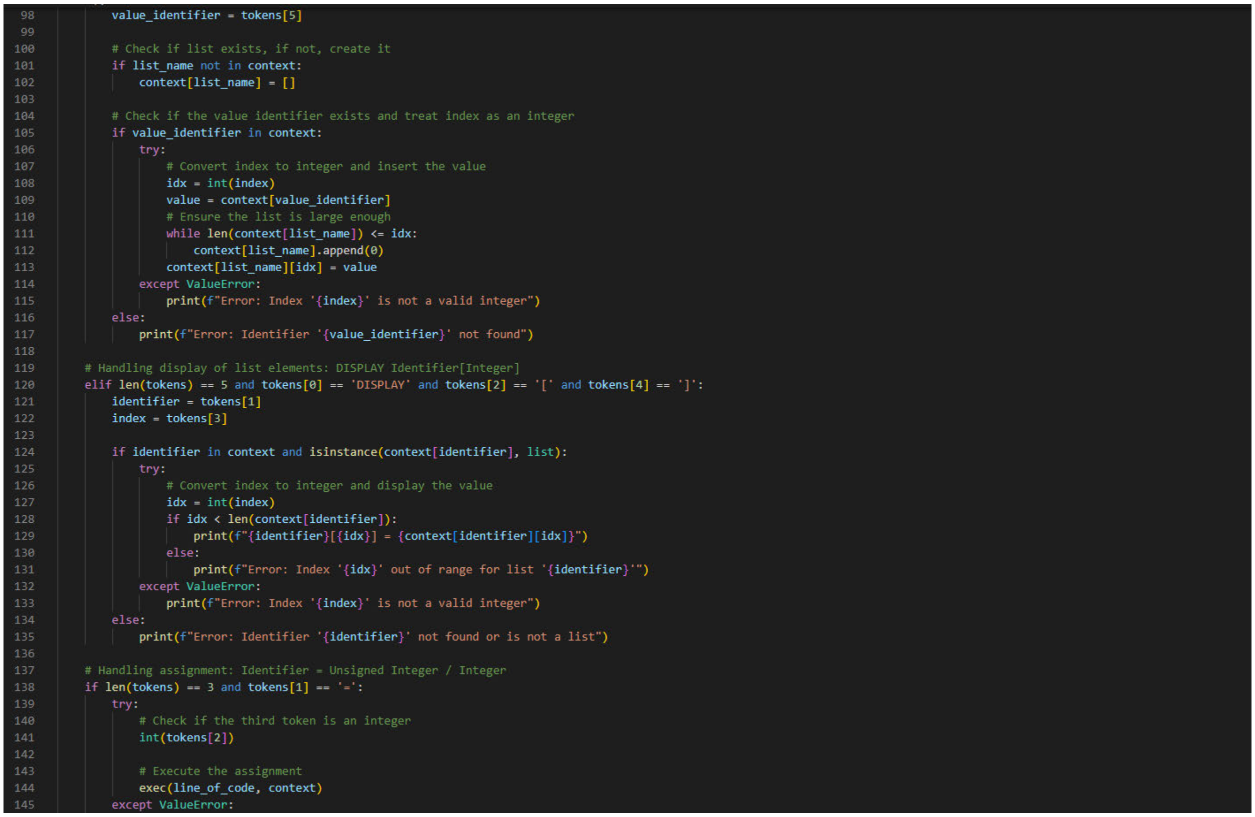
scl_executer.py Source Code:



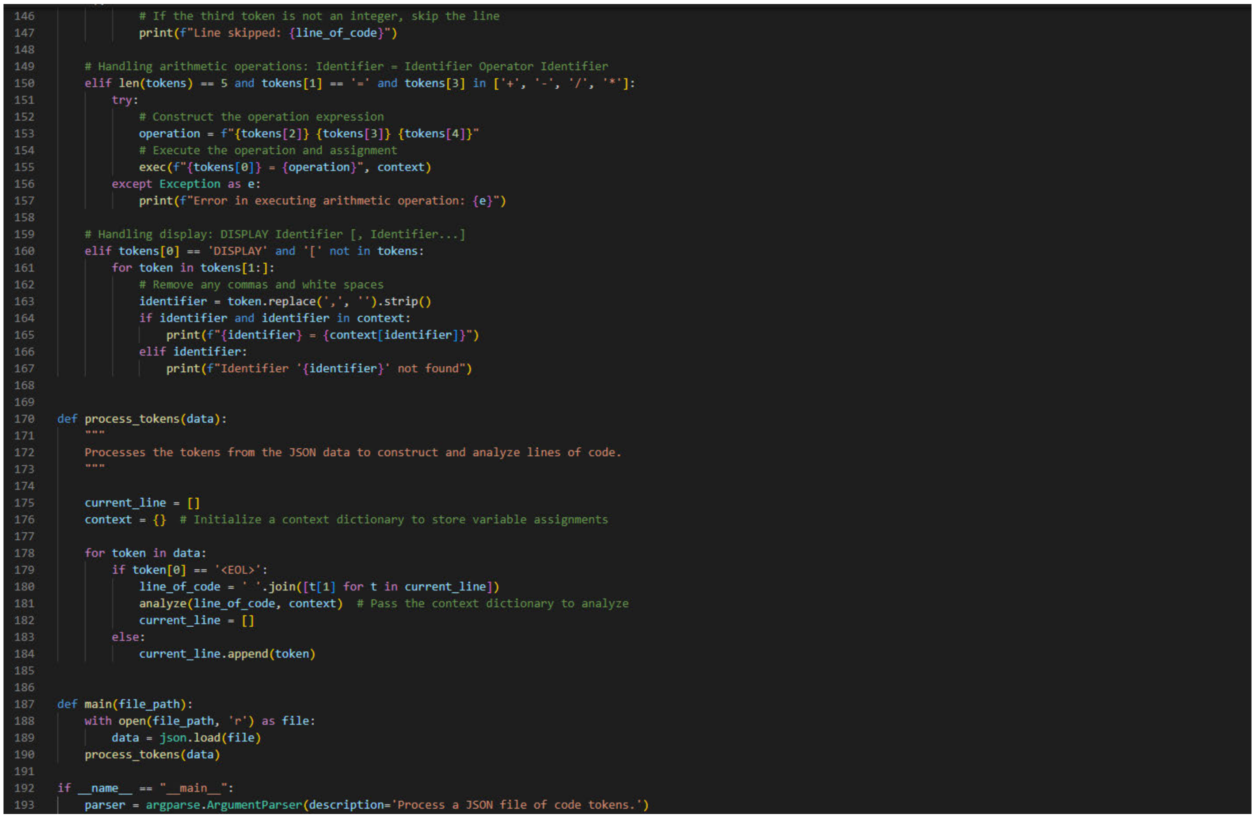
```python
import json
import argparse
skipline = 0

def analyze(line_of_code, context):
    global skipline
    """
    Modified function to analyze and execute or display a line of code.
    - Handles various operations like variable assignment, arithmetic operations, list creation, display, and list element display.
    """

    #Checking if line ends an IF Statment
    if 'ENDIF' in line_of_code:
        skipline = 0
        return

    #Checking if skipline is enabled
    if skipline == 1:
        return

    # Skip empty lines
    if not line_of_code.strip():
        return

    # Logs which line is being executed
    print(f"Executing line: {line_of_code}")

    # Split the line into tokens
    tokens = line_of_code.split()

    #Handle IF Statements
    if 'IF' in line_of_code and 'THEN' in line_of_code:
        # Find the positions of 'IF' and 'THEN'
        start = line_of_code.find('IF') + len('IF')
        end = line_of_code.find('THEN')

        # Extract the condition and strip any leading/trailing whitespace
        condition = line_of_code[start:end].strip()

        # Evaluate the condition
        try:
            # Evaluate the condition
            if not eval(condition,context):
                skipline = 1
                return
        except Exception as e:
            print(f"Error evaluating condition: {e}")

```



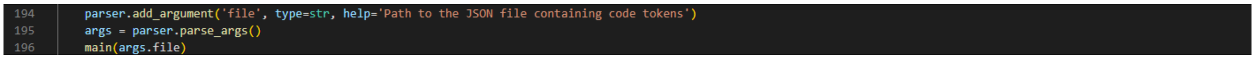
```
    # String Literal assignments
    if len(tokens) >= 3 and '"' in tokens[2]:
        context[tokens[0]] = tokens[2].replace('"', '')
        for token in tokens[3:]:
            context[tokens[0]] += ' ' + token.replace('"', '')


    # DISPLAY String Literal
    if len(tokens) > 3 and tokens[0] == 'DISPLAY' and type(tokens[1]) == str and '[' not in tokens and ',' not in tokens:
        value = tokens[1].replace('"', '')
        for token in tokens[2:]:
            value += " " + token.replace('"','')
        print(value)
        return

    # Handling list creation and modification: Identifier[Integer] = Identifier[Integer] op Integer
    if len(tokens) == 11 and tokens[6] == '[':
        #First List
        list_name = tokens[0]
        index = tokens[2]
        idex = int(index)

        #Second List
        list_name2 = tokens[5]
        index2 = tokens[7]
        idex2 = int(index2)

        #Operation and last integer
        ops = ['+','-','/','*']
        op = tokens[9]
        number = tokens[10]

        # Check if the lists exists, if not, create it
        if list_name not in context:
            context[list_name] = []

        if list_name2 not in context:
            context[list_name2] = []

        # Generate the code and execute it
        context[list_name][idex] = eval(f'{context[list_name][idex2]} {op} {number}')
        print(context[list_name][idex])


    # Handling list creation and modification: Identifier[Integer] = Identifier
    if len(tokens) == 6 and tokens[1] == '[' and tokens[3] == ']' and tokens[4] == '=':
        list_name = tokens[0]
        index = tokens[2]
```



```
        value_identifier = tokens[5]

        # Check if list exists, if not, create it
        if list_name not in context:
            context[list_name] = []

        # Check if the value identifier exists and treat index as an integer
        if value_identifier in context:
            try:
                # Convert index to integer and insert the value
                idx = int(index)
                value = context[value_identifier]
                # Ensure the list is large enough
                while len(context[list_name]) <= idx:
                    context[list_name].append(0)
                context[list_name][idx] = value
            except ValueError:
                print(f"Error: Index '{index}' is not a valid integer")
        else:
            print(f"Error: Identifier '{value_identifier}' not found")

    # Handling display of list elements: DISPLAY Identifier[Integer]
    elif len(tokens) == 5 and tokens[0] == 'DISPLAY' and tokens[2] == '[' and tokens[4] == ']':
        identifier = tokens[1]
        index = tokens[3]

        if identifier in context and isinstance(context[identifier], list):
            try:
                # Convert index to integer and display the value
                idx = int(index)
                if idx < len(context[identifier]):
                    print(f"{identifier}[{idx}] = {context[identifier][idx]}")
                else:
                    print(f"Error: Index '{idx}' out of range for list '{identifier}'")
            except ValueError:
                print(f"Error: Index '{index}' is not a valid integer")
        else:
            print(f"Error: Identifier '{identifier}' not found or is not a list")

    # Handling assignment: Identifier = Unsigned Integer / Integer
    if len(tokens) == 3 and tokens[1] == '=':
        try:
            # Check if the third token is an integer
            int(tokens[2])

            # Execute the assignment
            exec(line_of_code, context)
        except ValueError:
```



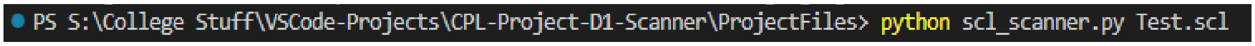
```python
            # If the third token is not an integer, skip the line
            print(f"Line skipped: {line_of_code}")

    # Handling arithmetic operations: Identifier = Identifier Operator Identifier
    elif len(tokens) == 5 and tokens[1] == '=' and tokens[3] in ['+', '-', '/', '*']:
        try:
            # Construct the operation expression
            operation = f"{tokens[2]} {tokens[3]} {tokens[4]}"
            # Execute the operation and assignment
            exec(f"{tokens[0]} = {operation}", context)
        except Exception as e:
            print(f"Error in executing arithmetic operation: {e}")

    # Handling display: DISPLAY Identifier [, Identifier...]
    elif tokens[0] == 'DISPLAY' and '[' not in tokens:
        for token in tokens[1:]:
            # Remove any commas and white spaces
            identifier = token.replace(',', '').strip()
            if identifier and identifier in context:
                print(f"{identifier} = {context[identifier]}")
            elif identifier:
                print(f"Identifier '{identifier}' not found")


def process_tokens(data):
    """
    Processes the tokens from the JSON data to construct and analyze lines of code.
    """

    current_line = []
    context = {}  # Initialize a context dictionary to store variable assignments

    for token in data:
        if token[0] == '<EOL>':
            line_of_code = ' '.join([t[1] for t in current_line])
            analyze(line_of_code, context)  # Pass the context dictionary to analyze
            current_line = []
        else:
            current_line.append(token)


def main(file_path):
    with open(file_path, 'r') as file:
        data = json.load(file)
    process_tokens(data)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Process a JSON file of code tokens.')
```



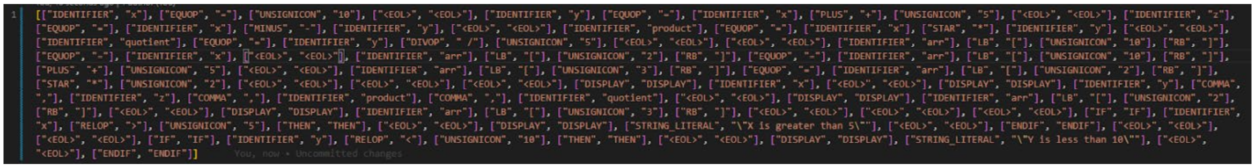
```python
    parser.add_argument('file', type=str, help='Path to the JSON file containing code tokens')
    args = parser.parse_args()
    main(args.file)
```

Source Code Execution Screenshot

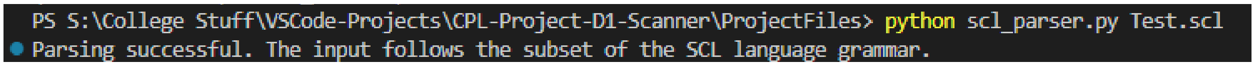
Test.scl → tokens.json File Source Code (is also printed in the Terminal after scanner is called):



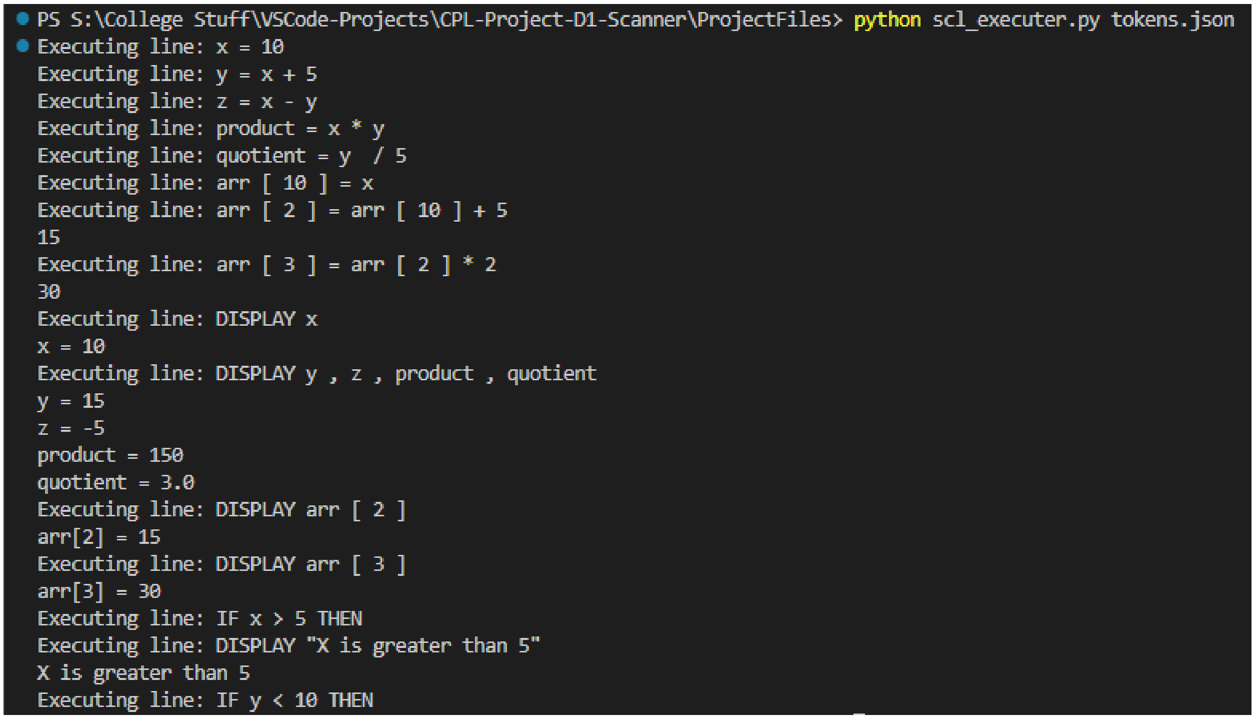
```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_scanner.py Test.scl
```



```
[["IDENTIFIER", "x"], ["EQUOP", "="], ["UNSIGNICON", "10"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "z"],
["EQUOP", "="], ["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "product"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["STAR", "*"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"],
["IDENTIFIER", "quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["UNSIGNICON", "5"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"],
["EQUOP", "="], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"],
["PLUS", "+"], ["UNSIGNICON", "5"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"],
["STAR", "*"], ["UNSIGNICON", "2"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "y"], ["COMMA",
","], ["IDENTIFIER", "z"], ["COMMA", ","], ["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"],
["RB", "]"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IF", "IF"], ["IDENTIFIER",
"x"], ["RELOP", ">"], ["UNSIGNICON", "5"], ["THEN", "THEN"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is greater than 5\""], ["<EOL>", "<EOL>"], ["ENDIF", "ENDIF"], ["<EOL>", "<EOL>"],
["<EOL>", "<EOL>"], ["IF", "IF"], ["IDENTIFIER", "y"], ["RELOP", "<"], ["UNSIGNICON", "10"], ["THEN", "THEN"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is less than 10\""], ["<EOL>",
"<EOL>"], ["ENDIF", "ENDIF"]]    You, now • Uncommitted changes
```

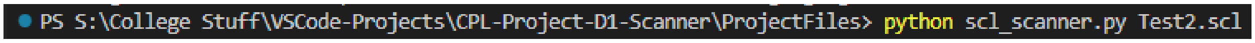


```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_parser.py Test.scl
Parsing successful. The input follows the subset of the SCL language grammar.
```

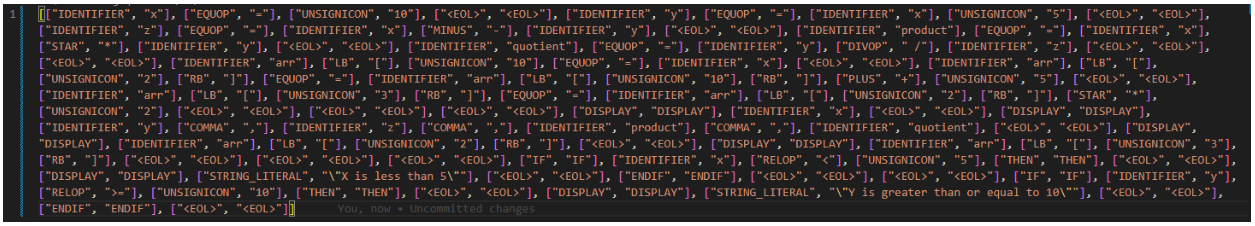


```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_executer.py tokens.json
Executing line: x = 10
Executing line: y = x + 5
Executing line: z = x - y
Executing line: product = x * y
Executing line: quotient = y  / 5
Executing line: arr [ 10 ] = x
Executing line: arr [ 2 ] = arr [ 10 ] + 5
15
Executing line: arr [ 3 ] = arr [ 2 ] * 2
30
Executing line: DISPLAY x
x = 10
Executing line: DISPLAY y , z , product , quotient
y = 15
z = -5
product = 150
quotient = 3.0
Executing line: DISPLAY arr [ 2 ]
arr[2] = 15
Executing line: DISPLAY arr [ 3 ]
arr[3] = 30
Executing line: IF x > 5 THEN
Executing line: DISPLAY "X is greater than 5"
X is greater than 5
Executing line: IF y < 10 THEN
```

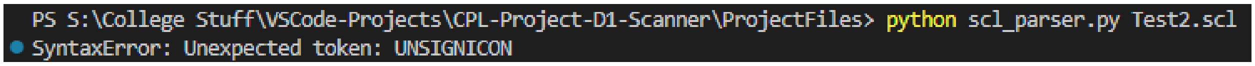
Test2.scl → tokens.json File Source Code (is also printed in the Terminal after scanner is called):



```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_scanner.py Test2.scl
```

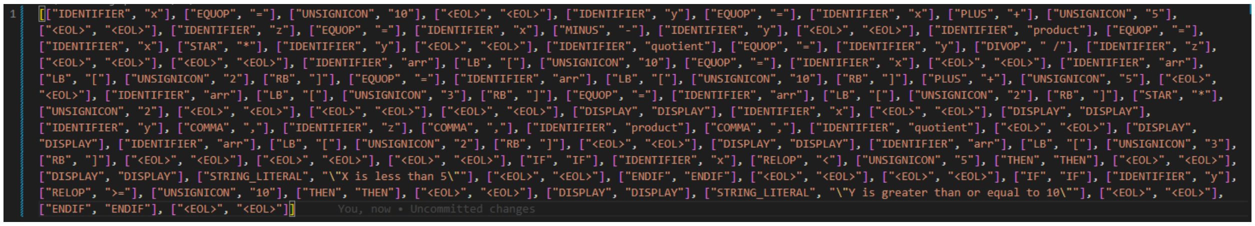


```
[["IDENTIFIER", "x"], ["EQUOP", "="], ["UNSIGNICON", "10"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["UNSIGNICON", "5"], ["<EOL>", "<EOL>"],
["IDENTIFIER", "z"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "product"], ["EQUOP", "="], ["IDENTIFIER", "x"],
["STAR", "*"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["IDENTIFIER", "z"], ["<EOL>", "<EOL>"],
["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["],
["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["<EOL>", "<EOL>"],
["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["STAR", "*"],
["UNSIGNICON", "2"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"],
["IDENTIFIER", "y"], ["COMMA", ","], ["IDENTIFIER", "z"], ["COMMA", ","], ["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["<EOL>", "<EOL>"], ["DISPLAY",
"DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"],
["RB", "]"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IF", "IF"], ["IDENTIFIER", "x"], ["RELOP", "<"], ["UNSIGNICON", "5"], ["THEN", "THEN"], ["<EOL>", "<EOL>"],
["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is less than 5\""], ["<EOL>", "<EOL>"], ["ENDIF", "ENDIF"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IF", "IF"], ["IDENTIFIER", "y"],
["RELOP", ">="], ["UNSIGNICON", "10"], ["THEN", "THEN"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is greater than or equal to 10\""], ["<EOL>", "<EOL>"],
["ENDIF", "ENDIF"], ["<EOL>", "<EOL>"]]        You, now • Uncommitted changes
```

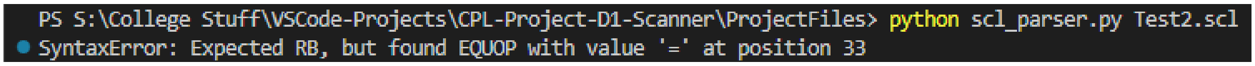


```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_parser.py Test2.scl
SyntaxError: Unexpected token: UNSIGNICON
```

After Fixing First Syntax Error:

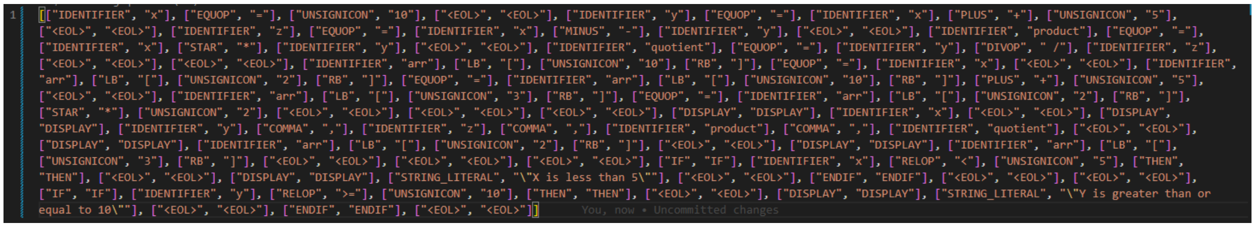


```
[["IDENTIFIER", "x"], ["EQUOP", "="], ["UNSIGNICON", "10"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "5"],
["<EOL>", "<EOL>"], ["IDENTIFIER", "z"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "product"], ["EQUOP", "="],
["IDENTIFIER", "x"], ["STAR", "*"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["IDENTIFIER", "z"],
["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"],
["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["<EOL>",
"<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["STAR", "*"],
["UNSIGNICON", "2"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"],
["IDENTIFIER", "y"], ["COMMA", ","], ["IDENTIFIER", "z"], ["COMMA", ","], ["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["<EOL>", "<EOL>"], ["DISPLAY",
"DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"],
["RB", "]"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IF", "IF"], ["IDENTIFIER", "x"], ["RELOP", "<"], ["UNSIGNICON", "5"], ["THEN", "THEN"], ["<EOL>", "<EOL>"],
["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is less than 5\""], ["<EOL>", "<EOL>"], ["ENDIF", "ENDIF"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IF", "IF"], ["IDENTIFIER", "y"],
["RELOP", ">="], ["UNSIGNICON", "10"], ["THEN", "THEN"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is greater than or equal to 10\""], ["<EOL>", "<EOL>"],
["ENDIF", "ENDIF"], ["<EOL>", "<EOL>"]]        You, now • Uncommitted changes
```

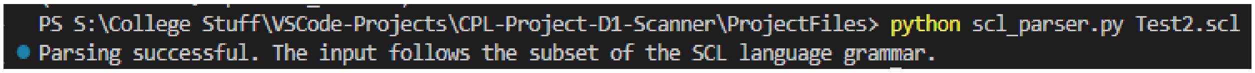


```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_parser.py Test2.scl
SyntaxError: Expected RB, but found EQUOP with value '=' at position 33
```

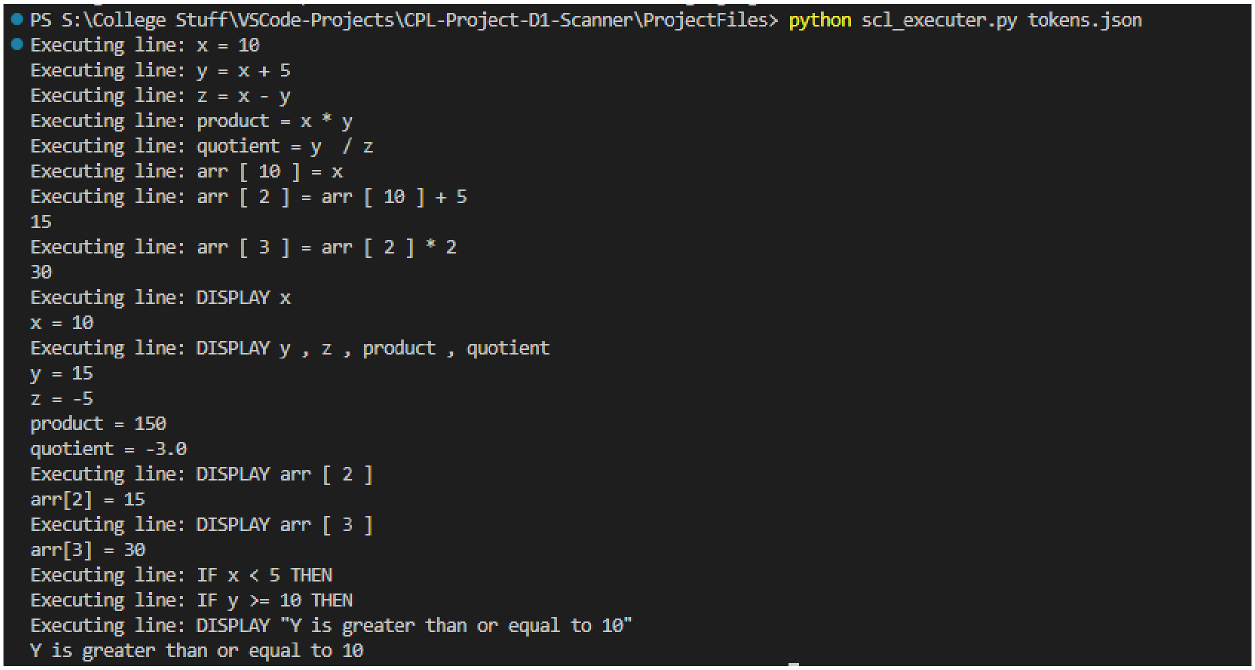
After Fixing Second Syntax Error:



```
[["IDENTIFIER", "x"], ["EQUOP", "="], ["UNSIGNICON", "10"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "5"],
["<EOL>", "<EOL>"], ["IDENTIFIER", "z"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "product"], ["EQUOP", "="],
["IDENTIFIER", "x"], ["STAR", "*"], ["IDENTIFIER", "y"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["IDENTIFIER", "z"],
["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["IDENTIFIER",
"arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"], ["PLUS", "+"], ["UNSIGNICON", "5"],
["<EOL>", "<EOL>"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"],
["STAR", "*"], ["UNSIGNICON", "2"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["<EOL>", "<EOL>"], ["DISPLAY",
"DISPLAY"], ["IDENTIFIER", "y"], ["COMMA", ","], ["IDENTIFIER", "z"], ["COMMA", ","], ["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["<EOL>", "<EOL>"],
["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["],
["UNSIGNICON", "3"], ["RB", "]"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IF", "IF"], ["IDENTIFIER", "x"], ["RELOP", "<"], ["UNSIGNICON", "5"], ["THEN",
"THEN"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is less than 5\""], ["<EOL>", "<EOL>"], ["ENDIF", "ENDIF"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"],
["IF", "IF"], ["IDENTIFIER", "y"], ["RELOP", ">="], ["UNSIGNICON", "10"], ["THEN", "THEN"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is greater than or
equal to 10\""], ["<EOL>", "<EOL>"], ["ENDIF", "ENDIF"], ["<EOL>", "<EOL>"]]        You, now • Uncommitted changes
```

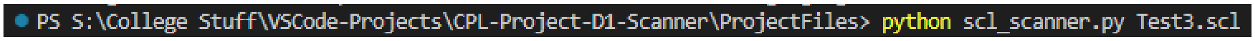


```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_parser.py Test2.scl
Parsing successful. The input follows the subset of the SCL language grammar.
```

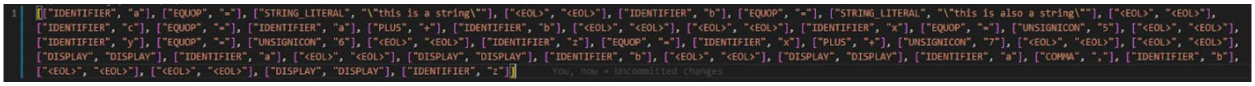


```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_executer.py tokens.json
Executing line: x = 10
Executing line: y = x + 5
Executing line: z = x - y
Executing line: product = x * y
Executing line: quotient = y  / z
Executing line: arr [ 10 ] = x
Executing line: arr [ 2 ] = arr [ 10 ] + 5
15
Executing line: arr [ 3 ] = arr [ 2 ] * 2
30
Executing line: DISPLAY x
x = 10
Executing line: DISPLAY y , z , product , quotient
y = 15
z = -5
product = 150
quotient = -3.0
Executing line: DISPLAY arr [ 2 ]
arr[2] = 15
Executing line: DISPLAY arr [ 3 ]
arr[3] = 30
Executing line: IF x < 5 THEN
Executing line: IF y >= 10 THEN
Executing line: DISPLAY "Y is greater than or equal to 10"
Y is greater than or equal to 10
```

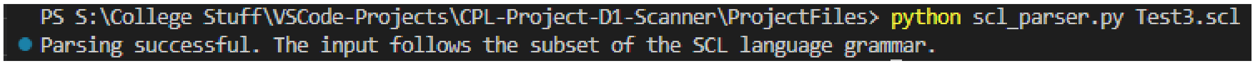
Test3.scl → tokens.json File Source Code (is also printed in the Terminal after scanner is called):



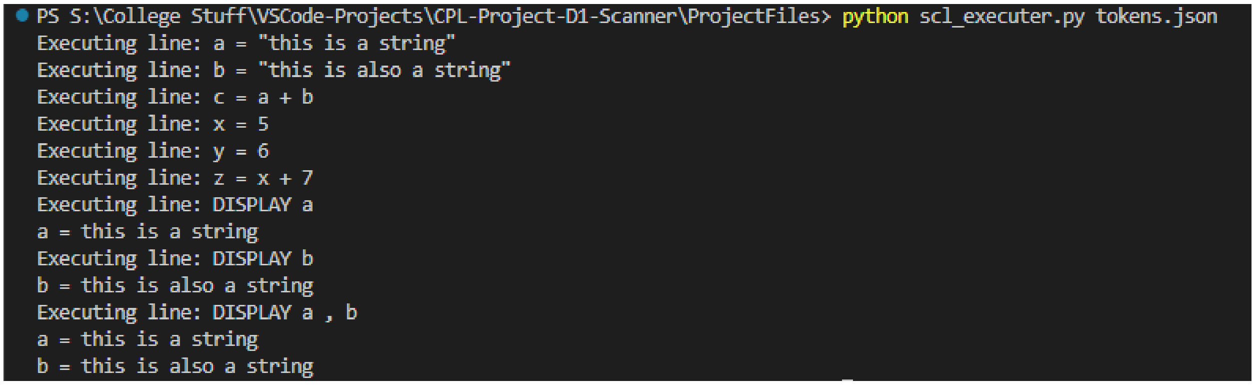
```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_scanner.py Test3.scl
```



```
[["IDENTIFIER", "a"], ["EQUOP", "="], ["STRING_LITERAL", "\"this is a string\""], ["<EOL>", "<EOL>"], ["IDENTIFIER", "b"], ["EQUOP", "="], ["STRING_LITERAL", "\"this is also a string\""], ["<EOL>", "<EOL>"],
["IDENTIFIER", "c"], ["EQUOP", "="], ["IDENTIFIER", "a"], ["PLUS", "+"], ["IDENTIFIER", "b"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "x"], ["EQUOP", "="], ["UNSIGNICON", "5"], ["<EOL>", "<EOL>"],
["IDENTIFIER", "y"], ["EQUOP", "="], ["UNSIGNICON", "6"], ["<EOL>", "<EOL>"], ["IDENTIFIER", "z"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "7"], ["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"],
["DISPLAY", "DISPLAY"], ["IDENTIFIER", "a"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "b"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "a"], ["COMMA", ","], ["IDENTIFIER", "b"],
["<EOL>", "<EOL>"], ["<EOL>", "<EOL>"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "z"]]     You, now • Uncommitted changes
```



```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_parser.py Test3.scl
Parsing successful. The input follows the subset of the SCL language grammar.
```



```
PS S:\College Stuff\VSCode-Projects\CPL-Project-D1-Scanner\ProjectFiles> python scl_executer.py tokens.json
Executing line: a = "this is a string"
Executing line: b = "this is also a string"
Executing line: c = a + b
Executing line: x = 5
Executing line: y = 6
Executing line: z = x + 7
Executing line: DISPLAY a
a = this is a string
Executing line: DISPLAY b
b = this is also a string
Executing line: DISPLAY a , b
a = this is a string
b = this is also a string
```

Comments and Conclusion

The program successfully interprets a ‘tokens.json’ file, parsed via the ‘argparse’ module which processes a JSON file of the code’s tokens. The interpreter manages ‘IF’ statements using a global variable ‘skipline’ to skiplines based on specific conditions. The ‘analyze’ function splits each line into tokens and performs actions, such as processing string literals, handling list operations and displaying variables and elements.

References

Sebesta, R. W. (2012). *Concepts of Programming Languages* (10th ed.). Pearson.