Kennesaw State University

College of Computing and Software Engineering

Department of Computer Science

CS 4308-W02: Concepts of Programming Languages

Deliverable 2 - Parser

Jesse Schultheis [redacted]

Katlin Scott [redacted]

William Stigall [redacted]

Hailey Walker [redacted]

October 11, 2023

Initial Problem Statement

The problem is to develop a parser program for a subset of the SCL language, working in conjunction with a previously developed scanner program, with the goal of demonstrating an understanding of the parsing process of compilation, including writing BNF-based parsing rules and ensuring the parser recognizes statements and identifiers while implementing three public functions: getNextToken(), identifierExists(string identifier), and begin().

Summary/Purpose

The purpose of this deliverable is to implement a Parser file that uses the previously implemented scanner.py file (from Deliverable 1) and its output tokens.json file and then goes through and checks the validity of its syntactical structure.

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.

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

The Parser halts upon generation of a SyntaxError, meaning that only the first SyntaxError will be raised, and the remaining will go undetected since the parser has halted. It is also only designed to support the limited subset of grammar that we used in the Scanner, and therefore, it is missing some features compared to other parsers.

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 take into account any errors that may occur during tokenization. There could also be more error handling when the file is being read. With the previous deliverable, we could have included more of the SCL grammar as part of our subset.

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):
        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()
                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:

```
import json

# Define constants for token types
# These constants represent the various types of tokens we might encounter in the SCL language.
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 are reserved words that have special meaning in the language.
KEYWORDS = [DISPLAY, IF, THEN, ENDIF, FUNCTION, IS, ENDFUN, PARAMETERS, INTEGER, FLOAT, CHAR, NOT]

# Parser class
jschul37, 2 hours 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.
    def get_next_token(self):
        if self.token_index < len(self.tokens):
            self.current_token = self.tokens[self.token_index]
            self.token_index += 1
        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
        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).
    # For example: arr[expression]
    def array_def(self):
        self.match(LB)
        self.expression()
        self.match(RB)

    # Parse an arithmetic expression that can include addition, subtraction, multiplication, or division.
    # Expressions can be combined using the aforementioned arithmetic operators.
    def expression(self):
        self.term()
        while self.current_token and self.current_token[0] in [PLUS, MINUS, STAR, DIVOP]:
            # Match arithmetic operators and then parse the next term.
            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)
            self.term()

```

```
    # Parse a term which can be an identifier, a signed or unsigned constant,
    # a parenthesized expression, a function call, or a string literal.
    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.
    # Conditions can be used in constructs like IF...THEN.
    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()
        # No else part required, since ε means do nothing

```

```python
    # Parse a list of parameters for a function.
    # Each parameter is an identifier followed by its data type.
    def param_list(self):
        self.match(IDENTIFIER)
        self.match(OF)
        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()

```

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"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["IDENTIFIER", "z"],
["EQUOP", "="], ["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["IDENTIFIER", "product"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["STAR", "*"], ["IDENTIFIER", "y"],
["IDENTIFIER", "quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["UNSIGNICON", "5"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"],
["EQUOP", "="], ["IDENTIFIER", "x"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"],
["RB", "]"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["],
["UNSIGNICON", "2"], ["RB", "]"], ["STAR", "*"], ["UNSIGNICON", "2"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "y"], ["COMMA", ","],
["IDENTIFIER", "z"], ["COMMA", ","], ["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON",
"2"], ["RB", "]"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["IF", "IF"], ["IDENTIFIER", "x"], ["RELOP", ">"], ["UNSIGNICON",
"5"], ["THEN", "THEN"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is greater than 5\""], ["ENDIF", "ENDIF"], ["IF", "IF"], ["IDENTIFIER", "y"], ["RELOP", "<"], ["UNSIGNICON",
"10"], ["THEN", "THEN"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is less than 10\""], ["ENDIF", "ENDIF"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "add_numbers"],
["PARAMETERS", "PARAMETERS"], ["IDENTIFIER", "a"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["COMMA", ","], ["IDENTIFIER", "b"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"],
["IDENTIFIER", "result"], ["EQUOP", "="], ["IDENTIFIER", "a"], ["PLUS", "+"], ["IDENTIFIER", "b"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "result"], ["ENDFUN", "ENDFUN"],
["IDENTIFIER", "add_numbers"], ["IDENTIFIER", "sum"], ["EQUOP", "="], ["IDENTIFIER", "add_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")
"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "sum"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "multiply_numbers"], ["PARAMETERS", "PARAMETERS"], ["IDENTIFIER", "m"], ["OF", "OF"],
["INTEGER", "INTEGER"], ["COMMA", ","], ["IDENTIFIER", "n"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"], ["IDENTIFIER", "multiplication"], ["EQUOP", "="], ["IDENTIFIER",
"m"], ["STAR", "*"], ["IDENTIFIER", "n"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "multiplication"], ["ENDFUN", "ENDFUN"], ["IDENTIFIER", "multiply_numbers"], ["IDENTIFIER",
"product_result"], ["EQUOP", "="], ["IDENTIFIER", "multiply_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")"], ["DISPLAY", "DISPLAY"],
["IDENTIFIER", "product_result"]]
```


```
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.
```

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"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["UNSIGNICON", "5"], ["IDENTIFIER", "z"], ["EQUOP", "="],
["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["IDENTIFIER", "product"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["STAR", "*"], ["IDENTIFIER", "y"], ["IDENTIFIER",
"quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["IDENTIFIER", "z"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["EQUOP", "="], ["IDENTIFIER", "x"],
["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"], ["PLUS", "+"],
["UNSIGNICON", "5"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"],
["STAR", "*"], ["UNSIGNICON", "2"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "y"], ["COMMA", ","], ["IDENTIFIER", "z"], ["COMMA", ","],
["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["DISPLAY",
"DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["IF", "IF"], ["IDENTIFIER", "x"], ["RELOP", "<"], ["UNSIGNICON", "5"], ["THEN", "THEN"],
["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is greater than 5\""], ["ENDIF", "ENDIF"], ["IF", "IF"], ["IDENTIFIER", "y"], ["RELOP", "<"], ["UNSIGNICON", "10"], ["THEN", "THEN"],
["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is less than 10\""], ["ENDIF", "ENDIF"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "add_numbers"], ["PARAMETERS", "PARAMETERS"],
["IDENTIFIER", "a"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["COMMA", ","], ["IDENTIFIER", "b"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"], ["IDENTIFIER", "result"],
["EQUOP", "="], ["IDENTIFIER", "a"], ["PLUS", "+"], ["IDENTIFIER", "b"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "result"], ["ENDFUN", "ENDFUN"], ["IDENTIFIER", "add_numbers"],
["IDENTIFIER", "sum"], ["EQUOP", "="], ["IDENTIFIER", "add_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")"], ["DISPLAY", "DISPLAY"],
["IDENTIFIER", "sum"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "multiply_numbers"], ["PARAMETERS", "PARAMETERS"], ["IDENTIFIER", "m"], ["OF", "OF"], ["INTEGER", "INTEGER"],
["COMMA", ","], ["IDENTIFIER", "n"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"], ["IDENTIFIER", "multiplication"], ["EQUOP", "="], ["IDENTIFIER", "m"], ["STAR", "*"],
["IDENTIFIER", "n"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "multiplication"], ["ENDFUN", "ENDFUN"], ["IDENTIFIER", "multiply_numbers"], ["IDENTIFIER", "product_result"], ["EQUOP",
"="], ["IDENTIFIER", "multiply_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "product_result"]]
```


```
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"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["IDENTIFIER", "z"],
["EQUOP", "="], ["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["IDENTIFIER", "product"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["STAR", "*"], ["IDENTIFIER", "y"],
["IDENTIFIER", "quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["IDENTIFIER", "z"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["EQUOP", "="],
["IDENTIFIER", "x"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"],
["PLUS", "+"], ["UNSIGNICON", "5"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"],
["RB", "]"], ["STAR", "*"], ["UNSIGNICON", "2"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "y"], ["COMMA", ","], ["IDENTIFIER", "z"],
["COMMA", ","], ["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"],
["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["IF", "IF"], ["IDENTIFIER", "x"], ["RELOP", "<"], ["UNSIGNICON", "5"], ["THEN",
"THEN"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is greater than 5\""], ["ENDIF", "ENDIF"], ["IF", "IF"], ["IDENTIFIER", "y"], ["RELOP", "<"], ["UNSIGNICON", "10"], ["THEN",
"THEN"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is less than 10\""], ["ENDIF", "ENDIF"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "add_numbers"], ["PARAMETERS",
"PARAMETERS"], ["IDENTIFIER", "a"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["COMMA", ","], ["IDENTIFIER", "b"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"], ["IDENTIFIER",
"result"], ["EQUOP", "="], ["IDENTIFIER", "a"], ["PLUS", "+"], ["IDENTIFIER", "b"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "result"], ["ENDFUN", "ENDFUN"], ["IDENTIFIER",
"add_numbers"], ["IDENTIFIER", "sum"], ["EQUOP", "="], ["IDENTIFIER", "add_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")"], ["DISPLAY",
"DISPLAY"], ["IDENTIFIER", "sum"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "multiply_numbers"], ["PARAMETERS", "PARAMETERS"], ["IDENTIFIER", "m"], ["OF", "OF"], ["INTEGER",
"INTEGER"], ["COMMA", ","], ["IDENTIFIER", "n"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"], ["IDENTIFIER", "multiplication"], ["EQUOP", "="], ["IDENTIFIER", "m"], ["STAR",
"*"], ["IDENTIFIER", "n"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "multiplication"], ["ENDFUN", "ENDFUN"], ["IDENTIFIER", "multiply_numbers"], ["IDENTIFIER", "product_result"],
["EQUOP", "="], ["IDENTIFIER", "multiply_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER",
```


```
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 27
```

After Fixing Second Syntax Error:


```
[["IDENTIFIER", "x"], ["EQUOP", "="], ["UNSIGNICON", "10"], ["IDENTIFIER", "y"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["IDENTIFIER", "z"],
["EQUOP", "="], ["IDENTIFIER", "x"], ["MINUS", "-"], ["IDENTIFIER", "y"], ["IDENTIFIER", "product"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["STAR", "*"], ["IDENTIFIER", "y"],
["IDENTIFIER", "quotient"], ["EQUOP", "="], ["IDENTIFIER", "y"], ["DIVOP", " /"], ["IDENTIFIER", "z"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"], ["RB", "]"],
["EQUOP", "="], ["IDENTIFIER", "x"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "2"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "10"],
["RB", "]"], ["PLUS", "+"], ["UNSIGNICON", "5"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["EQUOP", "="], ["IDENTIFIER", "arr"], ["LB", "["],
["UNSIGNICON", "2"], ["RB", "]"], ["STAR", "*"], ["UNSIGNICON", "2"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "x"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "y"], ["COMMA", ","],
["IDENTIFIER", "z"], ["COMMA", ","], ["IDENTIFIER", "product"], ["COMMA", ","], ["IDENTIFIER", "quotient"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON",
"2"], ["RB", "]"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "arr"], ["LB", "["], ["UNSIGNICON", "3"], ["RB", "]"], ["IF", "IF"], ["IDENTIFIER", "x"], ["RELOP", "<"], ["UNSIGNICON",
"5"], ["THEN", "THEN"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"X is greater than 5\""], ["ENDIF", "ENDIF"], ["IF", "IF"], ["IDENTIFIER", "y"], ["RELOP", "<"], ["UNSIGNICON",
"10"], ["THEN", "THEN"], ["DISPLAY", "DISPLAY"], ["STRING_LITERAL", "\"Y is less than 10\""], ["ENDIF", "ENDIF"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "add_numbers"],
["PARAMETERS", "PARAMETERS"], ["IDENTIFIER", "a"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["COMMA", ","], ["IDENTIFIER", "b"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"],
["IDENTIFIER", "result"], ["EQUOP", "="], ["IDENTIFIER", "a"], ["PLUS", "+"], ["IDENTIFIER", "b"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "result"], ["ENDFUN", "ENDFUN"],
["IDENTIFIER", "add_numbers"], ["IDENTIFIER", "sum"], ["EQUOP", "="], ["IDENTIFIER", "add_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")
"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "sum"], ["FUNCTION", "FUNCTION"], ["IDENTIFIER", "multiply_numbers"], ["PARAMETERS", "PARAMETERS"], ["IDENTIFIER", "m"], ["OF", "OF"],
["INTEGER", "INTEGER"], ["COMMA", ","], ["IDENTIFIER", "n"], ["OF", "OF"], ["INTEGER", "INTEGER"], ["IS", "IS"], ["IDENTIFIER", "multiplication"], ["EQUOP", "="], ["IDENTIFIER",
"m"], ["STAR", "*"], ["IDENTIFIER", "n"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "multiplication"], ["ENDFUN", "ENDFUN"], ["IDENTIFIER", "multiply_numbers"], ["IDENTIFIER",
"product_result"], ["EQUOP", "="], ["IDENTIFIER", "multiply_numbers"], ["LP", "("], ["IDENTIFIER", "x"], ["COMMA", ","], ["IDENTIFIER", "y"], ["RP", ")"], ["DISPLAY", "DISPLAY"],
["IDENTIFIER", "product_result"]]
```



```
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.
```

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\""], ["IDENTIFIER", "b"], ["EQUOP", "="], ["STRING_LITERAL", "\"this is also a string\""], ["IDENTIFIER",
"c"], ["EQUOP", "="], ["IDENTIFIER", "a"], ["PLUS", "+"], ["IDENTIFIER", "b"], ["IDENTIFIER", "x"], ["EQUOP", "="], ["UNSIGNICON", "5"], ["IDENTIFIER", "y"], ["EQUOP", "="],
["UNSIGNICON", "6"], ["IDENTIFIER", "z"], ["EQUOP", "="], ["IDENTIFIER", "x"], ["PLUS", "+"], ["UNSIGNICON", "7"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "a"], ["DISPLAY",
"DISPLAY"], ["IDENTIFIER", "b"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "a"], ["COMMA", ","], ["IDENTIFIER", "b"], ["DISPLAY", "DISPLAY"], ["IDENTIFIER", "z"]]
```


```
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.
```

Comments and Conclusion

The program successfully takes a tokens.json file as input and parses through it. The parser utilizes recursive descent to parse a sequence of statements until a specific token is found. If an unexpected token is found, a syntax error is raised, and if the parsing is successful, a message is printed to the system. We learned about the functionalities of a recursive descent parser, which we will build upon in the creation of a fully functional interpreter in a future deliverable.

References

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