Al-imam Mohammed bin Saud University
College of Computer and Information Sciences

Computer Science Department
CS445 - Compilers

Phase – 1: Develop a scanner

Team Members	Team ID’s	Team Email’s
Alya Fahad Alsalamah	[redacted]	[redacted]@sm.imamu.edu.sa
Nada Rahwan Alzahrani	[redacted]	[redacted]@sm.imamu.edu.sa
Lujain Fahad Almogren	[redacted]	[redacted]n@sm.imamu.edu.sa

Section:
371

Instructor:
Dr. Manal Alsabhan

Date: May 20, 2023

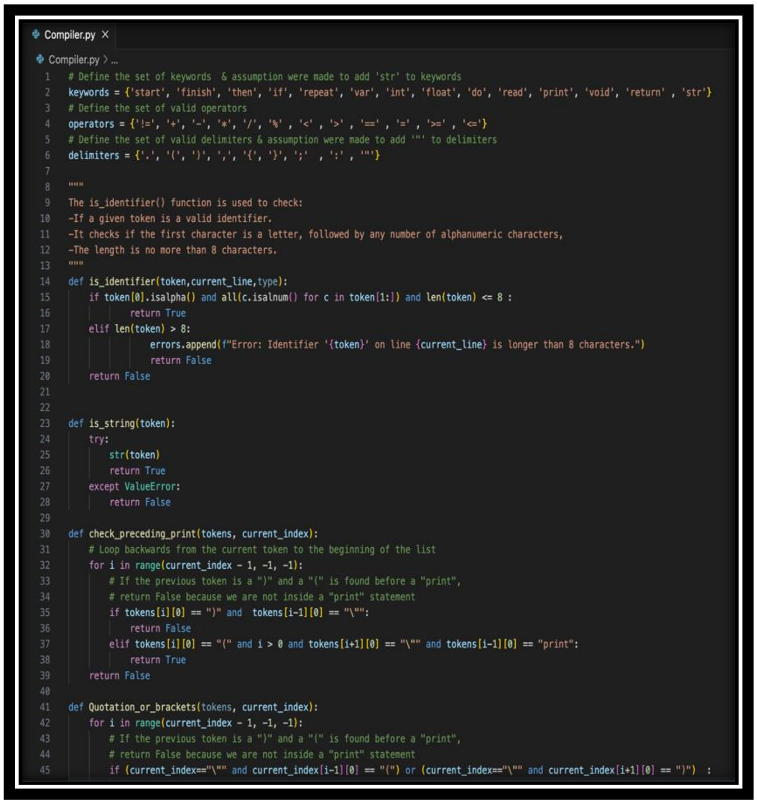
Table Contents

1. The Code

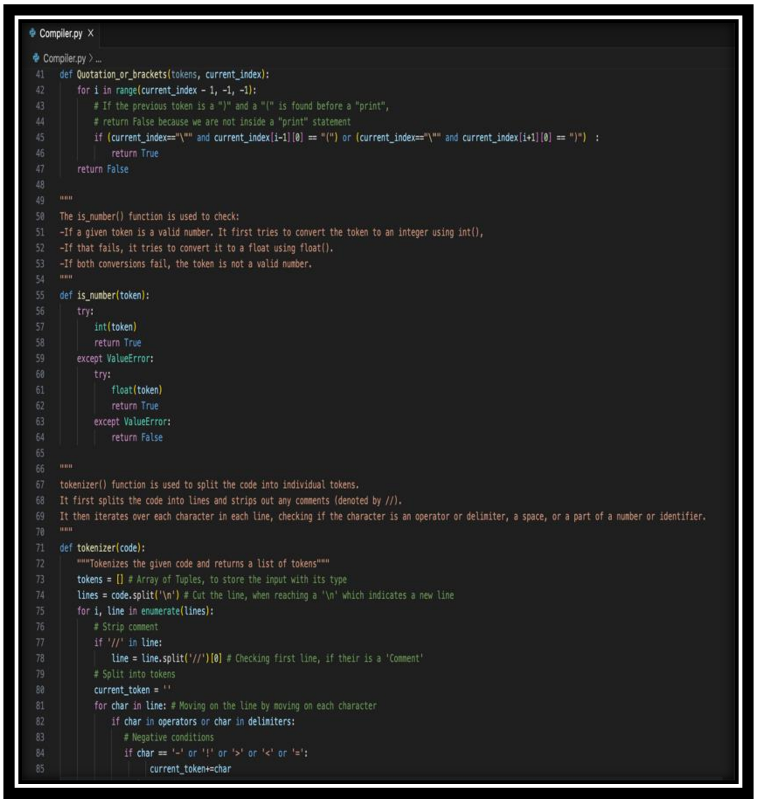
1.1 Assumptions

- For the delimiters, ' " ' and ':' have been added so the scanner can function correctly.
- For the keywords, 'str’ has been added so the scanner can function correctly.
- This scanner is space sensitive. To make the scanner read the code correctly, there must be a space after writing any lexeme.
- If the token is a keyword and it is written in upper case letters, the code handles it and converted it to small case letters.

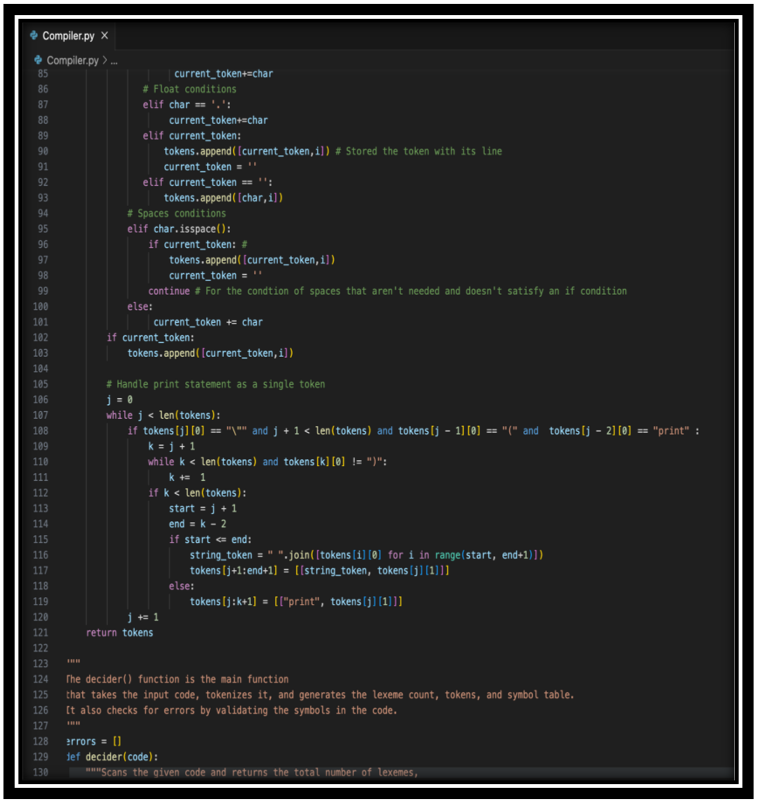
1.2 Code Implementation



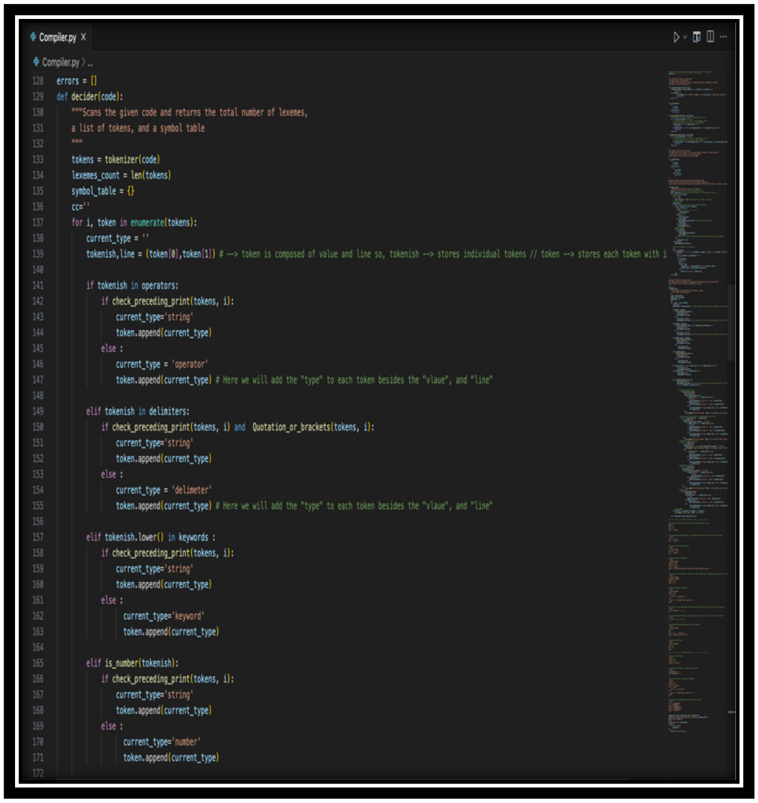
```python
# Define the set of keywords  & assumption were made to add 'str' to keywords
keywords = {'start', 'finish', 'then', 'if', 'repeat', 'var', 'int', 'float', 'do', 'read', 'print', 'void', 'return' , 'str'}
# Define the set of valid operators
operators = {'!=', '+', '-', '*', '/', '%' , '<' , '>' , '==' , '=' , '>=' , '<='}
# Define the set of valid delimiters & assumption were made to add '"' to delimiters
delimiters = {'.', '(', ')', ',', '{', '}', ';'  , ':' , '"'}

"""
The is_identifier() function is used to check:
-If a given token is a valid identifier.
-It checks if the first character is a letter, followed by any number of alphanumeric characters,
-The length is no more than 8 characters.
"""
def is_identifier(token,current_line,type):
    if token[0].isalpha() and all(c.isalnum() for c in token[1:]) and len(token) <= 8 :
            return True
    elif len(token) > 8:
                errors.append(f"Error: Identifier '{token}' on line {current_line} is longer than 8 characters.")
                return False
    return False


def is_string(token):
    try:
        str(token)
        return True
    except ValueError:
        return False

def check_preceding_print(tokens, current_index):
    # Loop backwards from the current token to the beginning of the list
    for i in range(current_index - 1, -1, -1):
        # If the previous token is a ")" and a "(" is found before a "print",
        # return False because we are not inside a "print" statement
        if tokens[i][0] == ")" and  tokens[i-1][0] == "\"":
            return False
        elif tokens[i][0] == "(" and i > 0 and tokens[i+1][0] == "\"" and tokens[i-1][0] == "print":
            return True
    return False

def Quotation_or_brackets(tokens, current_index):
    for i in range(current_index - 1, -1, -1):
        # If the previous token is a ")" and a "(" is found before a "print",
        # return False because we are not inside a "print" statement
        if (current_index=="\"" and current_index[i-1][0] == "(") or (current_index=="\"" and current_index[i+1][0] == ")")  :
```



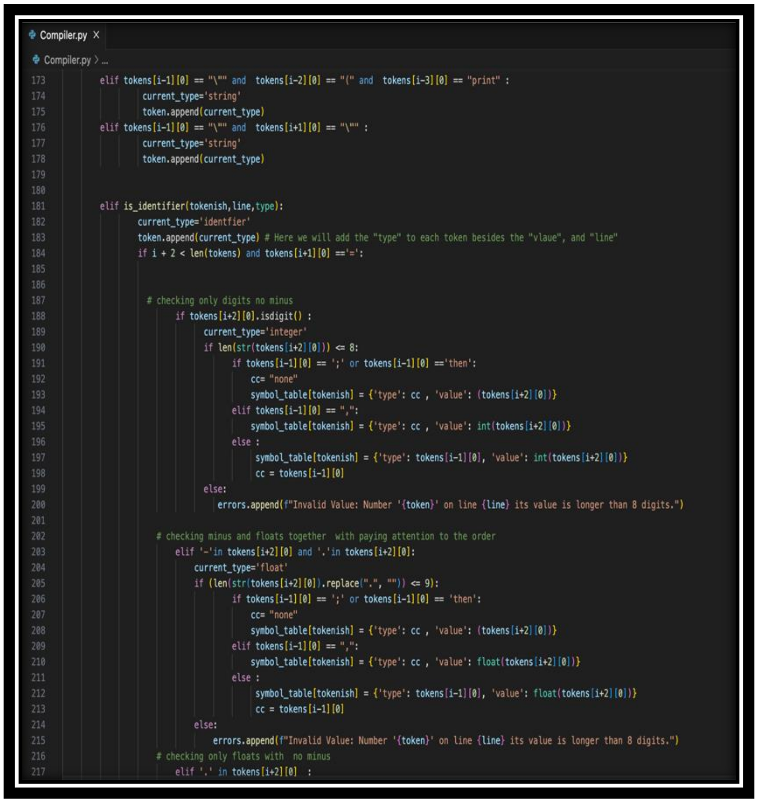
```python
def Quotation_or_brackets(tokens, current_index):
    for i in range(current_index - 1, -1, -1):
        # If the previous token is a ")" and a "(" is found before a "print",
        # return False because we are not inside a "print" statement
        if (current_index=="\"" and current_index[i-1][0] == "(") or (current_index=="\"" and current_index[i+1][0] == ")")  :
            return True
    return False

"""
The is_number() function is used to check:
-If a given token is a valid number. It first tries to convert the token to an integer using int(),
-If that fails, it tries to convert it to a float using float().
-If both conversions fail, the token is not a valid number.
"""
def is_number(token):
    try:
        int(token)
        return True
    except ValueError:
        try:
            float(token)
            return True
        except ValueError:
            return False

"""
tokenizer() function is used to split the code into individual tokens.
It first splits the code into lines and strips out any comments (denoted by //).
It then iterates over each character in each line, checking if the character is an operator or delimiter, a space, or a part of a number or identifier.
"""
def tokenizer(code):
    """Tokenizes the given code and returns a list of tokens"""
    tokens = [] # Array of Tuples, to store the input with its type
    lines = code.split('\n') # Cut the line, when reaching a '\n' which indicates a new line
    for i, line in enumerate(lines):
        # Strip comment
        if '//' in line:
            line = line.split('//')[0] # Checking first line, if their is a 'Comment'
        # Split into tokens
        current_token = ''
        for char in line: # Moving on the line by moving on each character
            if char in operators or char in delimiters:
               # Negative conditions
               if char == '-' or '!' or '>' or '<' or '=':
                    current_token+=char
```



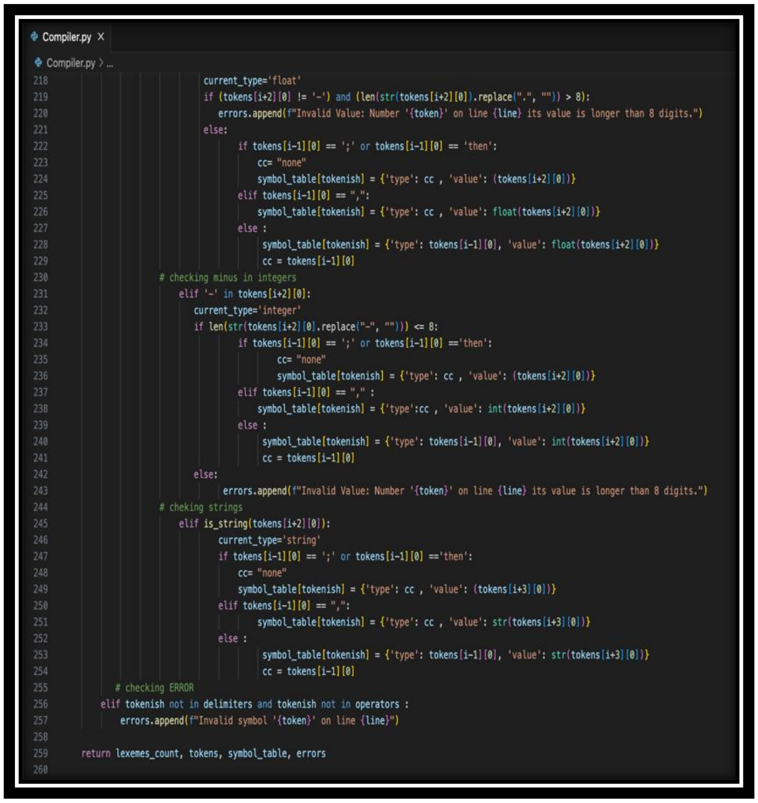
```python
                        current_token+=char
                # Float conditions
                elif char == '.':
                    current_token+=char
                elif current_token:
                    tokens.append([current_token,i]) # Stored the token with its line
                    current_token = ''
                elif current_token == '':
                    tokens.append([char,i])
            # Spaces conditions
            elif char.isspace():
                if current_token: #
                    tokens.append([current_token,i])
                    current_token = ''
                continue # For the condtion of spaces that aren't needed and doesn't satisfy an if condition
            else:
                current_token += char
        if current_token:
            tokens.append([current_token,i])

        # Handle print statement as a single token
        j = 0
        while j < len(tokens):
            if tokens[j][0] == "\"" and j + 1 < len(tokens) and tokens[j - 1][0] == "(" and  tokens[j - 2][0] == "print" :
                k = j + 1
                while k < len(tokens) and tokens[k][0] != ")":
                    k +=  1
                if k < len(tokens):
                    start = j + 1
                    end = k - 2
                    if start <= end:
                        string_token = " ".join([tokens[i][0] for i in range(start, end+1)])
                        tokens[j+1:end+1] = [[string_token, tokens[j][1]]]
                    else:
                        tokens[j:k+1] = [["print", tokens[j][1]]]
            j += 1
    return tokens

"""
The decider() function is the main function
that takes the input code, tokenizes it, and generates the lexeme count, tokens, and symbol table.
It also checks for errors by validating the symbols in the code.
"""
errors = []
def decider(code):
    """Scans the given code and returns the total number of lexemes,
```



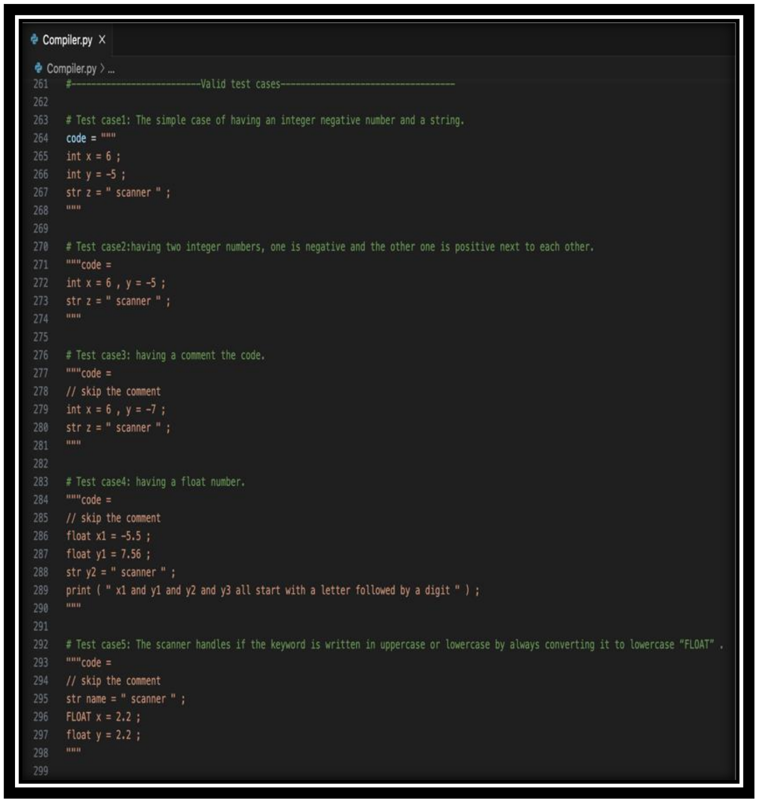
```python
errors = []
def decider(code):
    """Scans the given code and returns the total number of lexemes,
    a list of tokens, and a symbol table
    """
    tokens = tokenizer(code)
    lexemes_count = len(tokens)
    symbol_table = {}
    cc=''
    for i, token in enumerate(tokens):
        current_type = ''
        tokenish,line = (token[0],token[1]) # --> token is composed of value and line so, tokenish --> stores individual tokens // token --> stores each token with i

        if tokenish in operators:
            if check_preceding_print(tokens, i):
                current_type='string'
                token.append(current_type)
            else :
                current_type = 'operator'
                token.append(current_type) # Here we will add the "type" to each token besides the "vlaue", and "line"

        elif tokenish in delimiters:
            if check_preceding_print(tokens, i) and  Quotation_or_brackets(tokens, i):
                current_type='string'
                token.append(current_type)
            else :
                current_type = 'delimeter'
                token.append(current_type) # Here we will add the "type" to each token besides the "vlaue", and "line"

        elif tokenish.lower() in keywords :
            if check_preceding_print(tokens, i):
                current_type='string'
                token.append(current_type)
            else :
                  current_type='keyword'
                  token.append(current_type)

        elif is_number(tokenish):
            if check_preceding_print(tokens, i):
                current_type='string'
                token.append(current_type)
            else :
                  current_type='number'
                  token.append(current_type)

```



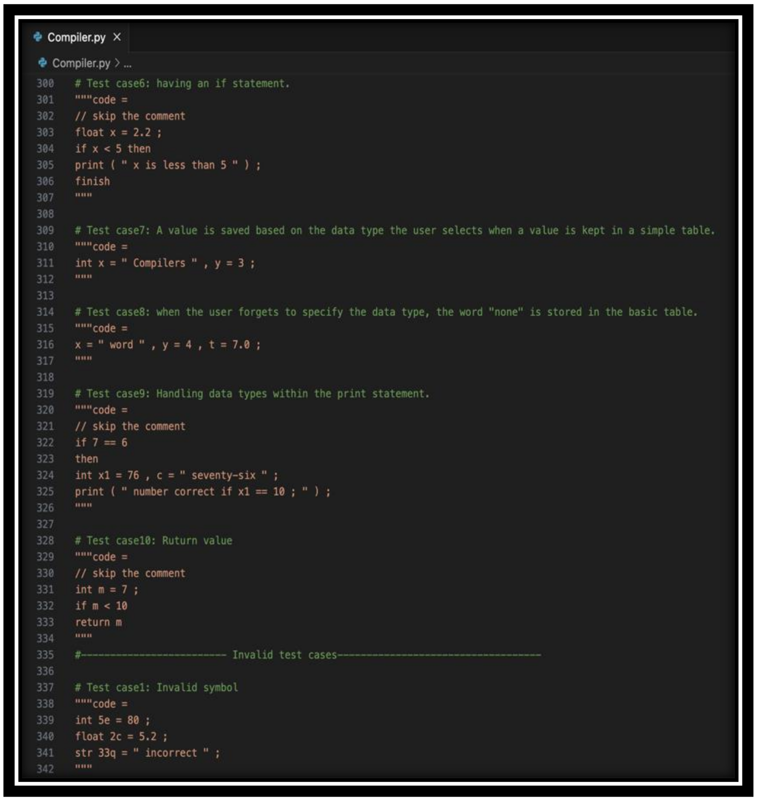
```python
        elif tokens[i-1][0] == "\"" and  tokens[i-2][0] == "(" and  tokens[i-3][0] == "print" :
                current_type='string'
                token.append(current_type)
        elif tokens[i-1][0] == "\"" and  tokens[i+1][0] == "\"" :
                current_type='string'
                token.append(current_type)


        elif is_identifier(tokenish,line,type):
                current_type='identfier'
                token.append(current_type) # Here we will add the "type" to each token besides the "vlaue", and "line"
                if i + 2 < len(tokens) and tokens[i+1][0] =='=':


                  # checking only digits no minus
                        if tokens[i+2][0].isdigit() :
                            current_type='integer'
                            if len(str(tokens[i+2][0])) <= 8:
                                  if tokens[i-1][0] == ';' or tokens[i-1][0] =='then':
                                      cc= "none"
                                      symbol_table[tokenish] = {'type': cc , 'value': (tokens[i+2][0])}
                                  elif tokens[i-1][0] == ",":
                                      symbol_table[tokenish] = {'type': cc , 'value': int(tokens[i+2][0])}
                                  else :
                                       symbol_table[tokenish] = {'type': tokens[i-1][0], 'value': int(tokens[i+2][0])}
                                       cc = tokens[i-1][0]
                            else:
                               errors.append(f"Invalid Value: Number '{token}' on line {line} its value is longer than 8 digits.")

                    # checking minus and floats together  with paying attention to the order
                       elif '-'in tokens[i+2][0] and '.'in tokens[i+2][0]:
                           current_type='float'
                           if (len(str(tokens[i+2][0]).replace(".", "")) <= 9):
                                  if tokens[i-1][0] == ';' or tokens[i-1][0] == 'then':
                                      cc= "none"
                                      symbol_table[tokenish] = {'type': cc , 'value': (tokens[i+2][0])}
                                  elif tokens[i-1][0] == ",":
                                      symbol_table[tokenish] = {'type': cc , 'value': float(tokens[i+2][0])}
                                  else :
                                       symbol_table[tokenish] = {'type': tokens[i-1][0], 'value': float(tokens[i+2][0])}
                                       cc = tokens[i-1][0]
                           else:
                              errors.append(f"Invalid Value: Number '{token}' on line {line} its value is longer than 8 digits.")
                    # checking only floats with  no minus
                       elif '.' in tokens[i+2][0]  :
```



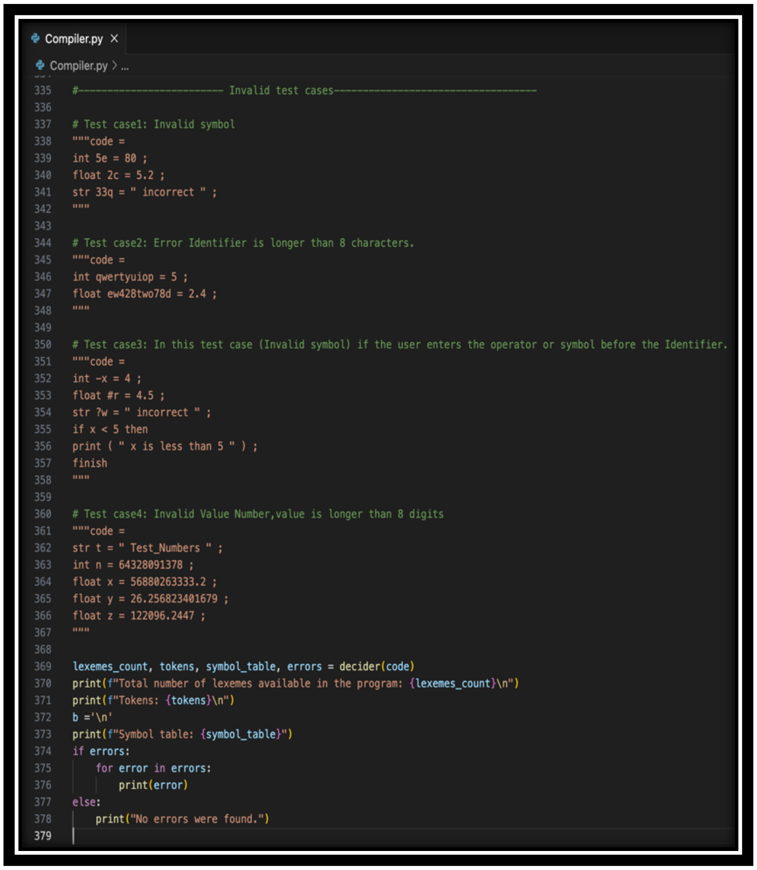
```python
                             current_type='float'
                             if (tokens[i+2][0] != '-') and (len(str(tokens[i+2][0]).replace(".", "")) > 8):
                                errors.append(f"Invalid Value: Number '{token}' on line {line} its value is longer than 8 digits.")
                             else:
                                    if tokens[i-1][0] == ';' or tokens[i-1][0] == 'then':
                                        cc= "none"
                                        symbol_table[tokenish] = {'type': cc , 'value': (tokens[i+2][0])}
                                    elif tokens[i-1][0] == ",":
                                        symbol_table[tokenish] = {'type': cc , 'value': float(tokens[i+2][0])}
                                    else :
                                         symbol_table[tokenish] = {'type': tokens[i-1][0], 'value': float(tokens[i+2][0])}
                                         cc = tokens[i-1][0]
                    # checking minus in integers
                        elif '-' in tokens[i+2][0]:
                           current_type='integer'
                           if len(str(tokens[i+2][0].replace("-", ""))) <= 8:
                                    if tokens[i-1][0] == ';' or tokens[i-1][0] =='then':
                                            cc= "none"
                                            symbol_table[tokenish] = {'type': cc , 'value': (tokens[i+2][0])}
                                    elif tokens[i-1][0] == "," :
                                        symbol_table[tokenish] = {'type':cc , 'value': int(tokens[i+2][0])}
                                    else :
                                         symbol_table[tokenish] = {'type': tokens[i-1][0], 'value': int(tokens[i+2][0])}
                                         cc = tokens[i-1][0]
                           else:
                                 errors.append(f"Invalid Value: Number '{token}' on line {line} its value is longer than 8 digits.")
                    # cheking strings
                        elif is_string(tokens[i+2][0]):
                                current_type='string'
                                if tokens[i-1][0] == ';' or tokens[i-1][0] =='then':
                                    cc= "none"
                                    symbol_table[tokenish] = {'type': cc , 'value': (tokens[i+3][0])}
                                elif tokens[i-1][0] == ",":
                                        symbol_table[tokenish] = {'type': cc , 'value': str(tokens[i+3][0])}
                                else :
                                         symbol_table[tokenish] = {'type': tokens[i-1][0], 'value': str(tokens[i+3][0])}
                                         cc = tokens[i-1][0]
           # checking ERROR
        elif tokenish not in delimiters and tokenish not in operators :
            errors.append(f"Invalid symbol '{token}' on line {line}")

    return lexemes_count, tokens, symbol_table, errors

```



```
#------------------------Valid test cases---------------------------------

# Test case1: The simple case of having an integer negative number and a string.
code = """
int x = 6 ;
int y = -5 ;
str z = " scanner " ;
"""

# Test case2:having two integer numbers, one is negative and the other one is positive next to each other.
"""code =
int x = 6 , y = -5 ;
str z = " scanner " ;
"""

# Test case3: having a comment the code.
"""code =
// skip the comment
int x = 6 , y = -7 ;
str z = " scanner " ;
"""

# Test case4: having a float number.
"""code =
// skip the comment
float x1 = -5.5 ;
float y1 = 7.56 ;
str y2 = " scanner " ;
print ( " x1 and y1 and y2 and y3 all start with a letter followed by a digit " ) ;
"""

# Test case5: The scanner handles if the keyword is written in uppercase or lowercase by always converting it to lowercase "FLOAT" .
"""code =
// skip the comment
str name = " scanner " ;
FLOAT x = 2.2 ;
float y = 2.2 ;
"""

```



```
# Test case6: having an if statement.
"""code =
// skip the comment
float x = 2.2 ;
if x < 5 then
print ( " x is less than 5 " ) ;
finish
"""

# Test case7: A value is saved based on the data type the user selects when a value is kept in a simple table.
"""code =
int x = " Compilers " , y = 3 ;
"""

# Test case8: when the user forgets to specify the data type, the word "none" is stored in the basic table.
"""code =
x = " word " , y = 4 , t = 7.0 ;
"""

# Test case9: Handling data types within the print statement.
"""code =
// skip the comment
if 7 == 6
then
int x1 = 76 , c = " seventy-six " ;
print ( " number correct if x1 == 10 ; " ) ;
"""

# Test case10: Ruturn value
"""code =
// skip the comment
int m = 7 ;
if m < 10
return m
"""
#------------------------ Invalid test cases----------------------------------

# Test case1: Invalid symbol
"""code =
int 5e = 80 ;
float 2c = 5.2 ;
str 33q = " incorrect " ;
"""
```



```
#------------------------ Invalid test cases----------------------------------

# Test case1: Invalid symbol
"""code =
int 5e = 80 ;
float 2c = 5.2 ;
str 33q = " incorrect " ;
"""

# Test case2: Error Identifier is longer than 8 characters.
"""code =
int qwertyuiop = 5 ;
float ew428two78d = 2.4 ;
"""

# Test case3: In this test case (Invalid symbol) if the user enters the operator or symbol before the Identifier.
"""code =
int -x = 4 ;
float #r = 4.5 ;
str ?w = " incorrect " ;
if x < 5 then
print ( " x is less than 5 " ) ;
finish
"""

# Test case4: Invalid Value Number,value is longer than 8 digits
"""code =
str t = " Test_Numbers " ;
int n = 64328091378 ;
float x = 56880263333.2 ;
float y = 26.256823401679 ;
float z = 122096.2447 ;
"""

lexemes_count, tokens, symbol_table, errors = decider(code)
print(f"Total number of lexemes available in the program: {lexemes_count}\n")
print(f"Tokens: {tokens}\n")
b ='\n'
print(f"Symbol table: {symbol_table}")
if errors:
    for error in errors:
        print(error)
else:
    print("No errors were found.")

```

1.3 Code Explanation

A set for the defined keywords, operators, and delimiters has been defined so that the scanner can find a pattern for the given lexeme. Method “tokenizer” loops through the input code lines to save the tokens in an array called “tokens” that stores the token and its line. The method can determine where each token starts and finishes with respect to space. So:

- if the current character is “-” then it is a negative number so add to it the rest of the number. Also, check whether the operator is consisting of two parts like “<=” or “>=” or “! =” to concatenate the two parts together as one token this can be done because the code is based sensitive, and it realizes that there is no space between the first operator and the second operator.

- If the current character is “.” then it is a float number so add the following digits to the previous digits that are between “.” to create a float token.

- If there is a comment, then ignore it and do not save it as a token.

- If the current character is a space this means that the single token has been read and finished. So, the current token should be assigned to space “ ” so it can read the next token with an empty “current_token”. The space condition needs to be checked in each read for a token.

The method “tokenizer” Returns a token that can be utilized in the method “decider” which decides the correct pattern to describe the lexeme. To be able for the “decider” method to do its job, it needs to call other Boolean methods to help build the symbol table and print the tokens with their pattern correctly. The Boolean methods are:

- “is_identifier” that returns true if the identifier meats the required conditions which are the identifier should start with an alphabet and it should not exceed 8 characters otherwise it returns false.
- “is_number” returns true if it is a number.
- “is_string” that returns true if it is a string.
- “check_preceding_print” checks if the current token is inside a print statement. If there is an identifier that meets the identifier conditions or a keyword or an operator or a number inside the print statement, it must always assassins the type to be a string since it sentence inside a print statement.
- “Quotation_or_brackets” pays respect if the print statement starts and ends with ‘(“ ”) ‘ and the opening and closing brackets and the quotation marks should not be considered as a string. So, the main job of this method

is to contradict “check_preceding_print” job so it does not store the brackets and quotations of the print statement as a string.

After tokenizing everything there is a part of the two analyzers that loop through the whole array to find “print” followed by aquotation mark and then an opening bracket so it can join every token that is in the print statement as one token and enter it finds a quotation mark followed by a closing bracket.

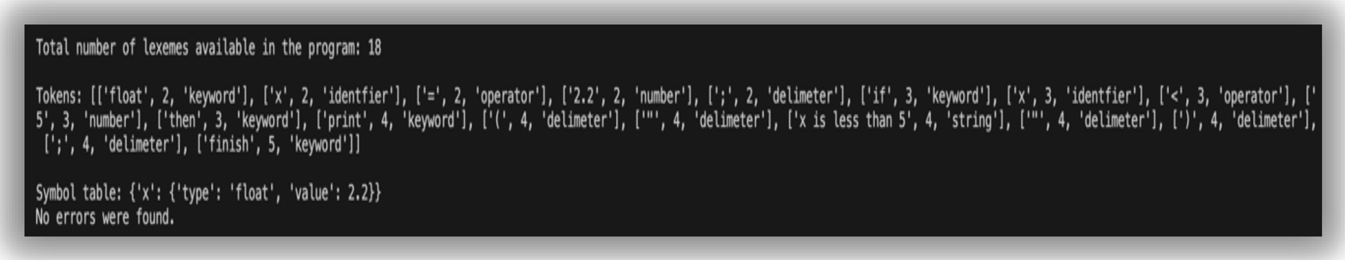
Also, “decider” checks every token so it can determine its pattern and if it is an identifier, it stores it in the symbol table. Most of the work is done when an identifier is detected, and the type of the identifier is stored in this simple table even if it doesn't match the type of the value because it is alexical analyzer. Therefore, if the type is not declared it should be stored in the simple table as “none”. The “decider” knows if there is an identifier by checking if the token next stored index is an equal sign. If such, then check if it is an integer or a float number with respect that the number should not exceed 8 digits. If there is an error that should be handled. So, if an identifier starts with any other symbol rather than a character it should append an error indicating which line of the code has failed. Also, it appends an error if the numbers and identifiers exceed the accepted length.

Moreover, “lexemes_count” is counted by assigning it to the length of the “tokens” array, and the tokens are assigned with three values which are the lexeme and the line it has occurred, and its pattern.

1.4 Code Output

An explanation of the code output is as follows:

```
code = """
// skip the comment
float x = 2.2 ;
if x < 5 then
print ( " x is less than 5 " ) ;
finish
"""
```



```
Total number of lexemes available in the program: 18

Tokens: [['float', 2, 'keyword'], ['x', 2, 'identfier'], ['=', 2, 'operator'], ['2.2', 2, 'number'], [';', 2, 'delimeter'], ['if', 3, 'keyword'], ['x', 3, 'identfier'], ['<', 3, 'operator'], ['
5', 3, 'number'], ['then', 3, 'keyword'], ['print', 4, 'keyword'], ['(', 4, 'delimeter'], ['"', 4, 'delimeter'], ['x is less than 5', 4, 'string'], ['"', 4, 'delimeter'], [')', 4, 'delimeter'],
 [';', 4, 'delimeter'], ['finish', 5, 'keyword']]

Symbol table: {'x': {'type': 'float', 'value': 2.2}}
No errors were found.
```

- The scanner output provides information about the number of lexemes, symbol table, and error checking of the given input.

- The first line of the output indicates that there are 18 lexemes (tokens) in the program. The three parts that make up each token's representation are the lexeme, the line number in the program where it appears, and the pattern that describes the lexeme (keyword, identifier, operator, integer, string, number, or delimiter).

- The second line displayed the symbol table, a data structure that houses details about identifiers used in the program. 'X' is the only identifier defined in this case's program; it has a value of 2.2 and a type of 'float'.

- The third line states that no mistakes were discovered while analyzing the program.

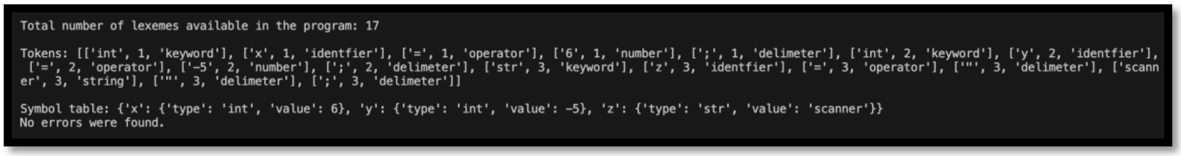
2. Test Cases

2.1 Valid Test Cases

Test case 1:

The simple case of having an integer negative number and a string.

```
code = """
int x = 6 ;
int y = -5 ;
str z = " scanner " ;
"""
```

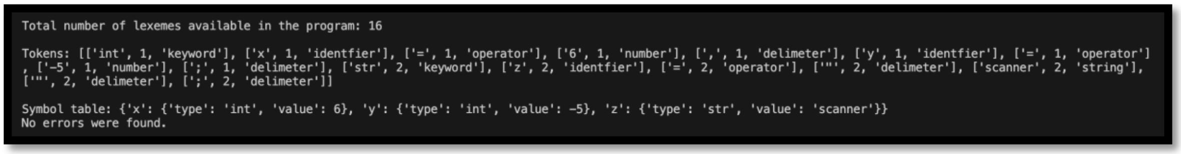


```
Total number of lexemes available in the program: 17

Tokens: [['int', 1, 'keyword'], ['x', 1, 'identfier'], ['=', 1, 'operator'], ['6', 1, 'number'], [';', 1, 'delimeter'], ['int', 2, 'keyword'], ['y', 2, 'identfier'],
 ['=', 2, 'operator'], ['-5', 2, 'number'], [';', 2, 'delimeter'], ['str', 3, 'keyword'], ['z', 3, 'identfier'], ['=', 3, 'operator'], ['"', 3, 'delimeter'], ['scann
er', 3, 'string'], ['"', 3, 'delimeter'], [';', 3, 'delimeter']]

Symbol table: {'x': {'type': 'int', 'value': 6}, 'y': {'type': 'int', 'value': -5}, 'z': {'type': 'str', 'value': 'scanner'}}
No errors were found.
```

Test case 2:

In the case of having two integer numbers when one is negative, and one is positive next to each other.

```
code = """
int x = 6 , y = -5 ;
str z = " scanner " ;
"""
```

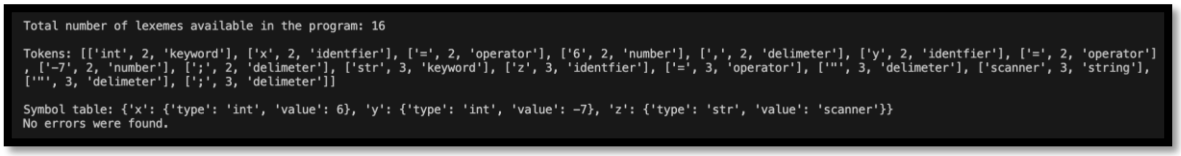


```
Total number of lexemes available in the program: 16

Tokens: [['int', 1, 'keyword'], ['x', 1, 'identfier'], ['=', 1, 'operator'], ['6', 1, 'number'], [',', 1, 'delimeter'], ['y', 1, 'identfier'], ['=', 1, 'operator']
, ['-5', 1, 'number'], [';', 1, 'delimeter'], ['str', 2, 'keyword'], ['z', 2, 'identfier'], ['=', 2, 'operator'], ['"', 2, 'delimeter'], ['scanner', 2, 'string'],
['"', 2, 'delimeter'], [';', 2, 'delimeter']]

Symbol table: {'x': {'type': 'int', 'value': 6}, 'y': {'type': 'int', 'value': -5}, 'z': {'type': 'str', 'value': 'scanner'}}
No errors were found.
```

Test case 3:

In the case of having a comment on the code.

```
code = """
// skip the comment
int x = 6 , y = -7 ;
str z = " scanner " ;
""
```

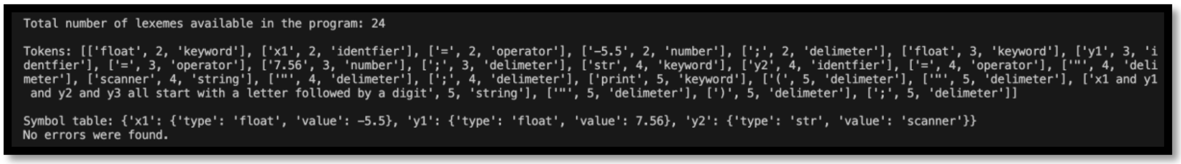


```
Total number of lexemes available in the program: 16

Tokens: [['int', 2, 'keyword'], ['x', 2, 'identfier'], ['=', 2, 'operator'], ['6', 2, 'number'], [',', 2, 'delimeter'], ['y', 2, 'identfier'], ['=', 2, 'operator']
, ['-7', 2, 'number'], [';', 2, 'delimeter'], ['str', 3, 'keyword'], ['z', 3, 'identfier'], ['=', 3, 'operator'], ['"', 3, 'delimeter'], ['scanner', 3, 'string'],
['"', 3, 'delimeter'], [';', 3, 'delimeter']]

Symbol table: {'x': {'type': 'int', 'value': 6}, 'y': {'type': 'int', 'value': -7}, 'z': {'type': 'str', 'value': 'scanner'}}
No errors were found.
```

Test case 4:

In the case of having negative, and positive float numbers.

```
code = """
// skip the comment
float x1 = -5.5 ;
float y1 = 7.56 ;
str y2 = " scanner " ;
print ( " x1 and y1 and y2 and y3 all start with a letter followed
by a digit " ) ;
"""
```

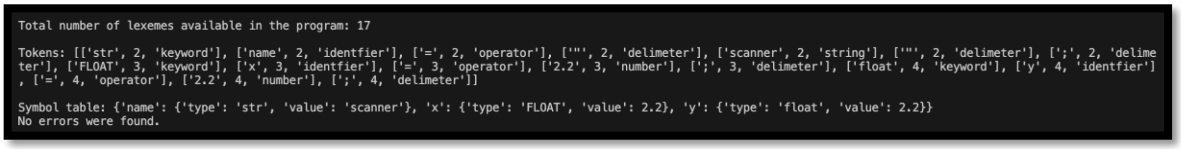


```
Total number of lexemes available in the program: 24

Tokens: [['float', 2, 'keyword'], ['x1', 2, 'identfier'], ['=', 2, 'operator'], ['-5.5', 2, 'number'], [';', 2, 'delimeter'], ['float', 3, 'keyword'], ['y1', 3, 'i
dentfier'], ['=', 3, 'operator'], ['7.56', 3, 'number'], [';', 3, 'delimeter'], ['str', 4, 'keyword'], ['y2', 4, 'identfier'], ['=', 4, 'operator'], ['"', 4, 'deli
meter'], ['scanner', 4, 'string'], ['"', 4, 'delimeter'], [';', 4, 'delimeter'], ['print', 5, 'keyword'], ['(', 5, 'delimeter'], ['"', 5, 'delimeter'], ['x1 and y1
 and y2 and y3 all start with a letter followed by a digit', 5, 'string'], ['"', 5, 'delimeter'], [')', 5, 'delimeter'], [';', 5, 'delimeter']]

Symbol table: {'x1': {'type': 'float', 'value': -5.5}, 'y1': {'type': 'float', 'value': 7.56}, 'y2': {'type': 'str', 'value': 'scanner'}}
No errors were found.
```

Test case 5:

The scanner handles if the keyword is written in uppercase or lowercase by always converting it to lowercase, for example: “ float ”.

```
code = """
// skip the comment
str name = " scanner " ;
FLOAT x = 2.2 ;
float y = 2.2 ;
"""
```

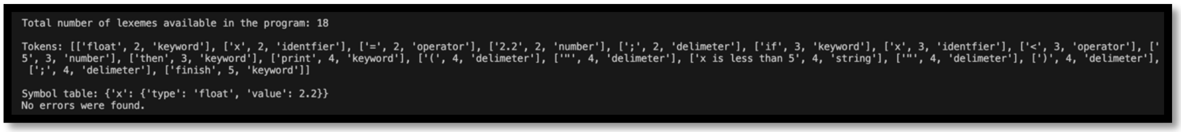


```
Total number of lexemes available in the program: 17

Tokens: [['str', 2, 'keyword'], ['name', 2, 'identfier'], ['=', 2, 'operator'], ['"', 2, 'delimeter'], ['scanner', 2, 'string'], ['"', 2, 'delimeter'], [';', 2, 'delime
ter'], ['FLOAT', 3, 'keyword'], ['x', 3, 'identfier'], ['=', 3, 'operator'], ['2.2', 3, 'number'], [';', 3, 'delimeter'], ['float', 4, 'keyword'], ['y', 4, 'identfier']
, ['=', 4, 'operator'], ['2.2', 4, 'number'], [';', 4, 'delimeter']]

Symbol table: {'name': {'type': 'str', 'value': 'scanner'}, 'x': {'type': 'FLOAT', 'value': 2.2}, 'y': {'type': 'float', 'value': 2.2}}
No errors were found.
```

Test case 6:

In the case of having an effective statement.

```
code = """
// skip the comment
float x = 2.2 ;
if x < 5 then
print ( " x is less than 5 " ) ;
finish
"""
```

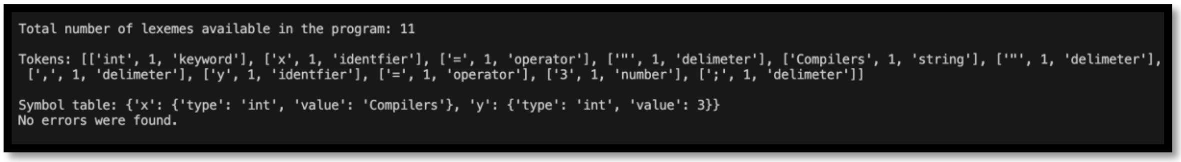


```
Total number of lexemes available in the program: 18

Tokens: [['float', 2, 'keyword'], ['x', 2, 'identfier'], ['=', 2, 'operator'], ['2.2', 2, 'number'], [';', 2, 'delimeter'], ['if', 3, 'keyword'], ['x', 3, 'identfier'], ['<', 3, 'operator'], ['
5', 3, 'number'], ['then', 3, 'keyword'], ['print', 4, 'keyword'], ['(', 4, 'delimeter'], ['"', 4, 'delimeter'], ['x is less than 5', 4, 'string'], ['"', 4, 'delimeter'], [')', 4, 'delimeter'],
 [';', 4, 'delimeter'], ['finish', 5, 'keyword']]

Symbol table: {'x': {'type': 'float', 'value': 2.2}}
No errors were found.
```

Test case 7:

A value is saved based on the data type the user selects when a value is kept in a simple table.

```
code = """
int x = " Compilers " , y = 3 ;
"""
```

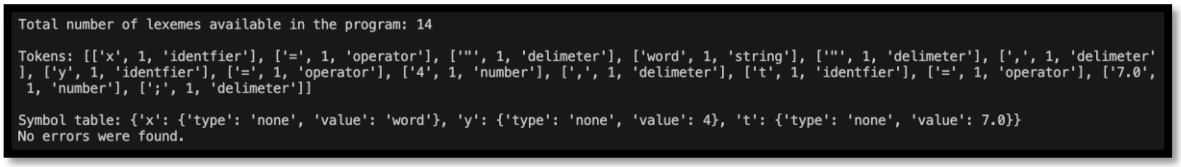


```
Total number of lexemes available in the program: 11

Tokens: [['int', 1, 'keyword'], ['x', 1, 'identfier'], ['=', 1, 'operator'], ['"', 1, 'delimeter'], ['Compilers', 1, 'string'], ['"', 1, 'delimeter'],
 [',', 1, 'delimeter'], ['y', 1, 'identfier'], ['=', 1, 'operator'], ['3', 1, 'number'], [';', 1, 'delimeter']]

Symbol table: {'x': {'type': 'int', 'value': 'Compilers'}, 'y': {'type': 'int', 'value': 3}}
No errors were found.
```

Test case 8:

when the user forgets to specify the data type, the word "none" for type is stored in the simple table.

```
code = """
x = " word " , y = 4 , t = 7.0 ;
"""
```

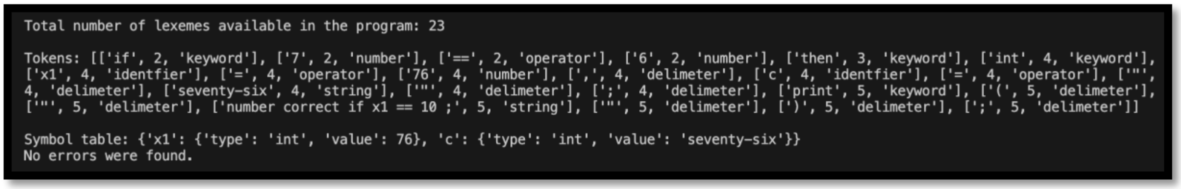


```
Total number of lexemes available in the program: 14

Tokens: [['x', 1, 'identfier'], ['=', 1, 'operator'], ['"', 1, 'delimeter'], ['word', 1, 'string'], ['"', 1, 'delimeter'], [',', 1, 'delimeter'
], ['y', 1, 'identfier'], ['=', 1, 'operator'], ['4', 1, 'number'], [',', 1, 'delimeter'], ['t', 1, 'identfier'], ['=', 1, 'operator'], ['7.0',
 1, 'number'], [';', 1, 'delimeter']]

Symbol table: {'x': {'type': 'none', 'value': 'word'}, 'y': {'type': 'none', 'value': 4}, 't': {'type': 'none', 'value': 7.0}}
No errors were found.
```

Test case 9:

Handling data types within the print statement.

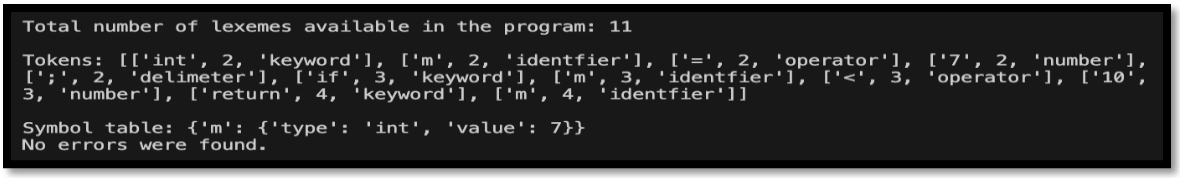
```
code = """
// Skip the comment
if 7 == 6
then
int x1 = 76 , c = " seventy-six " ;
print ( " number correct if x1 == 10 ; " ) ;
"""
```



```
Total number of lexemes available in the program: 23

Tokens: [['if', 2, 'keyword'], ['7', 2, 'number'], ['==', 2, 'operator'], ['6', 2, 'number'], ['then', 3, 'keyword'], ['int', 4, 'keyword'],
['x1', 4, 'identfier'], ['=', 4, 'operator'], ['76', 4, 'number'], [',', 4, 'delimeter'], ['c', 4, 'identfier'], ['=', 4, 'operator'], ['"',
4, 'delimeter'], ['seventy-six', 4, 'string'], ['"', 4, 'delimeter'], [';', 4, 'delimeter'], ['print', 5, 'keyword'], ['(', 5, 'delimeter'],
['"', 5, 'delimeter'], ['number correct if x1 == 10 ;', 5, 'string'], ['"', 5, 'delimeter'], [')', 5, 'delimeter'], [';', 5, 'delimeter']]

Symbol table: {'x1': {'type': 'int', 'value': 76}, 'c': {'type': 'int', 'value': 'seventy-six'}}
No errors were found.
```

Test case 10: Return value.

```
code = """
// skip the comment
int m = 7 ;
if m < 10
return m
"""
```

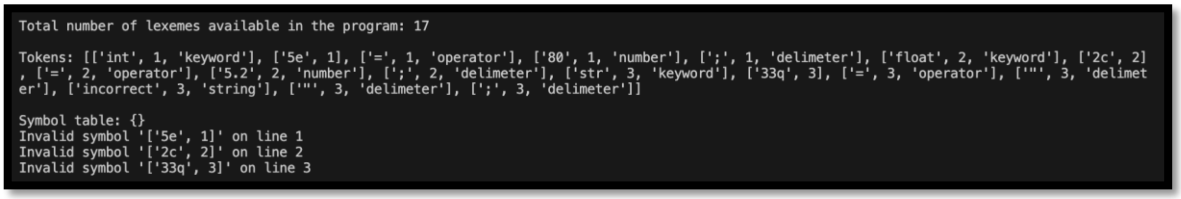


```
Total number of lexemes available in the program: 11

Tokens: [['int', 2, 'keyword'], ['m', 2, 'identfier'], ['=', 2, 'operator'], ['7', 2, 'number'],
[';', 2, 'delimeter'], ['if', 3, 'keyword'], ['m', 3, 'identfier'], ['<', 3, 'operator'], ['10',
3, 'number'], ['return', 4, 'keyword'], ['m', 4, 'identfier']]

Symbol table: {'m': {'type': 'int', 'value': 7}}
No errors were found.
```

2.2 Invalid Test Cases

Test case 1:
In this test case (Invalid symbol) if the user enters the number before the Identifier.

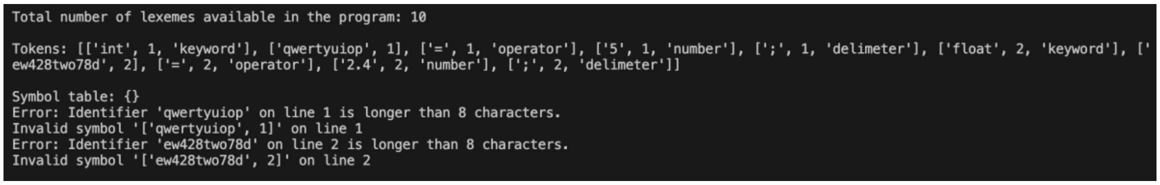
```
code = """
int 5e = 80 ;
float 2c = 5.2 ;
str 33q = " incorrect " ;
"""
```



```
Total number of lexemes available in the program: 17

Tokens: [['int', 1, 'keyword'], ['5e', 1], ['=', 1, 'operator'], ['80', 1, 'number'], [';', 1, 'delimeter'], ['float', 2, 'keyword'], ['2c', 2]
, ['=', 2, 'operator'], ['5.2', 2, 'number'], [';', 2, 'delimeter'], ['str', 3, 'keyword'], ['33q', 3], ['=', 3, 'operator'], ['"', 3, 'delimet
er'], ['incorrect', 3, 'string'], ['"', 3, 'delimeter'], [';', 3, 'delimeter']]

Symbol table: {}
Invalid symbol '['5e', 1]' on line 1
Invalid symbol '['2c', 2]' on line 2
Invalid symbol '['33q', 3]' on line 3
```

Test case 2:
In this test case (Error Identifier is longer than 8 characters.) if the user enters the Identifier longer than 8 characters.

```
code = """
int qwertyuiop = 5 ;
float ew428two78d = 2.4 ;
"""
```

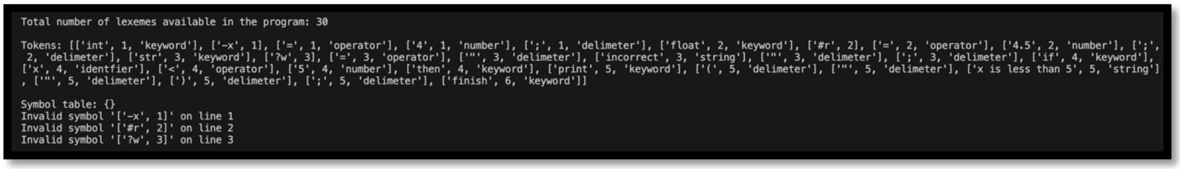


```
Total number of lexemes available in the program: 10

Tokens: [['int', 1, 'keyword'], ['qwertyuiop', 1], ['=', 1, 'operator'], ['5', 1, 'number'], [';', 1, 'delimeter'], ['float', 2, 'keyword'], ['
ew428two78d', 2], ['=', 2, 'operator'], ['2.4', 2, 'number'], [';', 2, 'delimeter']]

Symbol table: {}
Error: Identifier 'qwertyuiop' on line 1 is longer than 8 characters.
Invalid symbol '['qwertyuiop', 1]' on line 1
Error: Identifier 'ew428two78d' on line 2 is longer than 8 characters.
Invalid symbol '['ew428two78d', 2]' on line 2
```

Test case 3:

In this test case (Invalid symbol) if the user enters the operator or symbol before the Identifier.

```
code = """
int -x = 4 ;
float #r = 4.5 ;
str ?w = " incorrect " ;
if x < 5 then
print ( " x is less than 5 " ) ;
finish
"""
```

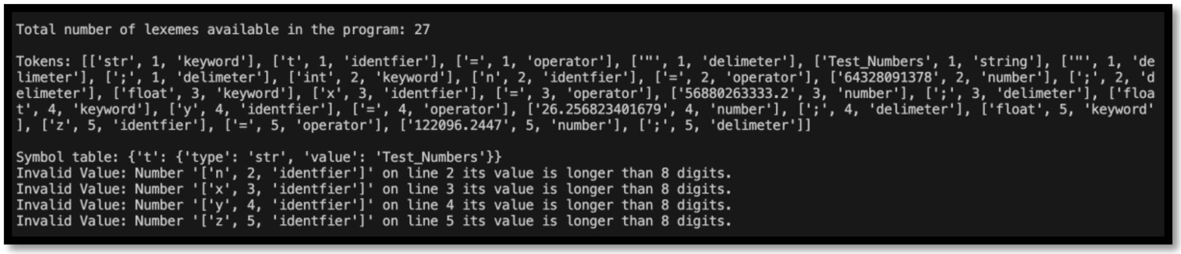


```
Total number of lexemes available in the program: 30

Tokens: [['int', 1, 'keyword'], ['-x', 1], ['=', 1, 'operator'], ['4', 1, 'number'], [';', 1, 'delimeter'], ['float', 2, 'keyword'], ['#r', 2], ['=', 2, 'operator'], ['4.5', 2, 'number'], [';',
 2, 'delimeter'], ['str', 3, 'keyword'], ['?w', 3], ['=', 3, 'operator'], ['"', 3, 'delimeter'], ['incorrect', 3, 'string'], ['"', 3, 'delimeter'], [';', 3, 'delimeter'], ['if', 4, 'keyword'],
['x', 4, 'identfier'], ['<', 4, 'operator'], ['5', 4, 'number'], ['then', 4, 'keyword'], ['print', 5, 'keyword'], ['(', 5, 'delimeter'], ['"', 5, 'delimeter'], ['x is less than 5', 5, 'string']
, ['"', 5, 'delimeter'], [')', 5, 'delimeter'], [';', 5, 'delimeter'], ['finish', 6, 'keyword']]

Symbol table: {}
Invalid symbol '['-x', 1]' on line 1
Invalid symbol '['#r', 2]' on line 2
Invalid symbol '['?w', 3]' on line 3
```

Test case 4:

In this test case (Invalid value number, value is longer than 8 digits.) If the user types a number that is longer than 8 digits.

```
code = """
str t = " Test_Numbers " ;
int n = 64328091378 ;
float x = 56880263333.2 ;
float y = 26.256823401679 ;
float z = 122096.2447 ;
"""
```



```
Total number of lexemes available in the program: 27

Tokens: [['str', 1, 'keyword'], ['t', 1, 'identfier'], ['=', 1, 'operator'], ['"', 1, 'delimeter'], ['Test_Numbers', 1, 'string'], ['"', 1, 'de
limeter'], [';', 1, 'delimeter'], ['int', 2, 'keyword'], ['n', 2, 'identfier'], ['=', 2, 'operator'], ['64328091378', 2, 'number'], [';', 2, 'd
elimeter'], ['float', 3, 'keyword'], ['x', 3, 'identfier'], ['=', 3, 'operator'], ['56880263333.2', 3, 'number'], [';', 3, 'delimeter'], ['floa
t', 4, 'keyword'], ['y', 4, 'identfier'], ['=', 4, 'operator'], ['26.256823401679', 4, 'number'], [';', 4, 'delimeter'], ['float', 5, 'keyword'
], ['z', 5, 'identfier'], ['=', 5, 'operator'], ['122096.2447', 5, 'number'], [';', 5, 'delimeter']]

Symbol table: {'t': {'type': 'str', 'value': 'Test_Numbers'}}
Invalid Value: Number '['n', 2, 'identfier']' on line 2 its value is longer than 8 digits.
Invalid Value: Number '['x', 3, 'identfier']' on line 3 its value is longer than 8 digits.
Invalid Value: Number '['y', 4, 'identfier']' on line 4 its value is longer than 8 digits.
Invalid Value: Number '['z', 5, 'identfier']' on line 5 its value is longer than 8 digits.
```

4. References

https://www.w3schools.com/python/python_ref_tuple.asp

https://www.w3schools.com/python/ref_string_isalpha.asp

https://www.w3schools.com/python/ref_string_isalnum.asp

https://pynative.com/python-count-number-of-lines-in-file/

https://www.youtube.com/watch?v=nZfovY1KoPo&list=LL&index=31&t=7s