Compilers Project Report

CSE439: Design of Compilers
Team 10

MEMBERS

Adham Hisham Kandil

22P0217

Mohamed Yehia Zakria

22P0064

Jana Hany El Shafie

22P0235

Seif ElHusseiny

22P0215

Mohamed Ahmed Abdelhamid

22P0287

Moaz Mohamed Zakaria

22P0307

Represented to

Dr. Wafaa Samy
Professor, Computer and Systems Engineering

Eng. Mohammed Abdelmegeed
Teaching Assistant, Computer and Systems Engineering

Table of Contents

INTRODUCTION 2

Python Specifications 3

- **KEYWORDS & VARIABLE IDENTIFIERS** 3
- **FUNCTION IDENTIFIERS** 5
- **DATA TYPES** 7
- **Functions** 9
- **Statements** 11
- **Expressions** 19

Lexical Analyzer 23

Syntax Analyzer 28

Full Project Implementation 31

- **Lexical Code Implementation** 31
- **Syntax Code Implementation** 62

Graphical User Interface 76

Testcases 77

Conclusion 1

References 4

INTRODUCTION

In today's world, programming languages like Python play a big role in technology, education, and research.

Our project is divided into three main pieces. First, the lexical analyzer (lexer) scans the raw Python code, breaking it down into meaningful "tokens" like keywords, variables, operators, and numbers. The lexer also keeps track of identifiers and their properties in a symbol table, helping us remember important details about every variable or function.

Next comes the syntax analyzer (parser), which checks whether the code follows the correct structure and grammar rules of Python. If the input code makes sense, the parser builds a "parse tree"—a kind of family tree that shows how all the parts of the program fit together. If there's a mistake, the parser points it out, telling the user exactly where things went wrong.

Finally, we wanted our tool to be easy and fun to use, so we built a graphical user interface (GUI) using Qt. With it, users can write Python code, press a button to analyze it, and instantly see the tokens, errors, symbol table, and parse tree—all in one place.

Throughout this report, we will walk you through each stage of our project: the theory, the design decisions we made, the challenges we faced, and the solutions we found. Our hope is to make compiler construction less intimidating, more approachable, and even enjoyable for anyone interested in how programming languages actually work.

Python Specifications

KEYWORDS & VARIABLE IDENTIFIERS

1.1 Keywords

The identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

Each keyword has a distinct role in Python programming, defining the structure and logic of programs.

Common Python Keywords and Their Uses:

1. **True, False** – Represent Boolean values.
2. **None** – Represents the absence of a value (null).
3. **and, or, not** – Logical operators for Boolean expressions.
4. **if, else, elif** – Implements conditional statements.
5. **except, raise, try, finally** – Used for exception handling.
6. **return** – Specifies the return value of a function.
7. **def** – Declares a function.
8. **for, while** – Implements loops for iteration.
9. **rom, import** – Used for importing modules.

There is description for the use of other keywords, I listed these because I will use them in the following Examples

1.2 Variables

A variable can have a simple short name (alphabetically like a, b, x , y) or a more descriptive one like (name , job , CalculateAVG, etc....)

Syntax when choosing a variable Name

- **Must start with a letter (A-Z, a-z) or an underscore (_)**
- **Can only contain letters, digits (0-9), and underscores (_)**
- **Cannot start with a number**
- **Cannot contain spaces** (Use underscores instead)
- **Cannot use special characters (@, $, %, etc.)**
- **Case-sensitive** (Python treats uppercase and lowercase letters differently)
- **Cannot be a Python keyword (e.g., if, for, while)**

Valid Statements ✅	***Invalid Statements*** ❌
my_var = 10	2name = "A"
_count = 5	user name = 3
price_2024 = 99	for = 10
UserAge = 30	total-price = 15
PI = 3.14159	@total = 100
num1 = 10	my-var = 50

FUNCTION IDENTIFIERS

Function identifiers are the names assigned to functions, allowing them to be defined, called, and referenced within code. Choosing appropriate function names is essential for ensuring clarity, readability, and maintainability. This documentation outlines the rules, best practices, and conventions for naming function identifiers in Python.

- **Must start with a letter (a-z, A-Z) or an underscore (_)**
- **Can contain letters, numbers (0-9), and underscores (_)**
- **Case-sensitive (Python treats uppercase and lowercase as different)**
- **Cannot use Python keywords (e.g., if, for, while)**

Examples:

valid

- def my_function(): #starts with a letter
- def _calculate_total(): #start with underscore

- def get_user_info():
- def sum_of_2_numbers(): # can contain numners

- def MYFUNCTION(): # python treates upper & lower case differently
- def myfunction()

Invalid

- def for(): # cannot use keywords
- def return():

2.1 Scope of Function Identifiers

The **scope** of a function identifier refers to the area of the program where the function name is accessible. In Python, function identifiers can have **local** or **global** scope depending on how and where they are defined.

2.1.1 Local Scope (Function-Level Scope)

*A function identifier defined inside a function has **local scope**. It can only be accessed within the function where it is defined and cannot be used outside of it.*

2.1.2 Global Scope

*A function defined at the top level of a script or module has **global scope**. It can be called and accessed from anywhere in the program.*

2.1.3 Modifying Global Variables Inside a Function

By default, functions cannot modify global variables unless explicitly declared using the global keyword

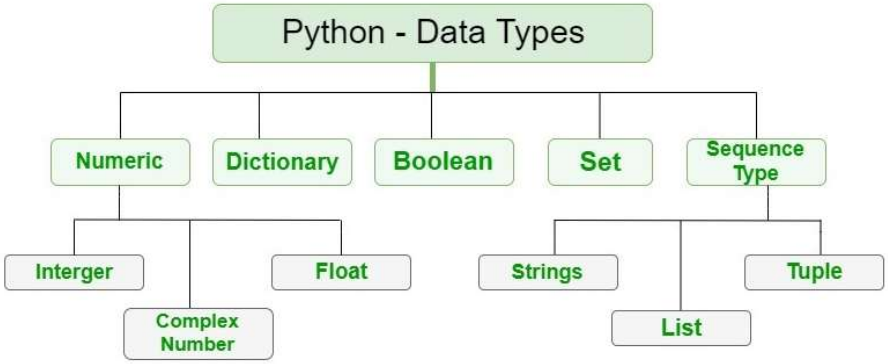
2.1.4 Nonlocal Scope (Nested Functions)

In nested functions, a variable declared in the **outer function** but not global can be modified using nonlocal. This allows the inner function to modify a variable from its enclosing function

```
# Global variable (accessible anywhere)
global_var = "I am global"

def outer_function():
    outer_var = "I am outer"  # Local to outer_function

    def inner_function():
        nonlocal outer_var  # Refers to outer_function's variable
        outer_var = "Modified by inner function"
        print("Inside inner_function:", outer_var)

        # Accessing global variable inside inner_function
        global global_var
        global_var = "Modified globally"
        print("Inside inner_function (global):", global_var)

    inner_function()
    print("Inside outer_function:", outer_var)

# Calling the functions
print("Before function calls:", global_var)
outer_function()
print("After function calls:", global_var)

```

DATA TYPES

Python provides various built-in data types that define the type of values a variable can hold. These data types can be categorized into the following groups:



3.1 Numeric Data Types

- int (Integer): Whole numbers (e.g., 10, -5).
- float (Floating-point): Numbers with decimals (e.g., 3.14, -0.99)
- complex: Numbers with real and imaginary parts (e.g., 2 + 3j).

```
a = 10    pi = 3.14159
b = -5    temperature = -10.5
c = 0
```

3.2.Boolean Data Type

Represents True or False values and is a subclass of int,
where True == 1 and False == 0.

```
is_valid = True
is_empty = False
```

3.3 Sequence Data Types

- **Strings (`str`)**: . Immutable sequences of Unicode characters used to store text Strings are indexed starting from 0 and -1 from the end. This allows us to retrieve specific characters from the string

```
name = "Alice"
print(name[0])  # Output: A
```

- **Lists (`list`)**: Ordered, mutable collections that can hold elements of different types.

```
numbers = [1, 2, 3, 4]
numbers.append(5)
print(numbers)  # Output: [1, 2, 3, 4, 5]
```

- **Tuples (`tuple`)**: Ordered, immutable collections used for fixed collections of items.

```
coordinates = (10, 20)
print(coordinates[0])  # Output: 10
```

3.4 Set Data Types

- **Set (set)**: Unordered collection of unique elements, useful for eliminating duplicates and performing set operations.

Example:

```
unique_numbers = {1, 2, 3, 3}
print(unique_numbers)  # Output: {1, 2, 3}
```

- **Frozen Set (frozenset)**: An immutable version of a set.

A frozenset in Python is a built-in data type that is similar to a set but with one key difference: immutability. This means that once a frozenset is created, we cannot modify its elements—that is, we cannot add, remove, or change any items in it. Like regular sets, a frozenset cannot contain duplicate elements.

If no parameters are passed, it returns an empty frozenset

3.5 Dictionary Data Type

Dictionary (dict): Collection of key-value pairs with fast lookups and dynamic size.

Example:

```
student = {"name": "Alice", "age": 25}
print(student["name"])  # Output: Alice
```

dictionary can be created by placing a sequence of elements within curly {} braces, separated by a comma.

we can access a value from a dictionary by using the key within square brackets or get () method.

we can remove items from a dictionary using the following methods:

- **del: Removes an item by key.**
- **pop(): Removes an item by key and returns its value.**
- **clear(): Empties the dictionary.**
- **popitem(): Removes and returns the last key-value pair.**

Functions

In Python, functions are fundamental building blocks that allow you to organize and reuse code

- Functions are blocks of code designed to perform specific tasks. Instead of writing the same code repeatedly, you can encapsulate it within a function and call it whenever needed.
- They promote modularity by breaking down complex programs into smaller, manageable units. This makes code easier to read, understand, and maintain.
- Functions can accept input values (arguments or parameters) and return output values.

4.1 Types of Functions in Python

4.1.1 Built-in Functions

• Description:

- These are functions that are readily available in Python without requiring any additional imports.

• Examples:

- print(): Outputs the specified object(s) to the standard output device (the screen).
- len(): Returns the length (number of items) of an object (string, list, tuple,etc.).
- type(): Returns the type of an object.
- range(): Generates a sequence of numbers.
- abs(): Returns the absolute value of a number.
- sum(): Returns the sum of the items in an iterable (list, tuple, etc.).
- max(): Returns the largest item in an iterable.
- min(): Returns the smallest item in an iterable.
- input(): Reads a line from standard input (the keyboard) and returns it as a string.

4.1.2 User-defined Functions

• Description:

- These are functions that you create to perform specific tasks tailored to your program's needs, you define them using the def keyword

Example:
calculate_rectangle_area(): takes two parameters, length and width, It calculates the area by multiplying them, The return statement sends the calculated area back to where the function was called, This example shows a function that returns a value.

```
def calculate_rectangle_area(length, width):
"""This function calculates the area of a rectangle."""
area = length * width
return area

 # Calling the function and storing the result
 rectangle_area = calculate_rectangle_area(5, 10)
print("The area of the rectangle is:", rectangle_area)
```

4.1.3 Lambda Functions (Anonymous Functions)

• Description:

- These are small, single-expression functions defined using the lambda keyword. They are often used for short, simple operations.

Example:

Adding Two Numbers: This lambda function takes two arguments, x and y, and returns their sum, It's assigned to the variable add, which can then be called like a regular function.

```
add = lambda x, y: x + y
result = add(5, 5)
print(result) # Output: 10
```

4.1.4 Recursive Functions

• Description:

- These are functions that call themselves within their own definition. This is very useful for problems that can be broken down into smaller, self similar problems.

Example:
Fibonacci Sequence: The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, The function checks if n is 0 or 1 (base cases). If it is, it returns n, Otherwise, it returns the sum of the (n-1)th and (n-2)th Fibonacci numbers, Again, this breaks the problem into smaller self similar problems.

```
def fibonacci(n):
"Calculates the nth Fibonacci number."
if n <= 1:
return n
else:
 return fibonacci(n - 1) + fibonacci(n - 2)
result = fibonacci(6)
print(result) # Output: 8 (0, 1, 1, 2, 3, 5, 8)
```

Statements

In Python, assignment statements are fundamental for binding names (variables) to objects. They're how you store and manipulate data.

5.1 Assignment Statements

The most common form uses the equals sign (=) to assign a value to a variable, this creates a reference from the variable name to the object in memory.

- **x = 10**
- **name = "Python"**
- **my_list = [1, 2, 3]**

5.1.1 Multiple Assignment

You can assign the same value to multiple variables simultaneously.

- **x = y = 5**

You can assign multiple values to multiple variables at the same time.

- **x, y, z = 1, 2, 3**

5.1.2 Augmented Assignment

These operators combine an arithmetic operation with an assignment.

- **+=** Addition assignment (e.g., x += 5 is equivalent to x = x + 5)
- **-=** Subtraction assignment (e.g., y -= 2 is equivalent to y = y - 2)
- ***=** Multiplication assignment (e.g., z *= 3 is equivalent to z = z * 3)
- **/=** Division assignment (e.g., a /= 4 is equivalent to a = a / 4)
- **//=** Floor division assignment (e.g., b //= 2 is equivalent to b = b // 2)
- **%=** Modulus assignment (e.g., c %= 3 is equivalent to c = c % 3).
- ****=** Exponentiation assignment (e.g., d **= 2 is equivalent to d = d **

5.1.3 Sequence Unpacking

You can unpack the elements of a sequence (like a tuple or list) into individual variables.

- **a, b = (10, 20)**

5.2 Declarations

In Python, declaration statements are not explicitly defined as a separate concept like in some other programming languages. Instead, variable declarations are done implicitly during variable assignments. Python is dynamically typed, meaning variables don't need to be declared with a type upfront, and the type is inferred from the assigned value.

5.2.1 Declaring Variables via Assignment

```
city = "New York"
price = 19.99
```

5.2.2 String Initialization

- Using Single or Double Quotes:

```
message1 = 'Hello'
message2 = "World"
```

- Using Triple Quotes for Multi-line Strings:

```
long_text = '''Python supports multi-line strings.
```

They are useful for documentation.'''

5.2.3 Declaring Conditional Statements

- *Using an if Statement:*

```
score = 75
if score >= 50:
  print("Pass")
```

- *Using if-else for Decision Making:*

```
time = 18
if time < 12:
  print("Good morning")
else:
  print("Good evening")
```

5.2.4 Defining Functions

Functions encapsulate reusable logic.

```
def greet():
  print("Welcome to Python!")
greet()
```

5.2.5 Declaring Data Structures

- **Lists** (Ordered, mutable collection):
  numbers = [1, 2, 3, 4, 5]

- **Tuples** (Ordered, immutable collection):
  dimensions = (1920, 1080)

- **Sets** (Unordered collection of unique elements):
  unique_colors = {"red", "blue", "green"}

- **Dictionaries** (Key-value pairs):
  person = {"name": "Alice", "age": 30}

5.2.6 Declaring Classes and Objects

Classes define object blueprints.

```
class Animal:
  def __init__(self, species, sound):
    self.species = species
    self.sound = sound
dog = Animal("Dog", "Bark")
print(dog.sound)  # Output: Bark
```

5.3 Return Statement

A return statement is used to end the execution of the function call and it "returns" the value of the expression following the return keyword to the caller. The statements after the return statements are not executed. If the return statement is without any expression, then the special value None is returned. A return statement is overall used to invoke a function so that the passed statements can be executed.

5.3.1 Basic Return Syntax

```
def square(n):
  return n * n
print(square(4))  # Output: 16
```

5.3.2 Returning Multiple Values

A function can return multiple values as a tuple.

```
def user_info():
  return "Alice", 25
name, age = user_info()
print(name, age)  # Output: Alice 25
```

5.3.3 Returning Lists and Dictionaries

```
def generate_sequence(n):
  return [n, n*2, n*3]
print(generate_sequence(5))  # Output: [5, 10, 15]
```

5.3.4 Returning a Function from Another Function

```
def multiply_by(factor):
  def multiplier(value):
    return value * factor
  return multiplier

double = multiply_by(2)
print(double(10))  # Output: 20
```

5.3.5 Using pass in Functions

The pass statement acts as a placeholder for future implementation.

```
def future_feature():
  pass  # Placeholder for code to be added later
```

5.3.6 Recursive Function Calls

Functions can call themselves for problems like factorial calculations.

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1)

print(factorial(6))  # Output: 720
```

5.4 Iterative Statements

Iterative statements, also known as loops, facilitate repeated execution of a specific code as long as a condition is true.

5.4.1 for Loop

Iterates over iterable data structures such as lists, tuples, dictionaries, and strings. For loops are mainly used for sequential traversal

```
for i in range(5):
print(i)
```

5.4.2 while Loop

Continuously executes a block of code as long as the specified condition remains True.

```
 x = 0
 while x < 5:
print(x)
 x += 1
```

while loops can also have an else statement which executes after the condition becomes False

5.5.3 Nested Loops

Loops can be nested to allow for extended behavior. In this example, the nested for loop allows iteration across multiple dimensions or layers.

```
 for i in range(3):
for j in range(2):
print(f"i: {i}, j: {j}")
```

5.5 Conditional Statements

Conditional statements allow a program to make decisions based on given conditions. These statements enable logical branching, executing different code blocks depending on whether a condition evaluates to True or False.
Python provides structured decision-making through the following conditional constructs:

5.5.1 if Statement

The if statement executes a block of code only when a specified condition is True.
Example:

```
age = 20
if age >= 18:
   print("You are eligible to vote.")
```

Output:
You are eligible to vote.

5.5.2 if-else Statement

The if-else structure provides an alternative execution path when the condition evaluates to False.
Example:

```
temperature = 15
if temperature > 20:
   print("It's warm outside.")
else:
   print("It's cold outside.")
```

Output:
It's cold outside.

5.5.3 The if-elif-else Statement

The if-elif-else construct evaluates multiple conditions sequentially. The first True condition is executed, and the rest are ignored.
Example:

```
grade = 85
if grade >= 90:
   print("Grade: A")
elif grade >= 80:
   print("Grade: B")
```

```
elif grade >= 70:
  print("Grade: C")
elif grade >= 60:
  print("Grade: D")
else:
  print("Grade: F")
```
Output:
Grade: B

5.5.5 Nested if Statements

An if statement can be placed inside another if statement for hierarchical decision-making.
Example:
```
age = 19
if age >= 18:
  if age < 21:
    print("You are an adult but cannot drink alcohol in some countries.")
  else:
    print("You are fully an adult.")
```
Output:
You are an adult but cannot drink alcohol in some countries.

5.6 Function Call Statement

A function call statement executes a function by referencing its name followed by parentheses. If the function requires arguments, they are provided inside the parentheses.

8.1 Calling a Function Without Arguments

```
def greet():
  print("Hello, welcome to Python!")

greet()
```
Output:
Hello, welcome to Python!

8.2 Calling a Function With Arguments

```
def add(a, b):
  return a + b

result = add(5, 7)
print(result)
```
Output:
12

8.3 Calling a Function With Default Arguments

```
def introduce(name="Guest"):
  print(f"Hello, {name}!")

introduce()  # Uses default value
introduce("Alice")
```

Output:

```
Hello, Guest!
Hello, Alice!
```

8.4 Calling a Function With Keyword Arguments

```
def describe_pet(animal, name):
  print(f"I have a {animal} named {name}.")

describe_pet(animal="dog", name="Buddy")
describe_pet(name="Whiskers", animal="cat")
```

Output:

```
I have a dog named Buddy.
I have a cat named Whiskers.
```

*8.5 Calling a Function With Arbitrary Arguments (*args)*

```
def print_numbers(*numbers):
  for num in numbers:
    print(num)

print_numbers(1, 2, 3, 4, 5)
```

Output: 1 2 3 4 5

8.6 Calling a Function That Returns a Value

```
def multiply(a, b):
  return a * b

product = multiply(4, 3)
print(product)
```

Output:

```
12
```

Expressions

Python supports various arithmetic operations, including basic mathematical operations, exponentiation, and modulo. These operations work with integers, floating-point numbers, and complex numbers.

6.1 Arithmetic Expression

Python supports various arithmetic operations, including basic mathematical operations, exponentiation, and modulo. These operations work with integers, floating-point numbers, and complex numbers.

6.1.1 Basic Arithmetic Operators

Operator	Description	Example	Output
+	Addition	7 + 2	9
-	Subtraction	9 - 4	5
*	Multiplication	6 * 3	18
/	Division (Floating-point)	8 / 2	4.0

Example:

```
x = 12
y = 4
print(x + y) # Output: 16
print(x - y) # Output: 8
print(x * y) # Output: 48
print(x / y) # Output: 3.0
```

6.1.2 Floor Division (//)

- Returns the quotient rounded down to the nearest integer.
- Works with both integers and floats.

Example:

```
print(15 // 4) # Output: 3
print(10.8 // 3) # Output: 3.0
```

6.1.3 Modulo (%)

- Returns the remainder of division.

Example:

```
print(17 % 5) # Output: 2
print(22 % 7) # Output: 1
```

6.1.4 Exponentiation (**)

- Raises a number to a power.

Example:

```
print(3 ** 3) # Output: 27
print(16 ** 0.5) # Output: 4.0 (square root)
```

6.1.5 Operator Precedence (Order of Execution)

Python follows PEMDAS (Parentheses, Exponents, Multiplication/Division, Addition/Subtraction):

Precedence	Operators
1 (Highest)	() (Parentheses)
2	** (Exponentiation)
3	*, /, //, % (Multiplication, Division, Modulo)
4 (Lowest)	+, - (Addition, Subtraction)

Example:

```
print(3 + 4 * 2) # Output: 11 (Multiplication first)
print((3 + 4) * 2) # Output: 14 (Parentheses first)
```

6.1.6 Unary Operations

- +x: Positive sign (no effect)
- -x: Negation (changes sign)

Example:

```
print(+7) # Output: 7
print(-7) # Output: -7
```

6.1.7 Augmented Assignment Operators

These combine an arithmetic operation with assignment.
Mentioned before

6.2 Boolean Expressions

A Boolean expression evaluates to either True or False. These expressions are commonly used in conditional statements, loops, and logical operations.

6.2.1 Boolean Values

Python provides two built-in Boolean values:

- True: Represents a true or affirmative state.
- False: Represents a false or negative state.

These values are case-sensitive, meaning true and false are invalid.

Example:

```
print(True)  # Output: True
print(False)  # Output: False
```

6.2.3 Boolean Operators

Python supports three logical operators that manipulate Boolean values:

Operator	Description	Example	Output
and	Logical AND	True and False	False
or	Logical OR	False or True	True
not	Logical NOT	not False	True

Example:

```
x = 8
y = 12
print(x > 5 and y < 15)  # Output: True
print(x > 10 or y > 15)  # Output: False
print(not (x < y))      # Output: False
```

6.2.4 Comparison Operators

Comparison operators evaluate conditions and return True or False based on the result of the comparison.

Operator	Description	Example	Output
==	Equal to	10 == 10	True
!=	Not equal to	7 != 9	True
>	Greater than	12 > 8	True
<	Less than	3 < 1	False
>=	Greater than or equal	6 >= 6	True
<=	Less than or equal	4 <= 2	False

Example:

```
age = 20
height = 175
print(age >= 18)   # Output: True
print(height < 160)  # Output: False
print(age != 25)   # Output: True
```

6.2.5 Operator Precedence

Boolean expressions in Python follow a precedence order:

Precedence	Operators
1 (Highest)	not
2	and
3 (Lowest)	or

Example:

```
print(True or False and False)  # Output: True (AND executes before OR)
print(not True or False)       # Output: False (NOT executes first)
print((False or True) and True)  # Output: True
```

6.2.6 Boolean Short-Circuiting

Python optimizes Boolean expressions using short-circuit evaluation:

- In x and y, if x is False, y is not evaluated.
- In x or y, if x is True, y is not evaluated.

Example:

```
def sample_function():
  print("Function executed")
  return True

print(False and sample_function())  # Output: False (Function is not called)
print(True or sample_function())    # Output: True (Function is not called)
```

6.2.7 Chained Comparisons

Python allows multiple comparisons in a single statement.

Example:

```
score = 85
print(50 < score < 100)  # Output: True (equivalent to 50 < score and score < 100)
```

Lexical Analyzer

Lexical Analyzer : Responsible for generating a stream of tokens <name,value> issued for each lexeme (chunk of text). It is also responsible for constructing the symbol Table.

For our purpose, we have considered 8 Lexical Categories. Actual Number of lexical categories, may vary and can be quite large.

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Figure 3.12: Tokens, their patterns, and attribute values

List Order of Lexical Categories with the types of token's issued by them :

Types of categories/NFAs	**Token issued <category name, value>**
• Keywords	• <if> <while> <else>.....
• Variables	• <id,...> eg. <id,3> [3rd row in symbol table]
• Numbers	• <num,60>
• Relational operators	• <relop,GT> for the lexeme >
• Arithmetic operators	• <arith,PLUS> for the lexeme +
• Parenthesis	• <PAREN, S_OPEN> for the lexeme (

Here I have implemented NFA's using Switch-case statements. However, the NFA for Keywords is implemented using Trie data-structure.

Also I have implemenented a Symbol Table using Class and a Hash-Map. Symbol table stores the variables and issues token number for variables depending on the row number.

Also, for generating tokens and navigating through the source text file, we have two techniques : Serial and Parallel.

If for a lexemme, no token is issues a Lexical Error is declared !

NFA's Example

Relational Operators category : RELOP

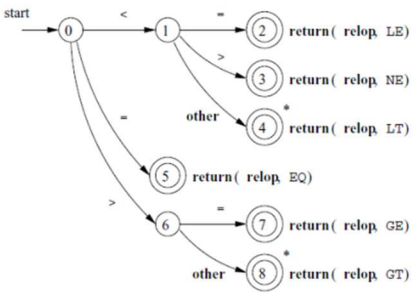


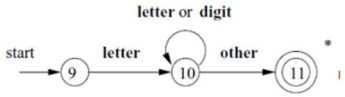
Figure 3.13: Transition diagram for **relop**

- Regular expression for NFA:

<=|<>|<|=|>=|>

- Meaning of * symbol along with **other** (final State 4, 8): RETRACT (go back one step in the input stream)

Variable

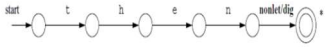


- Eg. A, b, a2c: <id,1><id,2><id,3>
- Regular expression for NFA:

letter.(letter|digit)*

- RETRACT (*)
- Eg. a2c=b+c

Keywords



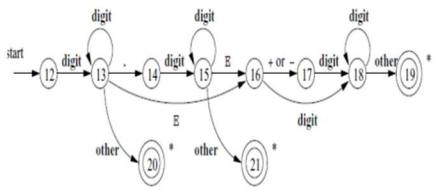
- *Eg. For the lexemes: respective tokens issued are: |*
- **Regular expression for NFAs**:

$$\begin{aligned} if &\rightarrow \texttt{if} \\ then &\rightarrow \texttt{then} \\ else &\rightarrow \texttt{else} \end{aligned}$$

- RETRACT (*)-input pointer move back one position

Numbers

Whitespaces



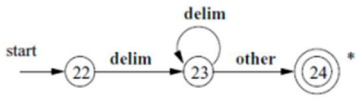
- *Eg. For the lexemes: respective tokens issued are:*

<num,1> <num,2> <num,3>

- **Regular expression for NFA**:

$$\begin{aligned} digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ digits &\rightarrow digit\ digit^{*} \\ optionalFraction &\rightarrow \texttt{.}\ digits \mid \epsilon \\ optionalExponent &\rightarrow (\ \texttt{E}\ (\ \texttt{+} \mid \texttt{-} \mid \epsilon\ )\ digits\ ) \mid \epsilon \\ number &\rightarrow digits\ optionalFraction\ optionalExponent \end{aligned}$$

- *Eg. For the lexemes ws: token issued is:*



- **Regular expression for NFA:**

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

Psuedo code for NFA's

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Lexer Approach

The Lexer uses an implementation to read Python code through individual character analysis which distinguishes token types such as:

Keywords and Identifiers: Valid identifiers (e.g., _abc9) and Python reserved keywords are distinguished during tokenization.

Numbers

The lexer identifies numeric values in four formats which include decimal numbers together with binary numbers (0b) and octal numbers (0o) and hexadecimal numbers (0x). It also supports underscores in numbers (e.g., 1_000).

Strings and Operators

The parser correctly analyzes string literals together with standard Python operators which include mathematical expressions and relational operators and quantitative assignments.

Type Annotations and Assignments

The lexer handles type hints (e.g., x: int) and basic arithmetic expressions for assignment tracking (e.g., x = a * b).

Error Handling

The parser detects and retains errors that stem from invalid characters as well as terminated strings and numbers with incorrect formats which are listed with their respective line and column positions to facilitate troubleshooting.

Symbol Table Management

A symbol table inserts identifiers whose attributes maintain name, type and value data. The system applies type inference as an available optio

Syntax Analyzer

Parses the string to check if the rule of grammar were followed, and if followed - creates a parse tree for the input string.

Parsing : Constructing a tree for an input string using the CFG (Context Free Grammar) Rules. There are two types of parsing, Top-down parsing and bottom-up parsing.

First

To Compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or **ε** symbols can be added to any FIRST set.

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i, a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ, then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

Follow

To Compute FOLLOW(A) for all nonterminal A, apply the following rules until nothing can be added to any FOLLOW set.

LL (1) Parser or Predictive Parser.

LL (1) is top-down parser, short for ***Left to Right Scanning***, ***Left most derivation*** with ***one Look ahead***.

To compute $\text{FOLLOW}(A)$ for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in $\text{FOLLOW}(S)$, where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in $\text{FIRST}(\beta)$ except ϵ is in $\text{FOLLOW}(B)$.
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where $\text{FIRST}(\beta)$ contains ϵ, then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

It starts from the root or the Start Variable S and constructs the tree. Before applying make sure that there is no Left Recursion in the Grammar.

It Constructs a parsing table, and the performs a sequence of moves on the input string. E.g.

- **Grammar**

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

- **Grammar After Removing Left-Recursion**

Left recursion is such that *A --> Aα*

Replace the left recursive productions by non-left recursive Productions.

$$A \rightarrow A\alpha \mid \beta$$

$$
\begin{aligned}
A &\rightarrow \beta A' \\
A' &\rightarrow \alpha A' \mid \epsilon
\end{aligned}
$$

For our Grammar:

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

- **Constructing Parsing Table from the Given Grammar**

Then, we construct a Predictive Parsing Table:

Algorithm 4.31: Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Example 4.32: For the expression grammar (4.28), Algorithm 4.31 produces the parsing table in Fig. 4.17. Blanks are error entries; nonblanks indicate a production with which to expand a nonterminal.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

- **Sequence of Moves** for input *id + id * id*

Moves made by the predictive parser on our input:

MATCHED	STACK	INPUT	ACTION
	$E\$$	**id** + **id** * **id**\$	
	$TE'\$$	**id** + **id** * **id**\$	output $E \to TE'$
	$FT'E'\$$	**id** + **id** * **id**\$	output $T \to FT'$
	id $T'E'\$$	**id** + **id** * **id**\$	output $F \to$ **id**
id	$T'E'\$$	+ **id** * **id**\$	match **id**
id	$E'\$$	+ **id** * **id**\$	output $T' \to \epsilon$
id	$+\ TE'\$$	+ **id** * **id**\$	output $E' \to +\ TE'$
id +	$TE'\$$	**id** * **id**\$	match +
id +	$FT'E'\$$	**id** * **id**\$	output $T \to FT'$
id +	**id** $T'E'\$$	**id** * **id**\$	output $F \to$ **id**
id + **id**	$T'E'\$$	* **id**\$	match **id**
id + **id**	$*\ FT'E'\$$	* **id**\$	output $T' \to *\ FT'$
id + **id** *	$FT'E'\$$	**id**\$	match *
id + **id** *	**id** $T'E'\$$	**id**\$	output $F \to$ **id**
id + **id** * **id**	$T'E'\$$	\$	match **id**
id + **id** * **id**	$E'\$$	\$	output $T' \to \epsilon$
id + **id** * **id**	\$	\$	output $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input **id** + **id** * **id**

Outcome : ***($,$)*** : Yes or Accept (No Syntax Error).

Full Project Implementation

Lexical Code Implementation

```
void PythonLexer::advance() {
    if (current() == '\n') {
        line++;
        column = 1;
    } else {
        column++;
    }
    pos++;
}

void PythonLexer::addToken(const std::string& lexeme, TokenType type) {
    tokens.push_back({ lexeme, type, line, column - static_cast<int>(lexeme.length()) });
}

bool PythonLexer::isOperatorChar(char c) {
    static const std::string opChars = "+-*/%=&|<>!^~.";
    return opChars.find(c) != std::string::npos;
}

bool PythonLexer::isDelimiter(char c) {
    static const std::string delimiters = ":,;()[]{}@";
    return delimiters.find(c) != std::string::npos;
}

bool PythonLexer::isHexadecimal(const std::string& str) {
    if (str.size() < 3 || (str[0] != '0' && (str[1] != 'x' && str[1] != 'X'))) {
        return false;
    }
```

advance()

Moves to the next character.

If it's a newline (\n), increase the line number and reset column to 1. Otherwise, just increase the column.

Always move the position forward.

addToken()

Adds a token to the tokens list.

Includes the token's text (lexeme), type, line number, and column.

isOperatorChar(char c)

Checks if the character is a valid operator (+, -, *, etc.).

Returns true if it is, false if not.

isDelimiter(char c)

Checks if the character is a delimiter like :, ;, ,, {, } etc.

Returns true if it is, false if not.

isHexadecimal(const std::string& str)

Checks if a string looks like a hexadecimal number (e.g., 0x1A3F).

Returns false if it's too short or doesn't start with 0x.

Return false if string looks like wrong hexadecimal eg,(0x1g)

```
        addToken(num, TokenType::HexadecimalNumber);
        return;
    } else if (current() == 'b' || current() == 'B') {  // Binary (0b or 0B)
        num += current();
        advance();
        bool hasBinaryDigits = false;
        while (current() == '0' || current() == '1' || current() == '_') {
            char c = current();
            num += c;
            advance();
            if (c == '0' || c == '1') hasBinaryDigits = true;  // Check the consumed character
        }
        if (!hasBinaryDigits) {
            addError("Invalid binary number: " + num + " (no binary digits after 0b)");
            return;
        }

        if (!validateUnderscores()) {
            addError("Invalid underscore placement in binary number: " + num);
            return;
        }

        // Check for invalid trailing characters (e.g., "0b1021")
        if (isalnum(current()) || current() == '_') {
            string invalid = num;
```

Handles binary numbers starting with 0b or 0B.
Checks for valid digits (0, 1, _).
Gives error if no binary digits present or if there are invalid underscores or trailing characters.

```
            string invalid = num;
            while (isalnum(current()) || current() == '_') {
                invalid += current();
                advance();
            }
            addError("Invalid binary number: " + invalid + " (invalid trailing characters)");
            return;
        }

        addToken(num, TokenType::BinaryNumber);
        return;
    } else if (current() == 'o' || current() == 'O') {  // Octal (0o or 0O)
        num += current();
        advance();
        bool hasOctalDigits = false;
        while ((current() >= '0' && current() <= '7') || current() == '_') {
            char c = current();
            num += c;
            advance();
            if (c >= '0' && c <= '7') hasOctalDigits = true;  // Check the consumed character
        }
        if (!hasOctalDigits) {
            addError("Invalid octal number: " + num + " (no octal digits after 0o)");
            return;
        }

        if (!validateUnderscores()) {
            addError("Invalid underscore placement in octal number: " + num);
```

Handles octal numbers starting with 0o or 0O.
Valid digits are from 0 to 7 or _.
Checks for errors like missing digits, bad underscores, or invalid trailing characters.
Adds the token if valid.

```
            addError("Invalid underscore placement in octal number: " + num);
            return;
        }

        // Check for invalid digits (e.g., "0o89")
        if (isdigit(current()) && (current() == '8' || current() == '9')) {
            string invalid = num;
            while (isdigit(current())) {
                invalid += current();
                advance();
            }
            addError("Invalid octal number: " + invalid + " (contains digits 8 or 9)");
            return;
        }

        // Check for invalid trailing characters (e.g., "0o7g")
        if (isalnum(current()) || current() == '_') {
            string invalid = num;
            while (isalnum(current()) || current() == '_') {
                invalid += current();
                advance();
            }
            addError("Invalid octal number: " + invalid + " (invalid trailing characters)");
            return;
        }

        addToken(num, TokenType::OCTALNUMBER);
        return;
```

Checks for invalid digits like 8 or 9 in octal.
Gives error if any invalid trailing characters follow.
Finalizes the octal number token if everything is good.

```
// Check for hexadecimal (0x or 0X), binary (0b or 0B), or octal (0o or 0O)
if (current() == '0' && pos + 1 < source.size()) {
    num += current();
    advance();  // Consume the '0'

    if (current() == 'x' || current() == 'X') {  // Hexadecimal (0x or 0X)
        num += current();
        advance();
        bool hasHexDigits = false;
        while (isxdigit(current()) || current() == '_') {
            char c = current();
            num += c;
            advance();
            if (isxdigit(c)) hasHexDigits = true;  // Check the consumed character
        }
        if (!hasHexDigits) {
            addError("Invalid hexadecimal number: " + num + " (no hexadecimal digits after 0x)");
            // Consume any trailing alphanumeric or underscore characters as part of the invalid token
            while (isalnum(current()) || current() == '_') {
                num += current();
                advance();
            }
            return;
        }
```

Hexadecimal number starts
Starts parsing hexadecimal numbers with 0x or 0X.
Gathers digits and _ characters.
Validates that at least one hex digit exists.

```
if (!validateUnderscores()) {
    addError("Invalid underscore placement in hexadecimal number: " + num);
    return;
}

// Check for invalid trailing characters (e.g., "0x12G")
if (isalnum(current()) || current() == '_') {
    string invalid = num;
    while (isalnum(current()) || current() == '_') {
        invalid += current();
        advance();
    }
```

Hexadecimal Validation

Checks for invalid underscore placement.
Check if there are alphanumeric or _ trailing characters.
Adds an error if any invalid part exists.

```
// Handle decimal point for floating-point numbers
bool hasDecimal = false;
if (current() == '.') {
    hasDecimal = true;
    num += '.';
    advance();
    bool hasFractionalDigits = false;
    while (isdigit(current()) || current() == '_') {
        num += current();
        advance();
        if (isdigit(current())) hasFractionalDigits = true;
    }
    // Check for additional decimal points (e.g., 1.2.2.2)
    while (current() == '.') {
        num += current();
        advance();
        // Consume any digits after the additional decimal point
        while (isdigit(current()) || current() == '_') {
            num += current();
            advance();
        }
        hasDecimal = true; // Mark that we've seen another decimal point
    }
    if (hasDecimal && num.find('.') != num.rfind('.')) { // If there are multiple decimal points
        addError("Invalid floating-point number: " + num + " (multiple decimal points)");
        return;
    }
```

Floating-point Decimal Handling:

Parses numbers with a decimal point .
Check for digits after the dot.
Ensures there's only one decimal point.
Adds error if multiple dots are found.

```
// Check for scientific notation (e.g., 1e-10, 2.5E+3)
bool hasExponent = false;
if (current() == 'e' || current() == 'E') {
    hasExponent = true;
    num += current();
    advance();
    // Check for sign
    if (current() == '+' || current() == '-') {
        num += current();
        advance();
    }
    // Ensure there are digits after the 'e' or 'E'
    bool hasExponentDigits = false;
    while (isdigit(current()) || current() == '_') {
        num += current();
        advance();
        if (isdigit(current())) hasExponentDigits = true;
    }
    if (!hasExponentDigits) {
        addError("Invalid scientific notation: " + num + " (missing exponent digits)");
        return;
    }
```

Exponential Validation:

.Supports scientific notation like 1.2e3.
.Handles signs (+, -) after e/E.
.Requires digits after e/E.
.Errors if exponent digits are missing.

```
// Validate underscore placement for decimal/floating-point/scientific notation
if (num.find("_") != string::npos) {
    if (!validateUnderscores()) {
        addError("Invalid underscore placement in number: " + num);
        return;
    }
    // Additional check: underscores cannot be adjacent to decimal point or 'e'/'E'
    if (hasDecimal && (num.find("._") != string::npos || num.find("_.") != string::npos)) {
        addError("Invalid underscore placement in number: " + num + " (underscore adjacent to decimal point)");
        return;
    }
    if (hasExponent) {
        size_t ePos = num.find('e') != string::npos ? num.find('e') : num.find('E');
        if (ePos > 0 && num[ePos - 1] == '_') {
            addError("Invalid underscore placement in number: " + num + " (underscore before 'e'/'E')");
            return;
        }
        if (ePos + 1 < num.size() && num[ePos + 1] == '_') {
            addError("Invalid underscore placement in number: " + num + " (underscore after 'e'/'E')");
            return;
        }
        // Check after the sign in the exponent (e.g., "1e-_10")
        if (ePos + 2 < num.size() && (num[ePos + 1] == '+' || num[ePos + 1] == '-') && num[ePos + 2] == '_') {
            addError("Invalid underscore placement in number: " + num + " (underscore after exponent sign)");
            return;
        }
    }
}
```

Underscore Validation:

.Ensures underscore is placed correctly.
.Disallows underscores next to . or e/E.
.Checks for cases like 1._2 or 1e_10 and throws error if found.

```
    // Check for complex number (ends with 'j' or 'J')
    if (current() == 'j' || current() == 'J') {
        num += current();
        advance();
        // Always treat complex numbers as invalid
        addError("Invalid token: " + num + " (complex numbers are not supported)");

        // Optionally consume any trailing alphanumeric or underscore characters
        while (isalnum(current()) || current() == '_') {
            num += current();
            advance();
        }
        return;
    }
    // Check for invalid trailing characters (e.g., "123abc")
    if (isalpha(current()) || current() == '_') {
        string invalid = num;
        while (isalnum(current()) || current() == '_') {
            invalid += current();
            advance();
        }
        addError("Invalid number: " + invalid + " (invalid trailing characters)");
        return;
    }
    if (hasExponent || hasDecimal) {
        addToken(num, TokenType::NUMBER);  // Floating-point or scientific notation
    } else if (num.size() > 1 && num[0] == '0' && (num[1] == 'o' || num[1] == 'O')) {
```

Complex validation:

.Detects complex numbers ending with j or J.

.Always treats them as invalid.

. Also checks and consumes trailing characters after j.

```
// Check for complex number (ends with 'j' or 'J')
if (current() == 'j' || current() == 'J') {
    num += current();
    advance();
    // Always treat complex numbers as invalid
    addError("Invalid token: " + num + " (complex numbers are not supported)");

    // Optionally consume any trailing alphanumeric or underscore characters
    while (isalnum(current()) || current() == '_') {
        num += current();
        advance();
    }
    return;
}
// Check for invalid trailing characters (e.g., "123abc")
if (isalpha(current()) || current() == '_') {
    string invalid = num;
    while (isalnum(current()) || current() == '_') {
        invalid += current();
        advance();
    }
    addError("Invalid number: " + invalid + " (invalid trailing characters)");
    return;
}
if (hasExponent || hasDecimal) {
    addToken(num, TokenType::NUMBER);  // Floating-point or scientific notation
} else if (num.size() > 1 && num[0] == '0' && (num[1] == 'o' || num[1] == 'O')) {
    addToken(num, TokenType::OCTALNUMBER);  // Single '0o' can be treated as octal
```

.Final check for invalid trailing characters (e.g., letters after numbers).

.If decimal or exponent exists, token is a floating-point number.

.If starts with 0o or 0O, treated as octal.

```cpp
void PythonLexer::processString(char quote) {
    std::string str;
    int startLine = line;
    int startColumn = column;
    advance();

    bool isTriple = false;
    if (current() == quote && peek() == quote) {
        isTriple = true;
        advance();
        advance();
    }

    if (current() == '\\') {
        advance();
        if (current() != 'n' && current() != 't' && current() != '\\' && current() != '"' && current() != '\'') {
            addError("Invalid escape sequence: \\" + std::string(1, current()));
        }
    }
```

String Start & Escape Check:

.Starts string processing.
.Stores starting line and column.
.Checks if it's a triple-quoted string.
.If it sees a backslash \, it checks for valid escape characters (like \n, \t).
.Invalid escape? → Adds an error.

```cpp
if (isTriple) {
    while (true) {
        if (current() == '\0') {
            addError("Unterminated triple-quoted string starting at line " +
                     std::to_string(startLine) + " column " + std::to_string(startColumn));
            return;
        }

        if (current() == quote && peek() == quote && (pos + 2 < source.size() && source[pos + 2] == quote)) {
            advance();
            advance();
            advance();
            break;
        }
        str += current();
        advance();
    }
    addToken(str, TokenType::STRING);
} else {
    while (current() != quote && current() != '\0') {
        if (current() == '\n') {
            addError("Unterminated string literal starting at line " +
                     std::to_string(startLine) + " column " + std::to_string(startColumn));
            while (current() != '\n' && current() != '\0') {
                advance();
            }
            return;
```

Triple vs Normal String:

If it's a triple-quoted string:
Loops until it finds closing triple quotes.
If it hits the end of input, it reports an error.
If it's a normal string:
Loops until it finds the closing quote.
If it sees a newline, reports an error.

```
            }
            if (current() == '\\') {
                advance();
                if (current() == '\0') break;
            }
            str += current();
            advance();
        }

        if (current() != quote) {
            addError("Unterminated string literal starting at line " +
                     std::to_string(startLine) + " column " + std::to_string(startColumn));
            return;
        }
        advance();
        addToken(str, TokenType::STRING);
    }
}
```

String Finishing:

.Handles escape sequences like \\ inside the string.

.Adds each character to the string.

.If closing quote is missing at the end, shows an "unterminated string" error.

.If all good, adds the string token.

```
void PythonLexer::processIdentifier() {
    std::string ident;

    // This check is redundant since tokenize() now handles invalid prefixes,
    // but we'll keep it as a safety net
    if (std::isdigit(current())) {
        std::string invalid;
        while (std::isalnum(current()) || current() == '_') {
            invalid += current();
            advance();
        }
        addError("Invalid identifier starts with digit: " + invalid);
        return;
    }

    if (current() == '_') {
        ident += current();
        advance();
        if (std::isdigit(current())) {
            std::string invalid = ident;
            invalid += current();
            advance();
            while (std::isalnum(current()) || current() == '_') {
                invalid += current();
                advance();
            }
            addError("Invalid identifier starts with underscore followed by digit: " + invalid);
            return;
        }
```

```
while (std::isalnum(current()) || current() == '_') {
    ident += current();
    advance();
}

std::string lowerIdent = toLower(ident);
```

Invalid Identifier Checks and Collect Valid Identifier:

.If it starts with a digit, it's invalid → error.

.If it starts with underscore + digit, also invalid → error.

.Both cases collect and display the invalid text.

.While the current character is a letter, digit, or underscore, it's part of the identifier.

.Converts the collected identifier to lowercase for comparison.

```cpp
// Check if the identifier is a keyword
std::string keywordMatch;
for (const auto& keyword : keywords) {
    std::string lowerKeyword = toLower(keyword);
    if (lowerIdent == lowerKeyword) {
        keywordMatch = keyword;
        break;
    }
}

// Check if the identifier is a built-in function
bool isBuiltinFunction = false;
std::string builtinMatch;
for (const auto& builtin : builtinFunctions) {
    std::string lowerBuiltin = toLower(builtin);
    if (lowerIdent == lowerBuiltin) {
        isBuiltinFunction = true;
        builtinMatch = builtin;
        break;
    }
}

if (!keywordMatch.empty()) {
    addToken(ident, TokenType::KEYWORD);
} else if (isBuiltinFunction) {
    addToken(ident, TokenType::KEYWORD);
```

Check for Keywords or Built-ins:

.Compares the lowercase identifier to known keywords.
.Also checks if it's a built-in function.
.If matches keyword → mark as KEYWORD.
.If matches built-in → also mark as KEYWORD.

```
        // Add to symbol table if not already added
        if (addedBuiltins.find(ident) == addedBuiltins.end()) {
            int id = symbolTable.addIdentifier(ident, line);
            symbolTable.setIdentifierInfo(ident, "function", "built-in");
            addedBuiltins.insert(ident); // Mark as added
        }
    } else {
        size_t tempPos = pos;
        int tempColumn = column;
        while (std::isspace(current()) && current() != '\n') {
            advance();
        }
        isFunctionCall = (current() == '(');
        pos = tempPos;
        column = tempColumn;

        if (!isFunctionCall) {
            symbolTable.addIdentifier(ident, line);
        }
        addToken(ident, TokenType::IDENTIFIER);
    }
}
```

Add to Symbol Table or Mark as Function:

.If it's a new built-in, add it to the symbol table with label "built-in".

.Otherwise, check if it's a function call by looking ahead for (.

.If not a function, add it as a regular identifier.

```
509     void PythonLexer::processComment() {
510         std::string com;
511         if (current() == '#') {
512             advance();
513 
514             while (current() != '\n' && current() != '\0') {
515                 com += current();
516                 advance();
517             }
518             addToken(com, TokenType::COMMENT);
519         }
520 
521         if (current() == '"' || current() == '\'') {
522             char quote = current();
523             bool isTriple = (peek() == quote && pos + 1 < source.size() && source[pos + 1] == quote);
524 
525             if (isTriple) {
526                 advance();
527                 advance();
528                 advance();
529 
530                 std::string docstring;
531                 while (true) {
532                     if (current() == '\0') {
533                         addError("Unterminated multi-line comment (docstring)");
534                         return;
535                     }
```

```
                    if (current() == quote && peek() == quote && (pos + 2 < source.size() && source[pos + 2] == quote)) {
                        advance();
                        advance();
                        advance();
                        break;
                    }
                    docstring += current();
                    advance();
                }
                addToken(docstring, TokenType::COMMENT);
            }
        }
    }
```

Line & Docstring Comment Start and End Handling :

.If it starts with #, it's a single-line comment.

.Reads until end of line and saves it as a COMMENT token.

.If it starts with ' or ", it may be a triple-quoted docstring.

.Checks if it's triple-quoted, then starts reading the content.

.If it reaches end of file without closing, adds an error.

End Handling

.Continues reading until it finds the closing triple quotes.

.If found, ends the loop.

.Adds the collected docstring as a COMMENT token.

```cpp
void PythonLexer::processOperator() {
    std::string op(1, current());
    const char next = peek();

    // Check for := specifically and flag it as invalid for assignments
    if (current() == ':' && next == '=') {
        op = ":=";
        addError("Invalid assignment operator: " + op + " (only '=' is allowed for variable assignments)");
        advance();
        advance();
        return;
    }

    if (current() == '=') {
        if (next == '=') {
            op = "==";
            addToken(op, TokenType::COMPAREOPERATOR);
            advance();
            advance();
            return;
        } else {
            addToken("=", TokenType::EQUALOPERATOR);
            advance();
            return;
        }
    }
```

Start of processOperator():

.Starts operator processing.
.Handles invalid := combo (not allowed for assignments).
.Handles = and ==:
== → Comparison
= → Assignment

```
if (current() == '!' && next == '=') {
    op = "!=";
    addToken(op, TokenType::COMPAREOPERATOR);
    advance();
    advance();
    return;
}
if (current() == '<') {
    if (next == '=') {
        op = "<=";
        addToken(op, TokenType::COMPAREOPERATOR);
        advance();
        advance();
        return;
    } else if (next == '<') {
        op = "<<";
        addToken(op, TokenType::OPERATOR);
        advance();
        advance();
        return;
    } else {
        addToken("<", TokenType::COMPAREOPERATOR);
        advance();
        return;
    }
}
```

Handles != and <:

!= → Not equal

<, <=, <<:

< → Compare

<= → Compare

<< → Bit shift

```
if (current() == '>') {
    if (next == '=') {
        op = ">=";
        addToken(op, TokenType::COMPAREOPERATOR);
        advance();
        advance();
        return;
    } else if (next == '>') {
        op = ">>";
        addToken(op, TokenType::OPERATOR);
        advance();
        advance();
        return;
    } else {
        addToken(">", TokenType::COMPAREOPERATOR);
        advance();
        return;
    }
}
```

Handles >, >=, >> :

> → Compare

>= → Compare

>> → Bit shift right

```
if (current() == '+') {
    if (next == '=') {
        op = "+=";
        addError("Invalid assignment operator: " + op + " (only '=' is allowed for variable assignments)");
        advance();
        advance();
        return;
    } else {
        addToken("+", TokenType::ADDOPERATOR);
        advance();
        return;
    }
}
if (current() == '-') {
    if (next == '=') {
        op = "-=";
        addError("Invalid assignment operator: " + op + " (only '=' is allowed for variable assignments)");
        advance();
        advance();
        return;
    } else {
        addToken("-", TokenType::MINUSOPERATOR);
        advance();
        return;
    }
}
```

Handles + and - :

+ → Add

+= → Invalid assignment (error)

- → Subtract

-= → Invalid assignment (error)

```
if (current() == '*') {
    if (next == '=') {
        op = "*=";
        addError("Invalid assignment operator: " + op + " (only '=' is allowed for variable assignments)");
        advance();
        advance();
        return;
    } else if (next == '*') {
        op = "**";
        addToken(op, TokenType::POWEROPERATOR);
        advance();
        advance();
        return;
    } else {
        addToken("*", TokenType::MULTIPLYOPERATOR);
        advance();
        return;
    }
}
```

Handles * and ** :

* → Multiply

** → Power

*= → Invalid assignment (error)

```
if (current() == '/') {
    if (next == '=') {
        op = "/=";
        addError("Invalid assignment operator: " + op + " (only '=' is allowed for variable assignments)");
        advance();
        advance();
        return;
    } else {
        addToken("/", TokenType::DIVIDEOPERATOR);
        advance();
        return;
    }
}
if (current() == '%') {
    if (next == '=') {
        op = "%=";
        addError("Invalid assignment operator: " + op + " (only '=' is allowed for variable assignments)");
        advance();
        advance();
        return;
    } else {
        addToken("%", TokenType::PERCENTAGEOPERATOR);
        advance();
        return;
    }
}
```

Handles / and % :

/ → Divide

/= → Invalid assignment (error)

% → Modulo

%= → Invalid assignment (error)

```
if (current() == '&') {
    addToken("&", TokenType::BITANDOPERATOR);
    advance();
    return;
}
if (current() == '|') {
    addToken("|", TokenType::BITOROPERATOR);
    advance();
    return;
}
if (current() == '^') {
    addToken("^", TokenType::POWEROPERATOR);
    advance();
    return;
}
if (current() == '.') {
    addToken(".", TokenType::OPERATOR);
    advance();
    return;
}

addError("Unexpected operator: " + std::string(1, current()));
advance();
```

Basic Symbols:

Checks for single-character operators: & → Bitwise AND

| → Bitwise OR

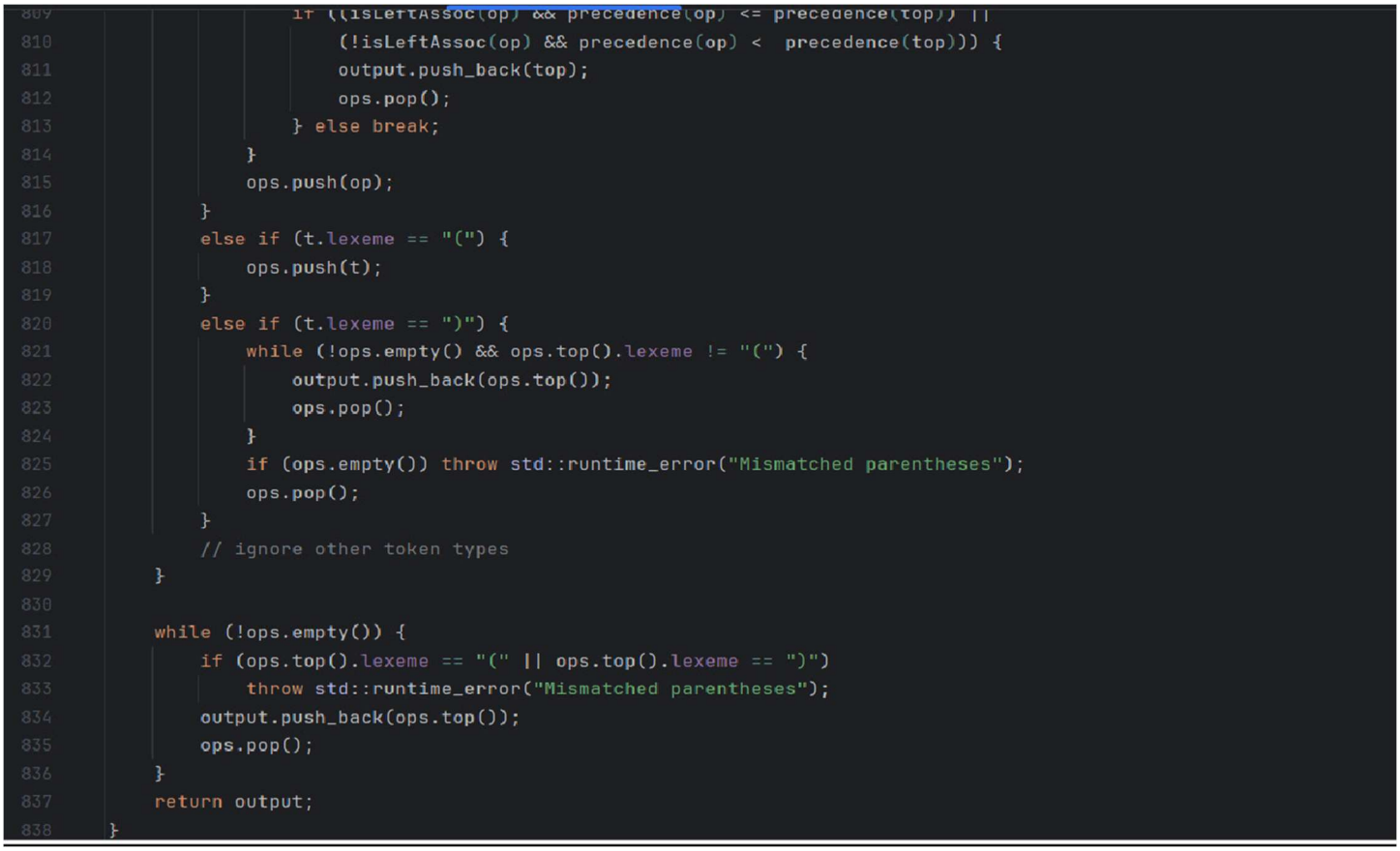
^ → Power or XOR

. → Dot operator

If unknown symbol → adds error.

```cpp
// Convert infix tokens to Reverse Polish Notation (RPN) using the Shunting-Yard algorithm
std::vector<Token> PythonLexer::toRPN(const std::vector<Token>& input) {
    std::vector<Token> output;
    std::stack<Token> ops;

    auto precedence = [&](const Token& t) {
        switch (t.type) {
        case TokenType::POWEROPERATOR:      return 4;
        case TokenType::MULTIPLYOPERATOR:
        case TokenType::DIVIDEOPERATOR:
        case TokenType::PERCENTAGEOPERATOR: return 3;
        case TokenType::ADDOPERATOR:
        case TokenType::MINUSOPERATOR:      return 2;
        default:                            return 0;
        }
    };
    auto isLeftAssoc = [&](const Token& t) {
        return t.type != TokenType::POWEROPERATOR;
    };

    for (size_t i = 0; i < input.size(); ++i) {
        const Token& t = input[i];

        // <<<- updated operand check:
        if (t.type == TokenType::NUMBER        ||
            t.type == TokenType::HexadecimalNumber ||
            t.type == TokenType::BinaryNumber  ||
            t.type == TokenType::OCTALNUMBER   ||
            t.type == TokenType::IDENTIFIER    ||
```

```
        t.type == TokenType::BinaryNumber  ||
        t.type == TokenType::OCTALNUMBER   ||
        t.type == TokenType::IDENTIFIER    ||
        t.type == TokenType::KEYWORD
        )
    {
        output.push_back(t);
    }
    else if (t.type == TokenType::ADDOPERATOR   ||
             t.type == TokenType::MINUSOPERATOR ||
             t.type == TokenType::MULTIPLYOPERATOR ||
             t.type == TokenType::DIVIDEOPERATOR   ||
             t.type == TokenType::PERCENTAGEOPERATOR||
             t.type == TokenType::POWEROPERATOR) {
        // (same unary minus + precedence logic as before)
        Token op = t;
        bool unary = (t.type == TokenType::MINUSOPERATOR) &&
                     (i == 0 || input[i-1].lexeme == "(" ||
                      input[i-1].type == TokenType::ADDOPERATOR ||
                      input[i-1].type == TokenType::MINUSOPERATOR ||
                      input[i-1].type == TokenType::MULTIPLYOPERATOR ||
                      input[i-1].type == TokenType::DIVIDEOPERATOR ||
                      input[i-1].type == TokenType::PERCENTAGEOPERATOR ||
                      input[i-1].type == TokenType::POWEROPERATOR);
        if (unary) op.type = TokenType::MINUSOPERATOR;
        while (!ops.empty()) {
            const Token& top = ops.top();
```



```
            if ((isLeftAssoc(op) && precedence(op) <= precedence(top)) ||
                (!isLeftAssoc(op) && precedence(op) <  precedence(top))) {
                output.push_back(top);
                ops.pop();
            } else break;
        }
        ops.push(op);
    }
    else if (t.lexeme == "(") {
        ops.push(t);
    }
    else if (t.lexeme == ")") {
        while (!ops.empty() && ops.top().lexeme != "(") {
            output.push_back(ops.top());
            ops.pop();
        }
        if (ops.empty()) throw std::runtime_error("Mismatched parentheses");
        ops.pop();
    }
    // ignore other token types
}

while (!ops.empty()) {
    if (ops.top().lexeme == "(" || ops.top().lexeme == ")")
        throw std::runtime_error("Mismatched parentheses");
    output.push_back(ops.top());
    ops.pop();
}
return output;
}
```

Creates output list and ops stack:

Defines:

precedence() — gives priority to operators. isLeftAssoc() — checks if operator is left-associative.
Starts looping through tokens.
If the token is a number, identifier, or keyword → it's an operand → send to output. If token is an operator:
Handles unary minus (e.g. -a).
While stack has higher/equal precedence operators → pop them to output. Then push current operator to the stack.
If token is '(' → push to stack.
If token is ')' → pop from stack to output until '(' is found. If no '(' found → throw error for mismatched parentheses. After loop ends → pop all remaining operators.
If any leftover parentheses → also throw error.

```cpp
// Evaluate an RPN token sequence; looks up identifiers in symbolTable
double PythonLexer::evalRPN(const std::vector<Token>& rpn) {
    std::stack<double> st;
    for (const Token& t : rpn) {
        if (t.type == TokenType::NUMBER) {
            st.push(std::stod(t.lexeme));
        }
        else if (t.type == TokenType::HexadecimalNumber) {
            st.push(static_cast<double>(std::stoll(t.lexeme, nullptr, 16)));
        }
        else if (t.type == TokenType::BinaryNumber) {
            st.push(static_cast<double>(std::stoll(t.lexeme.substr(2), nullptr, 2)));
        }
        else if (t.type == TokenType::OCTALNUMBER) {
            st.push(static_cast<double>(std::stoll(t.lexeme.substr(2), nullptr, 8)));
        }
        else if (t.type == TokenType::IDENTIFIER) {
            if (!symbolTable.getSymbols().count(t.lexeme))
                throw std::runtime_error("Undefined identifier: " + t.lexeme);
            st.push(std::stod(symbolTable.getValue(t.lexeme)));
        }
        else if (t.type == TokenType::KEYWORD) {
            std::string val = toLower(t.lexeme);
            if (val == "true") st.push(1.0);
            else if (val == "false") st.push(0.0);
            else throw std::runtime_error("Unexpected keyword in expression: " + t.lexeme);
        }
        else {
            if (t.type == TokenType::MINUSOPERATOR && st.size() == 1) {
                double a = st.top(); st.pop();
```

```
                double a = st.top(); st.pop();
                st.push(-a);
            } else {
                if (st.size() < 2) throw std::runtime_error("Invalid expression");
                double b = st.top(); st.pop();
                double a = st.top(); st.pop();
                double res;
                switch (t.type) {
                case TokenType::ADDOPERATOR:        res = a + b; break;
                case TokenType::MINUSOPERATOR:      res = a - b; break;
                case TokenType::MULTIPLYOPERATOR:   res = a * b; break;
                case TokenType::DIVIDEOPERATOR:
                    if (b == 0) throw std::runtime_error("Division by zero");
                    res = a / b; break;
                case TokenType::PERCENTAGEOPERATOR:
                    if (b == 0) throw std::runtime_error("Modulo by zero");
                    res = std::fmod(a, b); break;
                case TokenType::POWEROPERATOR:      res = std::pow(a, b); break;
                default:  throw std::runtime_error("Unknown operator");
                }
                st.push(res);
            }
        }
    }
    if (st.size() != 1) return 0.0;
    return st.top();
}
```

In this part of the code:

It Goes through each token in the input list. If the token is a number:
Converts it to a double and pushes it onto the stack. Supports all number formats: decimal, hex, binary, octal. If it's an identifier:
Looks it up in the symbolTable. If not found → error.
If it's a keyword:
"true" → 1.0
"false" → 0.0 Anything else → error.
If the token is a unary minus (like -5), it negates the top value. Else, it's a binary operator:
Pops two numbers.
Applies the operation (+, -, *, /, %, **). Pushes the result.
Handles division/modulo by zero. At the end:
If stack size is not exactly 1 → error. Otherwise, return the final result.

```
// Check that LHS is a valid identifier name only
if (!std::isalpha(lhs[0]) && lhs[0] != '_') {
    addError("Invalid identifier on left-hand side of assignment: '" + lhs + "'");
    i = j;
    continue;
}

// Handle single-token special cases
if (expr.size() == 1) {
    const Token &t = expr[0];
    switch (t.type) {
    case TokenType::STRING:
        symbolTable.setIdentifierInfo(lhs, "string", t.lexeme);
        i = j;
        continue;
    case TokenType::KEYWORD: {
        std::string v = toLower(t.lexeme);
        if (v == "true" || v == "false") {
            symbolTable.setIdentifierInfo(lhs, "bool", v);
            i = j;
            continue;
        }
        break;
    }
    case TokenType::HexadecimalNumber:
    case TokenType::BinaryNumber:
    case TokenType::OCTALNUMBER:
        symbolTable.setIdentifierInfo(lhs, "int", t.lexeme);
        i = j;
        continue;
    case TokenType::NUMBER: {
        bool isFloat = (t.lexeme.find('.') != std::string::npos) ||
                       (t.lexeme.find('e') != std::string::npos) ||
                       (t.lexeme.find('E') != std::string::npos);
        std::string dtype = isFloat ? "float" : "int";
        symbolTable.setIdentifierInfo(lhs, dtype, t.lexeme);
        i = j;
        continue;
    }
    case TokenType::IDENTIFIER: {
        if (symbolTable.getSymbols().count(t.lexeme)) {
            std::string v = symbolTable.getValue(t.lexeme);
            std::string dt = symbolTable.getDataType(t.lexeme);
            symbolTable.setIdentifierInfo(lhs, dt, v);
        } else {
            addError("Undefined identifier in assignment: " + t.lexeme);
        }
        i = j;
        continue;
    }
    default:
        break;
    }
```

```
            // General case: full expression (RPN)
            try {
                auto rpn = toRPN(expr);
                double result = evalRPN(rpn);
                bool isInt = std::floor(result) == result;
                std::string dtype = isInt ? "int" : "float";
                std::string sval = isInt
                                       ? std::to_string(static_cast<long long>(result))
                                       : std::to_string(result);
                symbolTable.setIdentifierInfo(lhs, dtype, sval);
            } catch (const std::exception &e) {
                addError(e.what());
            }
            i = j;
        } else {
            ++i;
        }
    }
}



void PythonLexer::addError(const std::string& message) {
    errors.push_back({ message, line, column });
}
```

Image 1 :
LHS Check & Single-Token Assignments Checks if the left-hand side (LHS) is a valid identifier.
If right-hand side (expr) has only 1 token, handles special cases: String → save as string.
Keyword "true"/"false" → save as boolean.

Image 2:
More Single-Token Assignments Handles number types:
Hex, Binary, Octal → stored as int.
Normal number → checks if it's float or int.
If it's an identifier, looks up its value and type in the symbol table. If found → copy it.
If not → show error.
Image 3: General Case – Expression Evaluation If it's not a single token:
Converts expression to RPN. Evaluates it using evalRPN. Checks if result is an int or float. Saves it in the symbol table.
If any error occurs → catches and logs it.

```cpp
std::pair<std::vector<Token>, std::vector<LexicalError>> PythonLexer::tokenize() {
    while (current() != '\0') {
        char c = current();
        switch (c) {
        // 1) Newline
        case '\n':
            addToken("\n", TokenType::NEWLINE);
            advance();
            break;
            // 2) Whitespace
        case ' ': case '\r': case '\t': case '\v': case '\f':
            advance();
            break;

            // 3) Comment
        case '#':
            processComment();
            break;

            // 4) Number (decimal)
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            processNumber();
            break;
            // 5) String literal
        case '"': case '\'':
            processString(c);
            break;

```

```
            // 6) Valid identifier start
        default:
            if (std::isalpha(c) || c == '_') {
                if (!processTypeAnnotation()) {
                    processIdentifier();
                }
                break;
            }
            // 7) Invalid identifier start like @, $, etc.
            else if (c == '@' || c == '$' || c == '`' || c == '\\' || c == '!') {
                int errLine = line;
                int errColumn = column;
                std::string bad = { c };
                advance();
                while (std::isalnum(current()) || current() == '_') {
                    bad += current();
                    advance();
                }
                addError("Invalid identifier at line " + std::to_string(errLine) +
                         " column " + std::to_string(errColumn) +
                         ": '" + bad + "' (identifiers must start with a letter or underscore)");
                break;
            }

            // 8) Operators
            else if (isOperatorChar(c)) {
                processOperator();
                break;
            }

```

```
            // 9) Delimiters
            else if (isDelimiter(c)) {
                addToken(std::string(1, c), TokenType::DELIMITER);
                advance();
                break;
            }

            // 10) Catch-all: unknown/unexpected characters
            else {
                int errLine = line;
                int errColumn = column;
                std::string bad = { c };
                advance();
                while (std::isalnum(current()) || current() == '_') {
                    bad += current();
                    advance();
                }
                addError("Invalid character sequence at line " + std::to_string(errLine) +
                         " column " + std::to_string(errColumn) +
                         ": '" + bad + "' (unknown or unsupported characters)");
                break;
            }
        }
    }

    addToken("", TokenType::ENDOFFILE);
    processAssignments();
    return { tokens, errors };
}
```

Token Type Detection:

Newline → Adds a NEWLINE token. Whitespace → Skips it.

Comments (#) → Calls processComment(). Numbers (0–9) → Calls processNumber(). String literals (" or ') → Calls processString(). Token Type Detection (page 2)

Valid identifiers (starts with letter or _) → Calls processIdentifier(). Invalid identifiers (starts with @, $, etc.) → Collects and logs an error. Operators (like +, -, ==, etc.) → Calls processOperator().

Token Type Detection (Page 3)

Delimiters (e.g., ,, :) → Adds a DELIMITER token.

Unknown characters → Logs an error with the invalid sequence.

Header File:

```
#ifndef PYTHONLEXER_H
#define PYTHONLEXER_H
#include <string>
#include <vector>
#include <unordered_set>
#include <unordered_map>
#include <optional>

enum class TokenType {
    KEYWORD, IDENTIFIER, HexadecimalNumber, BinaryNumber, OCTALNUMBER, NUMBER, COMPLEX_NUMBER, STRING, OPERATOR,
    ADDOPERATOR, MINUSOPERATOR, MULTIPLYOPERATOR, DELIMITER, EQUALOPERATOR, BITOROPERATOR, BITANDOPERATOR,
    PERCENTAGEOPERATOR, COMPAREOPERATOR, DIVIDEOPERATOR, POWEROPERATOR, INDENT, DEDENT, NEWLINE, COMMENT, ENDOFFILE
};
struct Token {
    std::string lexeme;
    TokenType type;
    int line;
    int column;
};
struct LexicalError {
    std::string message;
    int line;
    int column;
};
class SymbolTable {
private:
        std::string dataType;
        std::string value;
    };

    std::unordered_map<std::string, SymbolEntry> symbols;
    int current_id = 1;0
public:
    int addIdentifier(const std::string& identifier, int line) {
        if (!symbols.count(identifier)) {
            symbols[identifier] = { current_id++, "unknown", "N/A" };
        }
        return symbols[identifier].id;
    }

    void setIdentifierInfo(const std::string& identifier, const std::string& dataType, const std::string& value) {
        if (symbols.count(identifier)) {
            symbols[identifier].dataType = dataType;
            symbols[identifier].value = value;
        }
    }

    std::optional<int> lookup(const std::string& identifier) const {
        auto it = symbols.find(identifier);
        return it != symbols.end() ? std::optional<int>(it->second.id) : std::nullopt;
    }

    const std::unordered_map<std::string, SymbolEntry>& getSymbols() const {
        return symbols;
    }
```

```
    std::unordered_set<std::string> addedBuiltins; // Track added built-in functions
    std::vector<int> indentStack;  // Stack of indentation levels
    int currentIndent = 0;         // Current indentation level
    bool atStartOfLine = true;     // Flag for start of line

    const std::unordered_set<std::string> keywords = {
        "False", "None", "True", "and", "as", "assert", "async", "await",
        "break", "class", "continue", "def", "del", "elif", "else", "except",
        "finally", "for", "from", "global", "if", "import", "in", "is",
        "lambda", "nonlocal", "not", "or", "pass", "raise", "return",
        "try", "while", "with", "yield"
    };

    const std::unordered_set<std::string> builtinFunctions = {
        "print"
    };

    const std::unordered_set<std::string> typeHints = {
        "int", "float", "str", "bool", "complex"
    };

    char current() const { return pos < source.size() ? source[pos] : '\0'; }
    char peek() const { return pos + 1 < source.size() ? source[pos + 1] : '\0'; }

    void advance();
    void addToken(const std::string& lexeme, TokenType type);
    void addError(const std::string& message);
    bool isOperatorChar(char c);
```

```

    int getId(const std::string& identifier) const {
        auto it = symbols.find(identifier);
        return it != symbols.end() ? it->second.id : -1;
    }

    std::string getDataType(const std::string& identifier) const {
        auto it = symbols.find(identifier);
        return it != symbols.end() ? it->second.dataType : "unknown";
    }

    std::string getValue(const std::string& identifier) const {
        auto it = symbols.find(identifier);
        return it != symbols.end() ? it->second.value : "N/A";
    }
};

class PythonLexer {
private:
    std::string source;
    size_t pos = 0;
    int line = 1;
    int column = 1;
    SymbolTable symbolTable;
    std::vector<Token> tokens;
    std::vector<LexicalError> errors;
    std::unordered_map<std::string, std::string> typeAnnotations;
    bool isFunctionCall = false;
    std::unordered_set<std::string> addedBuiltins; // Track added built-in functions
```

```cpp
std::string tokenTypeToString(TokenType type) {
    switch (type) {
    case TokenType::KEYWORD:            return "KEYWORD";
    case TokenType::IDENTIFIER:         return "IDENTIFIER";
    case TokenType::HexadecimalNumber:  return "HEXADDECIMAL_NUMBER";
    case TokenType::BinaryNumber:       return "BINARY_NUMBER";
    case TokenType::OCTALNUMBER:        return "OCTAL_NUMBER";
    case TokenType::NUMBER:             return "NUMBER";
    case TokenType::COMPLEX_NUMBER:     return "COMPLEX_NUMBER";
    case TokenType::STRING:             return "STRING";
    case TokenType::OPERATOR:           return "OPERATOR";
    case TokenType::ADDOPERATOR:        return "ADD_OPERATOR";
    case TokenType::MINUSOPERATOR:      return "MINUS_OPERATOR";
    case TokenType::MULTIPLYOPERATOR:   return "MULTIPLY_OPERATOR";
    case TokenType::DELIMITER:          return "DELIMITER";
    case TokenType::EQUALOPERATOR:      return "EQUAL_OPERATOR";
    case TokenType::BITOROPERATOR:      return "BITWISE_OR_OPERATOR";
    case TokenType::BITANDOPERATOR:     return "BITWISE_AND_OPERATOR";
    case TokenType::PERCENTAGEOPERATOR: return "PERCENTAGE_OPERATOR";
    case TokenType::COMPAREOPERATOR:    return "COMPARE_OPERATOR";
    case TokenType::DIVIDEOPERATOR:     return "DIVIDE_OPERATOR";
    case TokenType::POWEROPERATOR:      return "POWER_OPERATOR";
    case TokenType::INDENT:             return "INDENT";
    case TokenType::DEDENT:             return "DEDENT";
    case TokenType::NEWLINE:            return "NEWLINE";
    case TokenType::COMMENT:            return "COMMENT";
    bool isOperatorChar(char c);
    bool isDelimiter(char c);
    bool isHexadecimal(const std::string& str);
    bool isTab(char c);
    void processNumber();
    void processString(char quote);
    void processIdentifier();
    void processComment();
    void processOperator();
    double evalRPN(const std::vector<Token>& rpn);
    std::vector<Token> toRPN(const std::vector<Token>& input);
    void processAssignments();
    bool processTypeAnnotation();

public:
    PythonLexer(const std::string& input);
    std::pair<std::vector<Token>, std::vector<LexicalError>> tokenize();
    const SymbolTable& getSymbolTable() const { return symbolTable; }
};

std::string tokenTypeToString(TokenType type);

#endif // PYTHONLEXER_H
```

Syntax Code Implementation

```
    // Debug: Print token stream
    std::cout << "Token stream:" << std::endl;
    for (size_t i = 0; i < tokens.size(); i++) {
        std::cout << "Token " << i << ": type=" << tokenTypeToString(tokens[i].type)
        << ", lexeme='" << tokens[i].lexeme
        << "', line=" << tokens[i].line
        << ", col=" << tokens[i].column << std::endl;
    }
    std::cout << std::endl;

    while (!isAtEnd()) {
        // Skip blank lines
        if (currentToken().type == TokenType::NEWLINE) {
            advance();
            continue;
        }

        size_t startPos = pos;
        ParseNode* stmt = parseStmt();

        // Only push if a valid node was returned
        if (stmt) {
            root->children.push_back(stmt);
        }

        // --- SAFETY: always ensure progress ---
        if (pos == startPos) {
            advance(); // move forward to escape the loop
        }
    }

    return root;
}
```

This part of the code is responsible for reading the list of tokens (words or symbols from the input program) and printing them for debugging. Then, it enters a loop to process the tokens one by one, skipping over any blank lines. For each statement, it tries to create a tree node (a ParseNode). If a valid node is created, it adds it to the root of the parse tree. There is a safety check to make sure the loop always moves forward, even if something goes wrong, so the parser doesn't get stuck.

```cpp
void SyntaxAnalyzer::populateTree(QTreeWidget* tree) {
    tree->clear();
    ParseNode* root = parseProgram();
    // Display parse tree
    QTreeWidgetItem* rootItem = new QTreeWidgetItem(tree);
    rootItem->setText(0, root->name);
    buildTree(rootItem, root);
    tree->addTopLevelItem(rootItem);
    tree->expandAll();
}

ParseNode* SyntaxAnalyzer::parseComparison() {
    // Start by parsing a simple arithmetic expression
    auto left = parseExpression();
    if (!left) return nullptr;

    // Loop to handle comparison operators chained: ==, !=, <, <=, >, >=
    while (true) {
        std::string op;
        if (match("==")) op = "==";
        else if (match("!=")) op = "!=";
        else if (match("<=")) op = "<=";
        else if (match(">=")) op = ">=";
        else if (match("<"))  op = "<";
        else if (match(">"))  op = ">";
        else break;

        auto right = parseExpression();
        if (!right) return nullptr;

        // Build a comparison operator node
        auto cmpNode = new ParseNode("CompareOp", QString::fromStdString(op));
        cmpNode->children.push_back(left);
        cmpNode->children.push_back(right);

        left = cmpNode; // allow chaining comparisons
    }

    return left;
}
```

Here, the parser has a function to display the parse tree using a graphical tree widget (for example, in a Qt GUI application). It first parses the whole program then builds and shows the tree structure. Below that, the code explains how the parser recognizes comparison operations (like ==, !=, <, >, etc.). It builds nodes for each comparison, allowing for chaining (like a < b < c), and constructs a tree that represents these comparisons.

```cpp
ParseNode* SyntaxAnalyzer::parseReturnStmt() {
    auto node = new ParseNode("ReturnStmt");
    // we know 'return' was just matched
    if (currentToken().type != TokenType::NEWLINE && currentToken().lexeme != ":") {
        auto expr = parseExpression();
        if (expr) {
            node->children.push_back(expr);
        } else {
            addSyntaxError("Invalid expression in return",
                           currentToken().line, currentToken().column);
        }
    }
    return node;
}

//—— Pass statement ——
// 'pass'
ParseNode* SyntaxAnalyzer::parsePassStmt() {
    auto node = new ParseNode("PassStmt");
    return node;
}

//—— Break statement ——
// 'break'
ParseNode* SyntaxAnalyzer::parseBreakStmt() {
    auto node = new ParseNode("BreakStmt");
    return node;
}

//—— Continue statement ——
// 'break'
ParseNode* SyntaxAnalyzer::parseContinueStmt() {
    auto node = new ParseNode("ContinueStmt");
    return node;
}

//—— Statement dispatch ——

ParseNode* SyntaxAnalyzer::parseStmt() {
    std::cout << "parseStmt: current token type=" << tokenTypeToString(currentToken().type)
    << ", lexeme='" << currentToken().lexeme
    << "', line=" << currentToken().line
    << ", col=" << currentToken().column << std::endl;

    // Handle DEDENT - it's not a statement, just return nullptr to end the block
    if (currentToken().type == TokenType::DEDENT) {
        std::cout << "  Found DEDENT - ending block" << std::endl;
        advance(); // consume the DEDENT
        return nullptr;
    }
```

For a return statement, it checks if there is an expression to return; if not, it reports an error.

For simple statements like pass, break, and continue, it creates a special node in the tree for each.

```
while (!isAtEnd() && currentToken().type == TokenType::NEWLINE) {
    advance();
    std::cout << "  Skipped newline" << std::endl;
}

// Handle whitespace/indentation without consuming statement tokens
while (!isAtEnd() && (currentToken().type == TokenType::WHITESPACE ||
                      currentToken().type == TokenType::INDENT ||
                      currentToken().type == TokenType::DEDENT)) {
    std::cout << "  Skipped " << tokenTypeToString(currentToken().type) << std::endl;
    advance();
}

// 2) Skip stray colons
while (!isAtEnd() && currentToken().lexeme == ":") {
    std::cout << "  Skipped colon" << std::endl;
    advance();
}

// 3) Statement heads
if (match("if")) {
    std::cout << "  Parsing if statement" << std::endl;
    auto core = parseIfCore();
    return parseIfChain(core);
}
if (match("for"))    return parseForStmt();
if (match("while")) return parseWhileStmt();
if (match("def"))    return parseFuncDef();
if (match("return")) return parseReturnStmt();
if (match("pass"))    return parsePassStmt();
if (match("else")) {
    addSyntaxError("'else' without matching 'if'",
                   currentToken().line, currentToken().column);
    return nullptr;
}
if (match("elif")) {
    addSyntaxError("'elif' without matching 'if'",
                   currentToken().line, currentToken().column);
    return nullptr;
}
if (match("break"))  return parseBreakStmt();
if (match("continue")) return parseContinueStmt();

// 4) Built-in functions like print
if (currentToken().type == TokenType::IDENTIFIER) {
    std::string funcName = currentToken().lexeme;
    std::cout << "  Found identifier: " << funcName << std::endl;
```

This section goes deeper into how the parser skips over unnecessary tokens, like blank lines, spaces, or stray colons. Then it looks at the main part of a statement (like if, for, while, etc.). For each recognized keyword, it calls the right function to parse that statement. If an elif or else is found without a matching if, it reports an error. If an identifier (like a variable or function name) is found, it prepares to handle function calls or variable use.

```
        // For other built-in functions, require parentheses
        if (!match("(")) {
            addSyntaxError("Expected '(' after '" + funcName + "'",
                           currentToken().line, currentToken().column);
            return nullptr;
        }

        // Parse arguments
        if (!isAtEnd() && currentToken().lexeme != ")") {
            do {
                // Skip any whitespace before argument
                while (!isAtEnd() && currentToken().type == TokenType::WHITESPACE) {
                    advance();
                }

                auto arg = parseExpression();
                if (!arg) {
                    delete node;
                    return nullptr;
                }
                node->children.push_back(arg);

                // Skip any whitespace after argument
                while (!isAtEnd() && currentToken().type == TokenType::WHITESPACE) {
                    advance();
                }
            } while (match(","));
        }

        if (!match(")")) {
            addSyntaxError("Expected ')' after arguments",
                           currentToken().line, currentToken().column);
            delete node;
            return nullptr;
        }

        return node;
    }
```

If the parser finds a function name, it checks that it is followed by parentheses and parses any arguments inside. For each argument, it skips spaces and expects a valid expression. If anything is missing (like a closing parenthesis), it reports an error. This part ensures that function calls are correctly structured in the code and in the parse tree.

```
    // 5) Assignment: x = ..., x += ..., etc.
    if (pos + 1 < tokens.size() &&
        (tokens[pos + 1].lexeme == "=" ||
         tokens[pos + 1].lexeme == "+=" ||
         tokens[pos + 1].lexeme == "-=" ||
         tokens[pos + 1].lexeme == "*=" ||
         tokens[pos + 1].lexeme == "/=")) {
        return parseAssignment();
    }

    // 6) Function call or identifier expression
    auto exprStmt = parseExprStmt();
    if (!exprStmt) {
        addSyntaxError("Invalid expression or unknown statement",
                       currentToken().line, currentToken().column);
        return nullptr;
    }
    return exprStmt;
}

// 7) Fallback: expression statement
auto exprStmt = parseExprStmt();
if (!exprStmt) {
    addSyntaxError("Invalid expression or unknown statement",
                   currentToken().line, currentToken().column);
    return nullptr;
}

return exprStmt;
```

This page checks if the next statement is an assignment (like x = 5 or x += 1). If so, it parses the assignment. If not, it tries to parse the statement as an expression or a function call. If it can't understand the statement, it reports an error. This acts as a fallback to handle any statement that doesn't match the earlier patterns.

```
ParseNode* SyntaxAnalyzer::parseWhileStmt() {
    auto node = new ParseNode("WhileStmt");

    // 1) Optional parentheses
    bool sawParen = match("(");

    // 2) Parse condition
    auto cond = parseComparison();
    if (!cond) {
        addSyntaxError("Invalid expression in while condition",
                       currentToken().line, currentToken().column);
        while (!isAtEnd() && currentToken().lexeme != ":" &&
               currentToken().type != TokenType::NEWLINE)
            advance();
    } else {
        node->children.push_back(cond);
    }

    // 3) Close parenthesis if opened
    if (sawParen && !match(")")) {
        addSyntaxError("Expected ')' after while condition",
                       currentToken().line, currentToken().column);
    }

    // 4) Check for assignment instead of comparison
    if (currentToken().lexeme == "=") {
        addSyntaxError("Invalid '=' in condition; did you mean '=='?",
                       currentToken().line, currentToken().column);
        advance();
        while (!isAtEnd() && currentToken().lexeme != ":" &&
               currentToken().type != TokenType::NEWLINE)
            advance();
    }

    // 5) Expect colon
    if (!match(":")) {
        addSyntaxError("Expected ':' after while condition",
                       currentToken().line, currentToken().column);
        while (!isAtEnd() && currentToken().type != TokenType::NEWLINE) {
            advance();
        }
        return node;
    }

    // 6) Check indentation
    if (!checkIndentation("while")) {
        return node;
    }

    // 7) Parse the body
    auto body = parseStmt();
    if (body) {
        node->children.push_back(body);
    }

    return node;
```

This section is about parsing while loops. It checks for optional parentheses, parses the loop condition (making sure it's valid), and checks for common mistakes like using = instead of ==. It also ensures that a colon is present after the condition, checks for correct indentation, and then parses the statement inside the loop.

```
ParseNode* SyntaxAnalyzer::parseFuncDef() {
    auto node = new ParseNode("FuncDef");

    // 1) Parse function name
    if (currentToken().type != TokenType::IDENTIFIER) {
        addSyntaxError("Expected function name after def",
                       currentToken().line, currentToken().column);
        return nullptr;
    }
    node->children.push_back(
        new ParseNode("Identifier",
                      QString::fromStdString(currentToken().lexeme)));
    advance();

    // 2) Parse parameter list
    if (!match("(")) {
        addSyntaxError("Expected '(' after function name",
                       currentToken().line, currentToken().column);
        delete node;
        return nullptr;
    }

    if (currentToken().type == TokenType::IDENTIFIER) {
        auto params = parseParamList();
        if (params) {
            node->children.push_back(params);
        }
    }

    if (!match(")")) {
        // If the next token is another identifier, it's almost certainly a missing comma
        if (currentToken().type == TokenType::IDENTIFIER) {
            addSyntaxError("Expected ',' between parameters",
                           currentToken().line, currentToken().column);
        } else {
            addSyntaxError("Expected ')' after parameters",
                           currentToken().line, currentToken().column);
        }
        delete node;
        return nullptr;
    }

    // 3) Expect colon
    if (!match(":")) {
        addSyntaxError("Expected ':' after def header",
                       currentToken().line, currentToken().column);
        delete node;
        return nullptr;
    }

    // 4) Check for proper indentation
    if (!checkIndentation("def")) {
        delete node;
        return nullptr;
    }

    // 5) Parse the function body
    auto body = parseStmt();
    if (!body) {
        delete node;
        return nullptr;
    }
    node->children.push_back(body);
    return node;
}
```

This part explains how function definitions (def) are parsed. It makes sure the function name is present and is followed by parentheses. It parses the list of parameters (variables inside the parentheses), checking for missing commas or parentheses and reporting errors if needed. It then expects a colon and the correct indentation before parsing the function's body (the code inside the function).

```
ParseNode* SyntaxAnalyzer::parseParamList() {
    auto node = new ParseNode("ParamList");

    bool expectComma = false;

    while (!isAtEnd()) {
        if (currentToken().type != TokenType::IDENTIFIER) {
            // If expecting comma but got an identifier, comma is missing
            if (expectComma) {
                addSyntaxError("Expected ',' between parameters",
                               currentToken().line, currentToken().column);
                return nullptr;
            } else {
                break; // maybe it's closing ')'
            }
        }

        // Add param
        node->children.push_back(
            new ParseNode("Param", QString::fromStdString(currentToken().lexeme)));
        advance();

        // After a param, expect either ',' or ')'
        if (match(",")) {
            expectComma = false;
        } else {
            expectComma = true;
            break;
        }
    }

    return node;
}
```

Here, the parser focuses on parsing the parameter list for a function. It checks that each parameter is an identifier and that commas separate them. If something is missing, like a comma between parameters, it reports an error. It continues until all parameters are read or the closing parenthesis is found.

```cpp
ParseNode* SyntaxAnalyzer::parseAssignment() {
    auto node = new ParseNode("Assignment");

    // Get the target identifier
    if (currentToken().type != TokenType::IDENTIFIER) {
        addSyntaxError("Expected identifier before assignment operator",
                       currentToken().line, currentToken().column);
        return nullptr;
    }

    node->children.push_back(
        new ParseNode("Identifier",
                      QString::fromStdString(currentToken().lexeme)));
    advance();

    // Handle both simple and compound assignments
    std::string op;
    if (match("=")) {
        op = "=";
    } else if (match("+=")) {
        op = "+=";
    } else if (match("-=")) {
        op = "-=";
    } else if (match("*=")) {
        op = "*=";
    } else if (match("/=")) {
        op = "/=";
    } else {
        addSyntaxError("Expected assignment operator",
                       currentToken().line, currentToken().column);
        delete node;
        return nullptr;
    }

    node->value = QString::fromStdString(op); // Store the operator type

    // Parse the right-hand side expression
    auto rhs = parseExpression();
    if (!rhs) {
        delete node;
        return nullptr;
    }
    node->children.push_back(rhs);
    return node;
}
```

This section handles assignment statements. It ensures there is an identifier (like a variable name) before the assignment operator, supports both simple (=) and compound (+=, -=, etc.) assignments, and reports errors if the operator is missing. It then parses the right side of the assignment (the expression being assigned) and adds it to the parse tree.

```cpp
ParseNode* SyntaxAnalyzer::parseExprStmt() {
    auto node = new ParseNode("ExprStmt");
    auto expr = parseExpression();
    if (!expr) return nullptr;
    node->children.push_back(expr);
    return node;
}

//—— Expression (E ::= T { (+|-) T }) ——

ParseNode* SyntaxAnalyzer::parseExpression() {
    // Skip any leading whitespace
    while (!isAtEnd() && currentToken().type == TokenType::WHITESPACE) {
        advance();
    }

    // Block endings are not expressions
    if (currentToken().type == TokenType::DEDENT ||
        currentToken().type == TokenType::NEWLINE ||
        currentToken().type == TokenType::ENDOFFILE) {
        return nullptr;
    }

    auto left = parseTerm();
    if (!left) return nullptr;

    while (!isAtEnd()) {
        // Skip whitespace between terms
        while (!isAtEnd() && currentToken().type == TokenType::WHITESPACE) {
            advance();
        }

        // Stop at block endings
        if (currentToken().type == TokenType::DEDENT ||
            currentToken().type == TokenType::NEWLINE ||
            currentToken().type == TokenType::ENDOFFILE ||
            currentToken().lexeme == ":") {
            break;
        }

        std::string op;
        if (match("+")) op = "+";
        else if (match("-")) op = "-";
        else break;
```

This page shows how the parser handles expressions. It first parses a basic expression and, if successful, creates a node for it. If the expression is invalid or incomplete, it returns null to signal an error

And responsible for building the parse tree for arithmetic expressions involving operators like + or -. After an operator is found, the code first skips any spaces that appear immediately after the operator. It then tries to parse the next term in the expression (for example, the number or variable on the right side of the operator). If successful, it creates a new node in the parse tree to represent the operator, and attaches both the left and right sides of the operation as children of this node. This way, the tree keeps track of the structure of complex expressions like a + b - c.

```cpp
//—— Factor (F ::= '(' E ')' | number | identifier ) ——

ParseNode* SyntaxAnalyzer::parseFactor() {
    std::cout << "parseFactor: current token type=" << tokenTypeToString(currentToken().type)
    << ", lexeme='" << currentToken().lexeme
    << "', line=" << currentToken().line
    << ", col=" << currentToken().column << std::endl;

    // Skip any leading whitespace
    while (!isAtEnd() && currentToken().type == TokenType::WHITESPACE) {
        advance();
        std::cout << "  Skipped whitespace" << std::endl;
    }

    // Parenthesized expression
    if (match("(")) {
        std::cout << "  Parsing parenthesized expression" << std::endl;
        auto expr = parseExpression();
        if (!match(")")) {
            addSyntaxError("Expected ')' after expression",
                           currentToken().line, currentToken().column);
            delete expr;
            return nullptr;
        }
        return expr;
    }

    const Token& tok = currentToken();
    std::cout << "  Checking token: type=" << tokenTypeToString(tok.type)
              << ", lexeme='" << tok.lexeme << "'" << std::endl;

    // String literals
    if (tok.type == TokenType::STRING) {
        std::cout << "  Found string literal" << std::endl;
        auto leaf = new ParseNode("String", QString::fromStdString(tok.lexeme));
        advance();
        return leaf;
    }

    // Boolean literals
    if (tok.type == TokenType::KEYWORD &&
        (tok.lexeme == "True" || tok.lexeme == "False"))
    {
        std::cout << "  Found boolean literal" << std::endl;
        auto leaf = new ParseNode("Bool", QString::fromStdString(tok.lexeme));
        advance();
        return leaf;
    }

    // Identifier or function call
    if (tok.type == TokenType::IDENTIFIER) {
        std::cout << "  Found identifier: " << tok.lexeme << std::endl;
        std::string name = tok.lexeme;
        advance();

        // Skip whitespace after identifier
        while (!isAtEnd() && currentToken().type == TokenType::WHITESPACE) {
            advance();
            std::cout << "  Skipped whitespace after identifier" << std::endl;
        }
```

This section defines how the parser handles the smallest building blocks of expressions, called "factors."

The function starts by printing debug info about the current token.

It skips any whitespace, then checks for different possibilities:

Parentheses: If it finds a '(', it parses a full sub-expression inside the parentheses.

String literals: If the token is a string, it creates a node for it.

Boolean literals: If the token is "True" or "False", it creates a boolean node.

Identifiers and function calls: If the token is an identifier (like a variable or function name), it prepares to handle it.

It also skips whitespace after an identifier, in case there’s extra space.

```
        // Function call: IDENTIFIER '(' [args] ')'
        if (match("(")) {
            std::cout << "  Found function call" << std::endl;
            auto callNode = new ParseNode("FuncCall", QString::fromStdString(name));

            // Parse zero or more comma-separated arguments
            if (!isAtEnd() && currentToken().lexeme != ")") {
                do {
                    auto arg = parseExpression();
                    if (!arg) {
                        delete callNode;
                        return nullptr;
                    }
                    callNode->children.push_back(arg);
                } while (match(","));
            }

            if (!match(")")) {
                addSyntaxError("Expected ')' after function call arguments",
                               currentToken().line, currentToken().column);
                delete callNode;
                return nullptr;
            }
            return callNode;
        }

        // Plain identifier
        std::cout << "  Creating identifier node for: " << name << std::endl;
        return new ParseNode("Identifier", QString::fromStdString(name));
    }

    // Number literals
    if (tok.type == TokenType::NUMBER ||
        tok.type == TokenType::HexadecimalNumber ||
        tok.type == TokenType::BinaryNumber ||
        tok.type == TokenType::OCTALNUMBER)
    {
        std::cout << "  Found number literal" << std::endl;
        QString nodeName;
        switch (tok.type) {
        case TokenType::NUMBER:            nodeName = "Number"; break;
        case TokenType::HexadecimalNumber: nodeName = "Hex";    break;
        case TokenType::BinaryNumber:      nodeName = "Binary"; break;
        case TokenType::OCTALNUMBER:       nodeName = "Octal";  break;
        default:                           nodeName = "Number"; break;
        }

        auto leaf = new ParseNode(nodeName, QString::fromStdString(tok.lexeme));
        advance();
        return leaf;
    }

    // If we get here, we couldn't parse a factor
    if (currentToken().type == TokenType::NEWLINE ||
        currentToken().type == TokenType::ENDOFFILE) {
        std::cout << "  Hit end of line or file" << std::endl;
        return nullptr;
    }

    std::cout << "  Failed to parse factor" << std::endl;
    addSyntaxError("Expected an identifier, number, or expression",
                   currentToken().line, currentToken().column);
    return nullptr;
}
```

This continues the parsing of factors:

Function calls: If an identifier is followed by '(', it treats it as a function call and parses any arguments (expressions separated by commas) inside the parentheses.

Plain identifiers: If it's just an identifier (variable name) by itself, it makes a node for it.

Number literals: If it finds a number (decimal, hex, binary, or octal), it creates a node with the correct type.

If none of the above cases match, it means the parser couldn't find a valid factor, so it prints an error and returns null to signal a problem.

```cpp
void SyntaxAnalyzer::buildTree(QTreeWidgetItem* parent, ParseNode* node) {
    for (auto child : node->children) {
        QTreeWidgetItem* item = new QTreeWidgetItem(parent);
        item->setText(0, child->name + (child->value.isEmpty() ? "" : ": " + child->value));
        buildTree(item, child);
    }
}

const Token& SyntaxAnalyzer::currentToken() const {
    return tokens[pos];
}

void SyntaxAnalyzer::advance() {
    if (!isAtEnd()) pos++;
}

bool SyntaxAnalyzer::match(const std::string& lexeme) {
    if (!isAtEnd() && currentToken().lexeme == lexeme) {
        advance();
        return true;
    }
    return false;
}

bool SyntaxAnalyzer::isAtEnd() const {
    return pos >= tokens.size() || currentToken().type == TokenType::ENDOFFILE;
}

void SyntaxAnalyzer::addSyntaxError(const std::string& msg, int line, int column) {
    syntaxErrors.push_back({ msg, line, column });
}
```

This page provides utility functions used throughout the parser:

buildTree: Recursively adds nodes to a tree widget for displaying the parse tree visually (for example, in a GUI).

currentToken: Returns the token currently being analyzed.

advance: Moves to the next token in the list.

match: Checks if the current token matches a specific string, and advances if it does.

isAtEnd: Checks if the parser has reached the end of the token list or the end of the file.

addSyntaxError: Records any syntax errors found, storing the message and its location for later reporting.

Graphical User Interface

GUI employs the QSpliter widget from Qt for designing flexible panels that users can resize. The main sections of the interface include two main divisions.

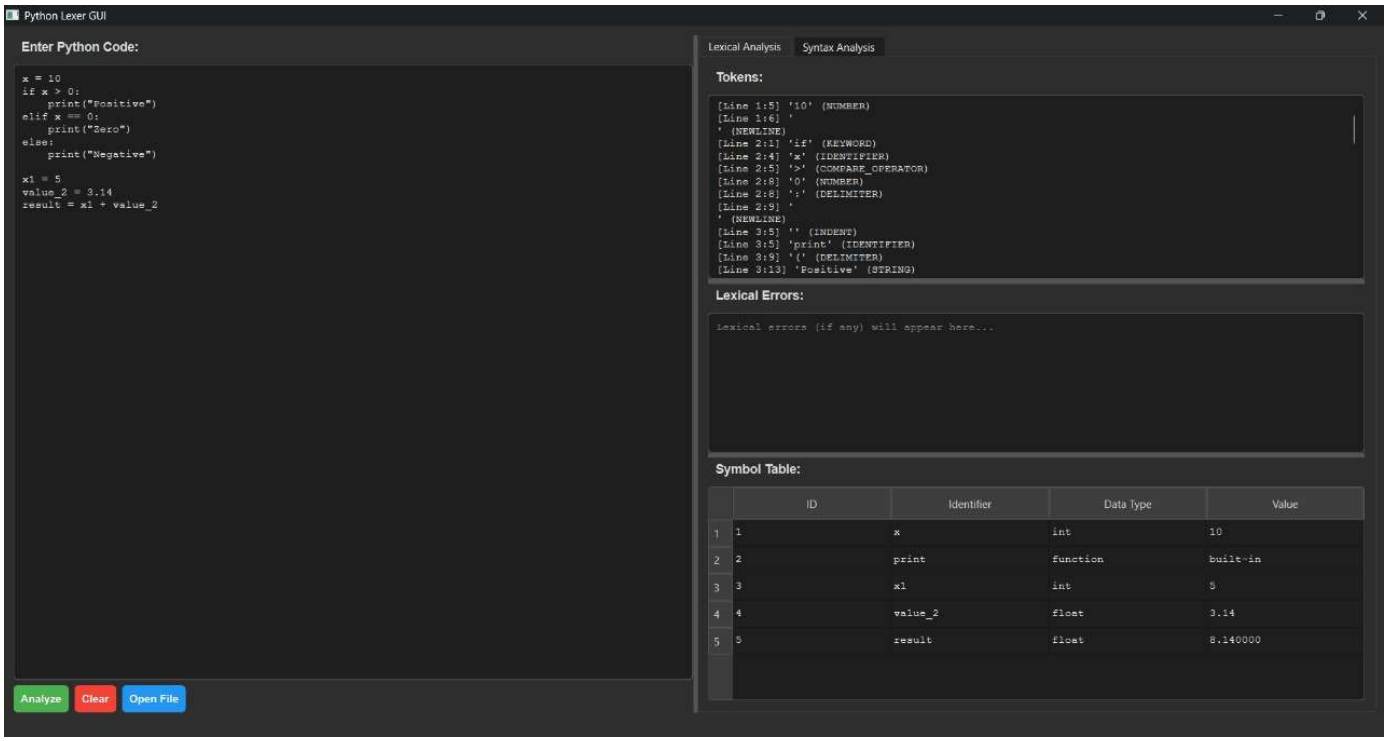
Input Panel:

The interface contains a QPlainTextEdit field that enables users to type Python code. Affixed beneath the entry field users find Analyze and Clear butons. The program starts with lexical analysis when users click the "Analyze" button and it clears all fields upon "Clear" button activation.

Output Panel:

Vertically split into three sections:

1. The Tokens section shows recognized tokens together with their line and column information in a read-only interface.
2. Errors: Lists any lexical errors such as invalid tokens or unterminated strings.
3. The Symbol Table contains all identifiers with their distinct IDs in addition to their determined data types and program-assigned values.



Demonstration of Our GUI

Testcases

We generated python testcases files for to make sure that the code covers all the statements these are sample of tests that we should include in our project in a small cases , in the code we used bigger and more complex tests to ensure more accuracy of our model.

```python
# 1. Keywords and control structures
def my_function():
    if True:
        return None
    elif False:
        pass
    else:
        global x
        x = 100

# 2. Identifiers and weird spacings
var1 = 10
_var2 = 20
__privateVar = 30
F unc = 40  # Should be treated as 'Func' by your custom rule
```

```python
# 3. Numbers (integers and floats)
integer = 123
negative = -456
floating = 3.14
exp = 1.2e10

# 4. Strings (single, double, triple quoted)
str1 = 'hello'
str2 = "world"
str3 = '''triple quoted'''
str4 = """also triple quoted"""

# 5. Comments
# this is a comment
x = 42  # inline comment
```

```python
# 6. Operators
a = 1 + 2 - 3 * 4 / 5 // 6 % 7 ** 2
b = a != 10 and a <= 100 or not False

# 7. Delimiters
my_list = [1, 2, 3]
my_tuple = (4, 5, 6)
my_dict = {'a': 1, 'b': 2}

# 8. Function and class definition with parameters
class MyClass:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print("Hello, " + self.name)
```

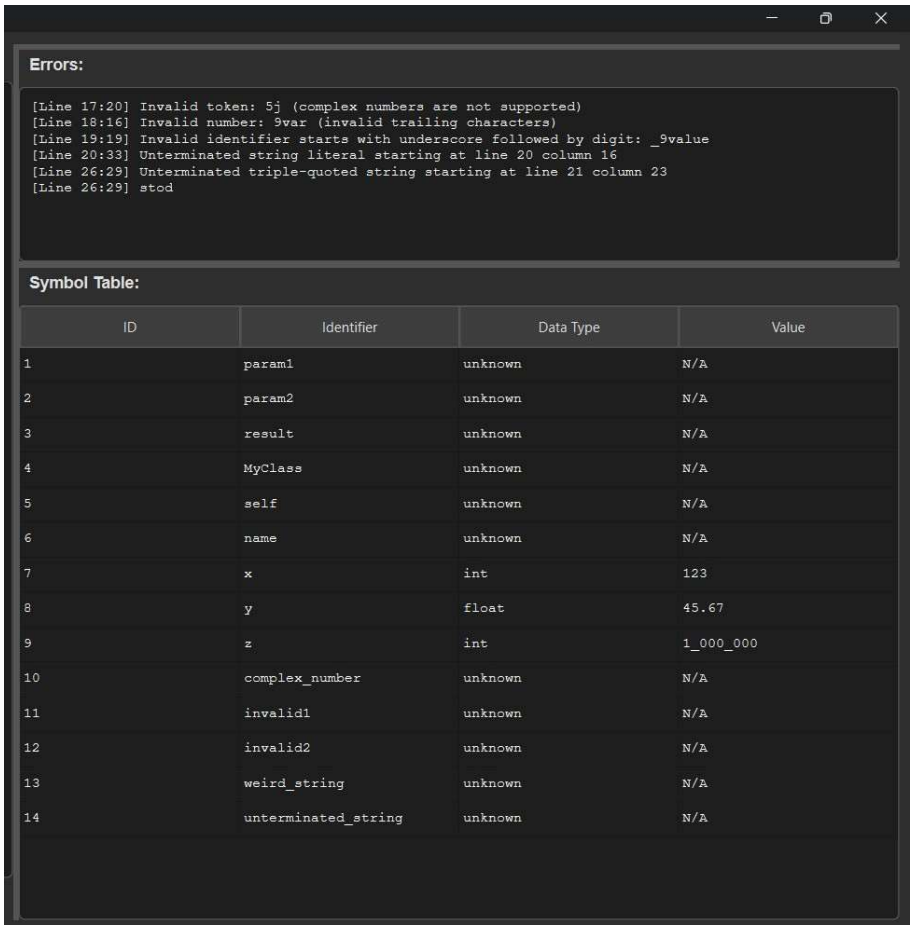
```python
# 9. Multiline string used as a docstring
"""
This is a docstring, not a comment.
"""

# 10. Combined edge cases
def calc(x,y):return x+y*2 if x!=0 else y//2
```

Identifier	Type	Value
my_function	Function	-
x	Integer	100, then 42
var1	Integer	10
_var2	Integer	20
__privateVar	Integer	30
Func	Integer	40
integer	Integer	123
negative	Integer	-456
floating	Float	3.14
exp	Float (Sci)	1.2e10
str1	String	'hello'
str2	String	"world"
str3	String	'''triple quoted'''
str4	String	"""also triple quoted"""
a	Expression	1 + 2 - 3 * 4 / 5 // 6 % 7 ** 2
b	Boolean Expr	a != 10 and a <= 100 or not False
my_list	List	[1, 2, 3]
my_tuple	Tuple	(4, 5, 6)
my_dict	Dictionary	{'a': 1, 'b': 2}
MyClass	Class	-
self	Reference	Instance of MyClass
name	Parameter/String	Passed to class constructor
greet	Method	print("Hello, " + self.name)
calc	Function	return x + y * 2 if x != 0 else y // 2
y	Parameter	-

Here is a preview of our testing of Lexer

First Testcase:

```python
# This is a comment
def my_function(param1, param2):
    """This is a docstring"""
    if param1 > 10 and param2 < 20:
        result = param1 + param2
        return result
    else:
        return None

class MyClass:
    def _init_(self, name):
        self.name = name

x = 123
y = 45.67
z = 1_000_000
complex_number = 5j
invalid1 = 9var
invalid2 = _9value
weird_string = "This is a string
unterminated_string = """Multi-line
but never ends...

multi_ops = a == b and c != d or e <= f
punctuation = (a, b; c: d)
arrow_func = lambda x: x + 1
```



Errors:

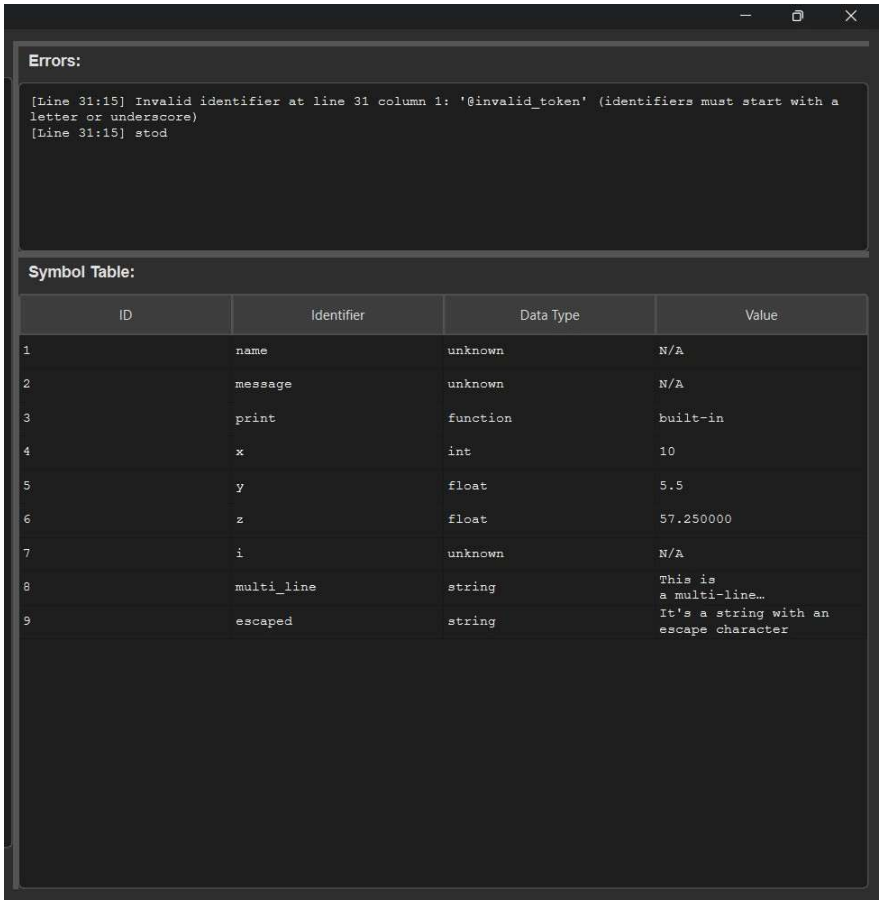
```
[Line 17:20] Invalid token: 5j (complex numbers are not supported)
[Line 18:16] Invalid number: 9var (invalid trailing characters)
[Line 19:19] Invalid identifier starts with underscore followed by digit: _9value
[Line 20:33] Unterminated string literal starting at line 20 column 16
[Line 26:29] Unterminated triple-quoted string starting at line 21 column 23
[Line 26:29] stod
```

Symbol Table:

ID	Identifier	Data Type	Value
1	param1	unknown	N/A
2	param2	unknown	N/A
3	result	unknown	N/A
4	MyClass	unknown	N/A
5	self	unknown	N/A
6	name	unknown	N/A
7	x	int	123
8	y	float	45.67
9	z	int	1_000_000
10	complex_number	unknown	N/A
11	invalid1	unknown	N/A
12	invalid2	unknown	N/A
13	weird_string	unknown	N/A
14	unterminated_string	unknown	N/A

Second Testcase:

```
# This is a single-line comment

def greet(name):  # function definition
    message = "Hello, " + name + "!"  # string concatenation
    print(message)

greet("World")  # function call

# Variables and operators
x = 10
y = 5.5
z = x * y + (x - y) / 2

# Conditional block
if z > 20:
    print("z is large")
else:
    print("z is small")

# Loop with indentation
for i in range(3):
    print("Iteration:", i)

# Multi-line string and escape characters
multi_line = """This is
a multi-line
string."""
escaped = 'It\'s a string with an escape character'

# Invalid token (should trigger an error if your analyzer checks for it)
@invalid_token
```



Errors:

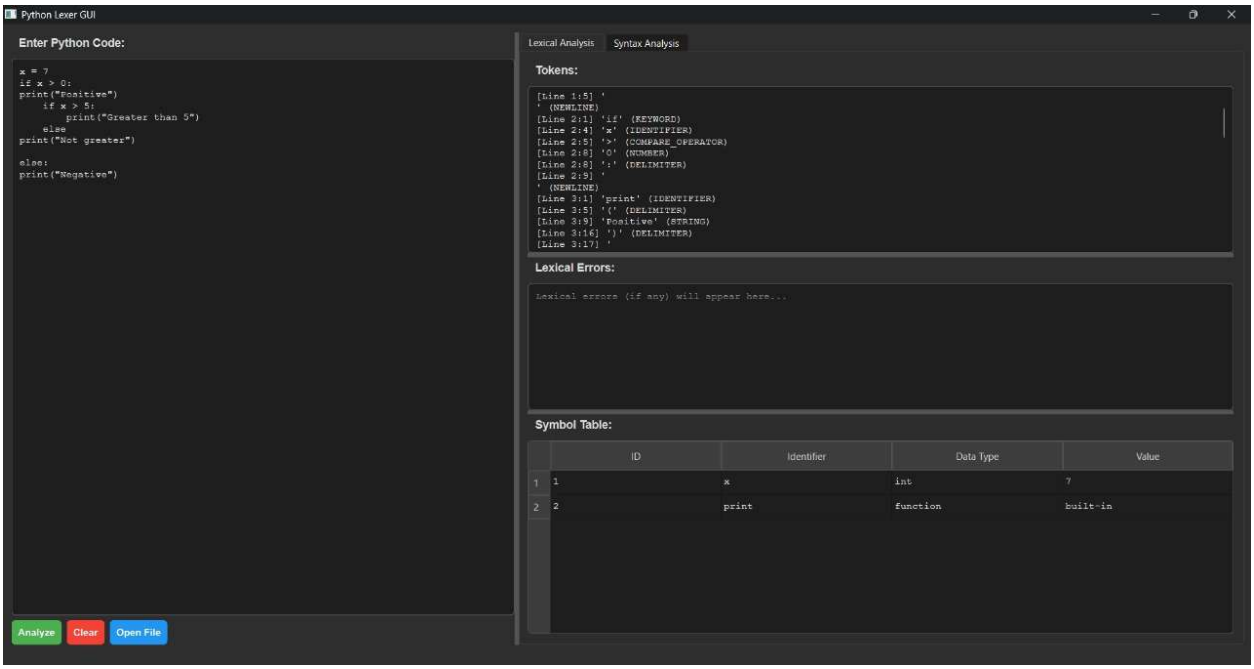
```
[Line 31:15] Invalid identifier at line 31 column 1: '@invalid_token' (identifiers must start with a
letter or underscore)
[Line 31:15] stod
```

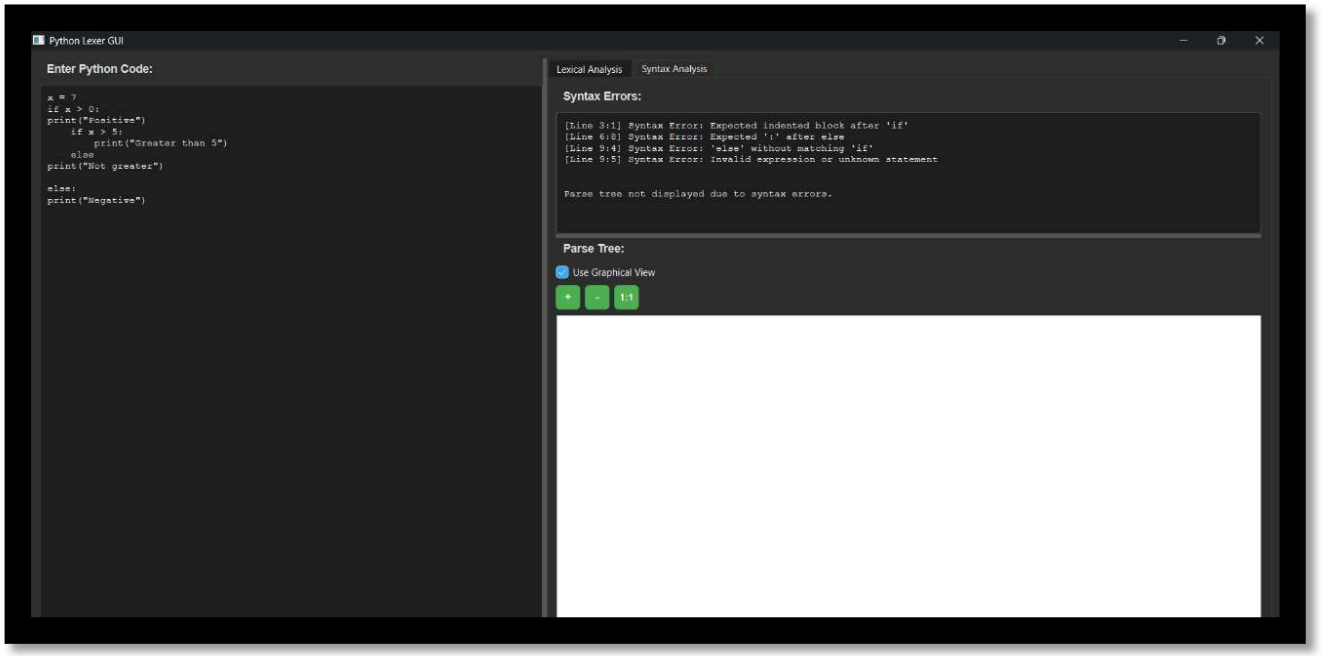
Symbol Table:

ID	Identifier	Data Type	Value
1	name	unknown	N/A
2	message	unknown	N/A
3	print	function	built-in
4	x	int	10
5	y	float	5.5
6	z	float	57.250000
7	i	unknown	N/A
8	multi_line	string	This is a multi-line…
9	escaped	string	It's a string with an escape character

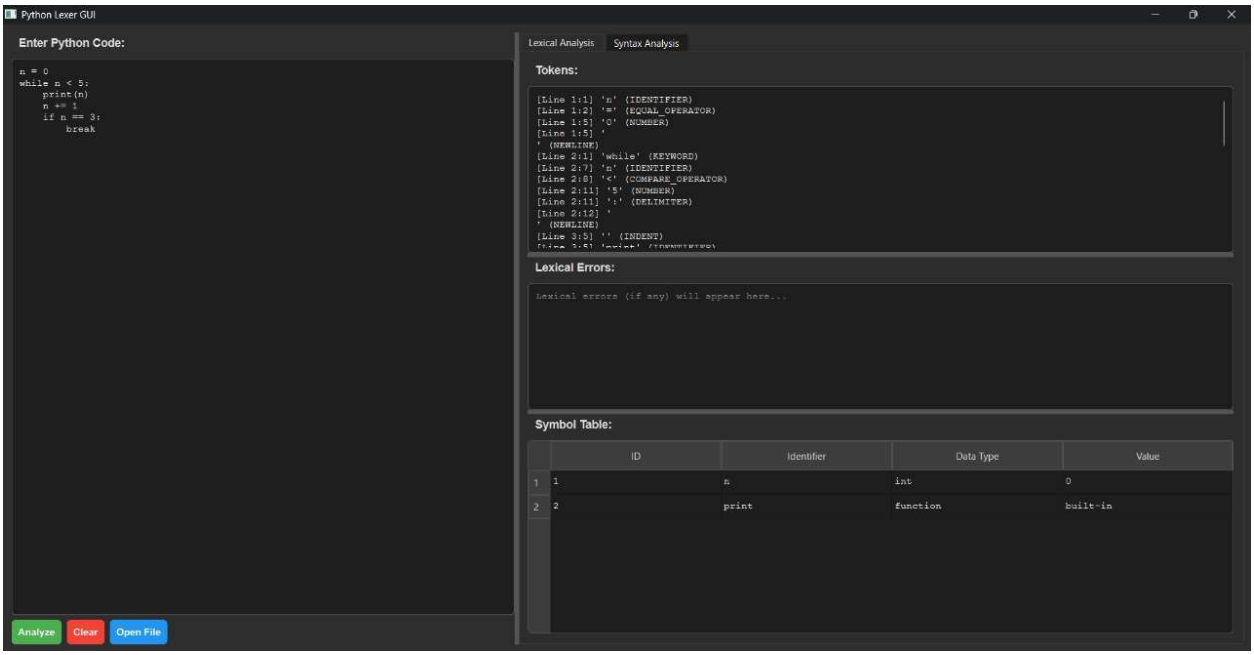
Testing Project as a Whole

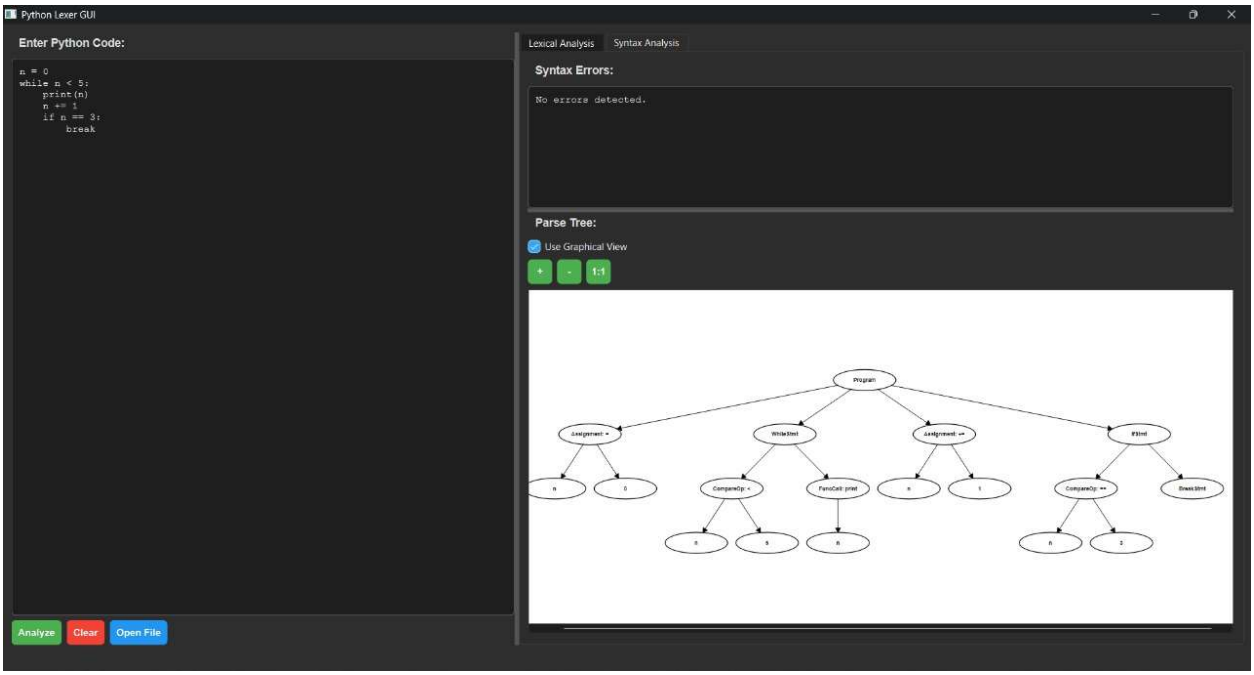
Case when there are errors in the code





Case when there is no errors





Conclusion

Completing this project has been a rewarding and eye-opening journey for every member of our team. We started with just a basic idea of what a compiler does, but by working together, we managed to create a real, working system that brings the inner workings of Python code to life.

Our compiler front-end doesn't just break code into tokens or check for syntax errors—it helps users understand the structure of their programs, spot mistakes, and see what's really happening behind the scenes. The interactive GUI means anyone can experiment with code, test different scenarios, and instantly get feedback.

References

1 https://docs.python.org/3/tutorial/controlflow.html

2 https://www.geeksforgeeks.org/conditional-statements-in-python/

3 https://stackoverflow.com/questions/9195455/how-to-document-a-method-with-parameters

4 https://www.datacamp.com/tutorial/functions-python-tutorial

5 https://www.linode.com/docs/guides/if-statements-and-conditionals-in-python/

6 **Eric Matthes, Python Crash Course, No Starch Press, 2019**

7 **lecture Slides**