Universidad Nacional Autónoma de México

Ingeniería en Computación

Compiladores

Interpreter

TEAM 4:
422023645

Grupo:
#5
Semestre:
2025-I

México,CDMX. Noviembre 2024

Index

1. Introduction

1.1 Problem Statement

An interpreter is a program that reads and executes source code written in a high-level programming language. Compared to a compiler, the compiler translates code into machine language before execution, an interpreter directly executes the instructions line by line. The creation of this interpreter involves different phases, starting with the lexical analyzer to break down the source code into tokens, followed by the parser to analyze the structure, and then the Semantic Directed Translation or SDT to add meaning to the parsed structure. After these phases, code optimization should be employed to improve efficiency, and finally, the interpreter executes the code directly. [1] [2]

1.2 Motivation

Building an interpreter helps us understand the key concepts of programming language processing and how code is executed. Unlike a compiler, an interpreter gives immediate feedback on each line of code, making it useful for learning, quick testing, and debugging. This project builds upon the phases we have already developed: the lexer and parser, and adds further phases like: code optimization, and generating the final output code, that are seen theoretically. Our motivation is to have more experience with language processing and understand the challenges of interpreting code in real time. [3]

1.3 Objectives

- Develop an interpreter for a high-level programming language using Python and the PLY library.

- ➢ Integrate the previously developed lexer and parser to create a cohesive system that analyzes, optimizes, and executes source code.
- ➢ Design mechanisms to dynamically execute expressions and statements.
- ➢ Evaluate the correctness and efficiency of the interpreter through rigorous testing with varied inputs.
- ➢ Better understanding of the entire interpretation process.

2. Framework

This framework outlines the theoretical foundation for developing a Python interpreter that integrates lexical analysis, syntax analysis, and semantic analysis using the PLY library. The goal of this project is to process user inputs such as variable declarations (int a = 10), evaluate mathematical expressions, and implement advanced features like solving the quadratic formula.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

1. Lexical Analysis: Tokenizing Input

The first step in our interpreter is to break down the input into manageable components called tokens. This process, known as lexical analysis, ensures that each piece of the input is classified according to its role (e.g., keywords, identifiers, operators, literals).

How It Was Used?

- **Token Categories**: The project required defining categories such as keywords (print, int), operators (+, *, =), and literals (e.g., integers).
- **Regular Expressions**: These patterns identify tokens from the input string. For example, a regex like \d+ is used to match numbers.
- **Handling Errors**: The lexical analyzer skips invalid characters or reports errors when unexpected inputs are encountered.

2. Syntax Analysis: Parsing Tokens

The next step, syntax analysis, validates the structure of the input. The goal is to determine if the sequence of tokens matches the grammar rules of the language being interpreted. For instance, the input int a = 10 follows the pattern of a variable declaration.

Why Syntax Analysis?

Syntax analysis provides structure to the input. Without parsing, the interpreter would struggle to differentiate valid expressions from errors. It also organizes tokens into a hierarchical format, often represented as a parse tree.

How It Was Used?

- **Grammar Definition**: The interpreter uses rules to describe the language. For example, variable declarations are defined as <type> <identifier> = <value>.
- **Operator Precedence**: To handle expressions like a+b*a, the parser ensures that multiplication has higher precedence than addition.
- **Shift-Reduce Parsing**: A bottom-up parsing method is employed to build the parse tree from tokens, reducing them to higher-level constructs based on grammar rules.

3. Semantic Analysis: Adding Meaning

While syntax analysis checks structural correctness, semantic analysis ensures the input is logically valid. For example, it verifies that variables are declared before use, checks type compatibility in expressions, and enforces scope rules.

Without semantic analysis, the interpreter could process syntactically correct but logically flawed code. For instance, the statement a*b is invalid if a and b are not declared. Semantic analysis adds a layer of logical validation.

How It Was Used?

- **Symbol Table**: A dictionary-like structure stores information about variables, including their names, types, and values.
- **Type Checking**: The interpreter ensures operations involve compatible types. For example, adding a float to an integer data would result in an error.
- **Error Detection**: When undefined variables or mismatched types are encountered, the semantic analyzer provides meaningful feedback.

4. Expression Evaluation

Evaluating mathematical expressions is central to the interpreter’s functionality. Expressions like a+b or a+b*a are processed dynamically, respecting the order of operations.

Expression evaluation allows the interpreter to compute results dynamically based on user input. It also enables advanced features like solving the quadratic formula.

How It Was Used?

- **Recursive Evaluation**: The interpreter traverses the parse tree to compute results, starting from the deepest nodes (e.g., handling b*b before a+a).
- **Operator Precedence**: Multiplication and division are given higher priority than addition and subtraction, aligning with mathematical conventions.
- **Variable Substitution**: Identifiers (e.g., a, b) are replaced with their values from the symbol table during evaluation.

5. Quadratic Formula Implementation

The interpreter extends its capabilities by solving quadratic equations. This requires evaluating the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This demonstrates the interpreter's ability to handle complex mathematical expressions and perform advanced computations based on user-defined inputs.

How It Was Used?

- **Mathematical Functions**: Square roots and powers are incorporated into the grammar, expanding the parser's functionality.
- **Dynamic Evaluation**: The user declares a, b, and c as variables, and the formula is evaluated using their values.
- **Error Handling**: Special cases, such as negative discriminants ($b^2 - 4ac < 0$), are managed by using python mathematical tools (cmath). Other cases such as indeterminate / indefinite operations are also handled by the interpreter.

6. Integration of Concepts

The interpreter combines the key concepts from lexical, syntax, and semantic analysis:

- **Lexical Analysis** provides structured tokens from raw input.
- **Syntax Analysis** organizes these tokens into a meaningful hierarchy.
- **Semantic Analysis** validates the logic of the input, ensuring meaningful execution.

The quadratic formula and expression evaluation serve as practical applications of these concepts, showcasing how they work together to form a cohesive interpreter.

3. Body

The lexer is the first stage of the interpretation process, taking care of splitting the source code into tokens that the parser can interpret. In the code, the lexer is implemented using the Python PLY library, which allows defining regular expressions to identify different types of tokens.

Reserved words were defined, being keywords such as while, if, else, for, among others, which are assigned to specific constants to facilitate interpretation in the parsing stage.

General tokens are classified: including tokens for numbers (NUMBER), operators (LESS, MORE, DIVISION, MULTIPLICATION, etc.), parentheses and other programming symbols.

Negative number handling was implemented; the lexer also recognizes negative numbers, which is fundamental for mathematical operations, especially when performing exponential operations with square roots of negative numbers.

The parser receives the tokens generated by the lexer and organizes them in a hierarchical structure according to the grammatical rules of the language. In the code, a set of rules is implemented for expressions, assignments and print statements.

Arithmetic operations and value assignments, the parser defines hierarchy rules for operations such as addition, subtraction, multiplication, division and modulo. It also manages value assignments to variables, which can be with explicit or implicit type, that is, without declared type.

For the handling of unary operators, operators such as “+” and “-” are handled explicitly, with special attention to the precedence of these unary operators.

For operator precedence, a precedence table is used to ensure that complex expressions are evaluated in the correct order.

One of the most important parts of the code is the evaluation of expressions, which is where

the main work of the interpreter takes place: the execution of operations and the manipulation of values in real time.

The evaluation process includes:

- Evaluation of arithmetic and exponential operations: mathematical expressions are solved using the operators and operands present in the code. Especially important is the handling of exponentiation, which includes support for square roots of negative numbers, using Python's cmath module to work with complex numbers.
- Variable management: The interpreter allows assigning values to variables, storing them in a symbol table “symbol_table”. This aspect is clearly related to the behavior of a linker, since the interpreter must manage the references between symbols. When using a variable in an expression, the interpreter consults the symbol table to obtain the value of that variable, similar to how a linker resolves references between different parts of the code.
- Expression evaluation can be considered a form of “run-time resolution”. Although a linker performs compile-time resolution, the interpreter resolves these dependencies dynamically as it executes the program.

About the symbol table, the symbol table is a structure in which variables defined during code execution are stored. Each variable has a type and a value associated with it, which allows the interpreter to perform assignments and queries efficiently.

From the type assignment, the interpreter validates that the variables are assigned with values compatible with their type, either int or float. If an incompatible value is assigned, an error is generated.

Regarding the dynamic management of variables, the interpreter also allows assigning values to variables without an explicit type, which gives it flexibility.

4. Results

In the following screenshots we will see the outputs shown by the interpreter in different situations:

- Test 1:

  Starting from the following quadratic equation:

  $$x^2 - 5x + 6 = 0$$

  we define the variables with the coefficients of each term:

```
>> int a = 1
>> int b = -5
>> int c = 6
>> int neg = -1
```

Screenshot T1.1 - Declaring the inputs to the equation.

NOTE: we add an extra variable (neg) to handle subtractions.

Then we declare two variables that will be defined by Bhaskara's Formula:

```
>> float x_1 = (-b+((b^2)+(neg*4*a*c))^0.5)/(2*a)
>> float x_2 = (-b-((b^2)+(neg*4*a*c))^0.5)/(2*a)
```

Screenshot T1.2 - Defining the solution to the equation.

Finally, we print the results of the variables, giving the correct results.

```
>> print(x_1)
Result: 3.0
>> print(x_2)
Result: 2.0
```

Screenshot T1.3 - Results of applying the general equation.

- Test 2:

Now, we will see the use of roots of negative numbers, that is, with results in complex numbers.

First, we define a negative variable, for practical purposes, we will use the integer -4.

```
>> int d = -4
```

Screenshot T2.1 - Declaring the input.

Subsequently, we print the square root of the number, knowing that:

$$(-4)^{\frac{1}{2}} = (-4)^{0.5} = \sqrt{-4}$$

Finally, we see the results:

```
>> print((d)^0.5)
Result: 2j
```

Screenshot T2.2 - Results of applying square root to a negative number.

- Test 3:

Combining the two previous tests, with a quadratic equation which we know that its roots are complex numbers with real part and imaginary part:

$$x^2 + 4x + 5 = 0$$

We look at the variable definitions and results:

```
>> int a = 1
>> int b = 4
>> int c = 5
>> int neg = -1
>> print((-b+((b^2)+(neg*4*a*c))^0.5)/(2*a))
Result: (-2+1j)
>> print((-b-((b^2)+(neg*4*a*c))^0.5)/(2*a))
Result: (-2-1j)
```

Screenshot T3.1 - Defining the inputs of the equation and the expression to see the complex roots with real and imaginary parts.

- Test 4 (Errors):

In the following we will see the possible errors that can occur, with their respective outputs:

```
>> int a 5
Syntax error at '5'
```

Screenshot T4.1 - Syntax error: missing '=' before '5'.

```
>> int pi = 3.14
Error: Type mismatch: variable 'pi' of type 'int' cannot hold value
'3.14'.
```

Screenshot T4.2 - Semantic error: semantic error: assigning a floating value to an integer data type.

```
>> print(0/0)
Error: Division by zero is undefined.
>> print(a/0)
Error: Division by zero is undefined.
>> print(0^0)
Error: The operation 0^0 is undefined (for certain cases, the result
is 1).
```

Screenshot T4.3 - Errors in indeterminate / indefinite operations.

5. Conclusion

The progressive implementation of the project across different deliverables was fundamental to its success. By dividing the development into clear stages (lexical, syntactic, and semantic analysis), it was possible to build a robust and scalable system. Each stage allowed us to focus on solving specific problems before advancing to the next level, ensuring a more efficient and organized integration of all components. Finally, by combining and linking each of these analyses, a functional interpreter was achieved, capable of handling mathematical expressions, performing type validations, and executing operations with precise results.

Another crucial aspect was the incorporation of mathematical concepts into the project, which not only enriched its functionality but also proved essential for its implementation. For instance, operations like 0^0, which require special handling due to their mathematical nature, were addressed; complex number handling was included for cases like square roots of negative numbers, ensuring the interpreter could solve problems beyond the basics. Furthermore, the implementation of the quadratic formula as part of the available operations highlights the importance of mathematics in designing advanced computational tools.

This progressive and interdisciplinary approach not only facilitated technical implementation but also underscored the necessity of a solid foundation in mathematics and programming. It enabled the development of a product that not only meets the initial objectives but also lays the groundwork for future improvements and extensions.

6. Bibliography

[1] “Language Processors: Assembler, Compiler and Interpreter - GeeksforGeeks,” GeeksforGeeks, Aug. 09, 2018. https://www.geeksforgeeks.org/language-processors-assembler-compiler-and-interpreter/ (Nov. 24, 2024).

[2] S. Thakur, “Difference Between Compiler, Interpreter And Assembler,” Unstop.com, Mar. 25, 2022. https://unstop.com/blog/difference-between-compiler-interpreter-assembler (Nov. 24, 2024).

[3] GeeksforGeeks, “Introduction to Interpreters,” GeeksforGeeks, May 30, 2020. https://www.geeksforgeeks.org/introduction-to-interpreters/ (Nov. 24, 2024).

[4] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson Education.

[5] Beazley, D. M. (2001). *PLY (Python Lex-Yacc)*. Recuperado de http://www.dabeaz.com/ply/