Lecture 03: A_mini_interpreter

Vincenzo Ciancia

October 23, 2025

Section 1: Introduction to Interpreters

An interpreter is a program that executes source code directly, without requiring compilation to machine code. Let's explore how interpreters work and how to build a simple one in Python.

The Role of Interpreters

Interpreters serve several important purposes in programming language implementation:

1. **Direct Execution**: Execute source code without a separate compilation step
2. **Immediate Feedback**: Provide instant results for interactive programming
3. **Portability**: Run on any platform that supports the interpreter
4. **Simplicity**: Often easier to implement than full compilers
5. **Debugging**: Allow for interactive debugging and inspection

Languages like Python, JavaScript, and Ruby primarily use interpreters, while others like Java use a hybrid approach with compilation to bytecode followed by interpretation.

Components of an Interpreter

A typical interpreter includes the following components:

1. **Lexer (Tokenizer)**: Converts source code text into tokens
2. **Parser**: Transforms tokens into an abstract syntax tree (AST)
3. **Evaluator**: Executes the AST to produce results
4. **Environment**: Stores variables and their values
5. **Error Handler**: Manages and reports errors

We'll implement each of these components in our mini interpreter.

Section 2: Using Lark Parser

Instead of manually implementing a lexer and parser, we'll use the Lark parsing library, which provides a powerful and declarative way to parse expressions.

Why Use Lark?

Lark is a modern parsing library for Python that offers several advantages:

1. **Declarative Grammar**: Define your language using EBNF notation
2. **Automatic Tokenization**: No need to write a manual lexer
3. **Parse Tree Generation**: Automatically creates parse trees
4. **Pattern Matching**: Easy to transform parse trees into ASTs

Installing Lark

First, ensure Lark is installed in your environment:

```
pip install lark
```

Defining the Grammar

Our arithmetic expression grammar uses Lark's EBNF format:

```
from lark import Lark, Token, Tree


# Define the grammar in Lark's EBNF format

grammar = r"""

    expr: bin | mono

    mono: ground | paren

    paren: "(" expr ")"

    bin: expr OP mono

    ground: NUMBER


    NUMBER: /[0-9]+/

    OP: "+" | "-" | "*" | "/" | "%"


    %import common.WS

    %ignore WS

"""
```

Understanding the Grammar

The grammar defines:

- **expr**: An expression can be binary operation or monadic (single value)
- **mono**: A monadic expression is either a ground value or parenthesized expression
- **ground**: A basic number literal
- **bin**: A binary operation (left operand, operator, right operand)
- **NUMBER**: A regex pattern matching digits
- **OP**: Operators (+, -, *, /, %)
- **WS**: Whitespace is imported and ignored

Creating the Parser

```python
# Create the Lark parser
parser = Lark(grammar, start="expr")

# Parse an expression
parse_tree = parser.parse("(1 + 2) - 3")
```

Lark provides a convenient method to visualize the parse tree:

```python
print(parse_tree.pretty())
```

```
expr
  bin
    expr
      mono
        paren
          expr
            bin
              expr
                mono
                  ground       1
              +
              mono
                ground  2
    -
    mono
      ground    3
```

Section 3: Transforming Parse Trees to ASTs

While Lark creates parse trees automatically, we need to transform them into our own Abstract Syntax Tree (AST) representation for evaluation.

Defining AST Nodes

We define our AST using Python's type system and dataclasses:

```
from dataclasses import dataclass

from typing import Literal


# Define operator type

type Op = Literal["+", "-", "*", "/", "%"]
```

```python
@dataclass
class Number:
    value: int


@dataclass
class BinaryExpression:
    op: Op
    left: Expression
    right: Expression


type Expression = Number | BinaryExpression
```

Understanding AST vs Parse Tree

- **Parse Tree**: Generated by Lark, contains all grammar details
- **AST**: Simplified tree with only semantically relevant information
- **Transformation**: We convert parse trees to ASTs using pattern matching

Transforming Parse Trees with Pattern Matching

We use Python's pattern matching to transform Lark's parse trees into our AST:

```python
def transform_parse_tree(tree: Tree) -> Expression:
    match tree:
        case Tree(data="expr", children=[subtree]):
            return transform_parse_tree(subtree)

        case Tree(data="mono", children=[subtree]):
            return transform_parse_tree(subtree)

        case Tree(data="ground", children=[Token(type="NUMBER", value=actual_value)]):
            return Number(value=int(actual_value))
```

```python
        case Tree(data="paren", children=[subtree]):
            return transform_parse_tree(subtree)

        case Tree(
            data="bin",
            children=[
                left,
                Token(type="OP", value=op),
                right,
            ],
        ):
            return BinaryExpression(
                op=op,
                left=transform_parse_tree(left),
                right=transform_parse_tree(right),
            )
```

```
        case _:
            raise ValueError(f"Unexpected parse tree structure")
```

Complete Parsing Function

Combining Lark parsing with our transformation:

```python
def parse_ast(expression: str) -> Expression:
    parse_tree = parser.parse(expression)
    return transform_parse_tree(parse_tree)


# Example usage
ast = parse_ast("(1+2)-3")
```

Section 4: Evaluating Expressions

Now that we have an AST, we can evaluate it to produce a result.

The Evaluation Process

Evaluation is the process of computing the result of an expression. It typically involves:

1. Walking the AST recursively
2. Computing the value of each node based on its type and children
3. Combining results according to the language semantics

Implementing an Evaluator

We implement the evaluator using pattern matching:

```
def evaluate(ast: Expression) -> int:
    match ast:
        case Number(value):
            return value
```

```
    case BinaryExpression(op, left, right):
        left_value = evaluate(left)
        right_value = evaluate(right)
        match op:
            case "+":
                return left_value + right_value
            case "-":
                return left_value - right_value
```

```python
        case "*":
            return left_value * right_value
        case "/":
            if right_value == 0:
                raise ValueError("Division by zero")
            return left_value // right_value
        case "%":
            if right_value == 0:
                raise ValueError("Division by zero")
            return left_value % right_value
```

Convenience Function

We can combine parsing and evaluation:

```
def evaluate_string(expression: str) -> int:
    ast = parse_ast(expression)
    return evaluate(ast)


# Example
result = evaluate_string("(1+2)*3")  # Returns 9
```

Section 5: Building a REPL

Now we'll create a Read-Eval-Print Loop (REPL) for interactive use.

Implementing the REPL

```python
def REPL():
    exit = False
    while not exit:
        expression = input("Enter an expression (exit to quit): ")
        if expression == "exit":
            exit = True
        else:
            try:
                print(evaluate_string(expression))
            except Exception as e:
                print(e)
```

Running the REPL

```
# Start the interactive interpreter

REPL()
```

Example Session

```
Enter an expression (exit to quit): 1+2

3

Enter an expression (exit to quit): (10+20)*2

60

Enter an expression (exit to quit): 100/0

Division by zero

Enter an expression (exit to quit): 17%5

2

Enter an expression (exit to quit): exit
```

Section 6: Debugging Parse Trees

Lark provides useful tools for debugging and understanding parse trees.

Printing Parse Trees

You can inspect the raw parse tree structure:

```python
def print_tree(tree: Tree | Token):
    match tree:
        case Tree(data=data, children=children):
            print("Tree", data)
            for child in children:
                print_tree(child)
        case Token(type=type, value=value):
            print("Token", type, value)


# Usage
parse_tree = parser.parse("(1+2)-3")
print_tree(parse_tree)
```

Pretty Printing

Lark's built-in pretty printer is even more convenient:

```
parse_tree = parser.parse("(1+2)-3")

print(parse_tree.pretty())
```

This shows the hierarchical structure with indentation, making it easy to understand how the parser interpreted the expression.

Section 7: Extending the Interpreter

Our mini interpreter is basic but extensible. Here are some ideas for enhancements:

Possible Extensions

1. **Variables**: Add variable storage and assignment
2. **Functions**: Support function definitions and calls
3. **More Operators**: Add comparison ($<$, $>$, $==$) and logical operators (and, or, not)
4. **Control Flow**: Implement if-then-else expressions
5. **Type System**: Add type checking before evaluation
6. **Error Messages**: Improve error reporting with line numbers and context

Modifying the Grammar

To extend the language, you can modify the Lark grammar:

```
# Add comparison operators
grammar = r"""
    expr: comparison | bin | mono
    comparison: expr COMP mono
    mono: ground | paren
    paren: "(" expr ")"
    bin: expr OP mono
    ground: NUMBER

    NUMBER: /[0-9]+/
    OP: "+" | "-" | "*" | "/" | "%"
    COMP: "==" | "!=" | "<" | ">" | "<=" | ">="

    %import common.WS
    %ignore WS
"""
```

Key Takeaways

1. **Lark simplifies parsing**: No need to write manual lexers and parsers
2. **Pattern matching is powerful**: Transforming parse trees to ASTs is elegant with pattern matching
3. **Recursive evaluation**: AST evaluation naturally uses recursion
4. **Extensibility**: The architecture supports easy addition of new features

Conclusion

We've built a mini interpreter that:

1. Uses Lark to parse arithmetic expressions into parse trees
2. Transforms parse trees into ASTs using pattern matching
3. Evaluates ASTs recursively to produce results
4. Provides an interactive REPL for testing

This foundation can be expanded to build more complex languages and tools.