# Lecture 03: A_mini_interpreter

Vincenzo Ciancia

April 04, 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: Lexical Analysis

The first step in interpreting code is breaking it down into tokens - the smallest meaningful units in the language.

## Tokens and Lexical Structure

Tokens are the building blocks of a language, similar to words in natural language. Common token types include:

- **Keywords**: Reserved words with special meaning (e.g., `if`, `while`)
- **Identifiers**: Names given to variables, functions, etc.
- **Literals**: Constant values (numbers, strings, booleans)
- **Operators**: Symbols that perform operations (`+`, `-`, `*`, `/`)
- **Punctuation**: Symbols that structure the code (`;`, `,`, `{}`, `()`)

## Implementing a Lexer

Our lexer will convert a string of source code into a list of tokens:

```
from dataclasses import dataclass
from enum import Enum, auto
from typing import List, Optional


# Define token types
class TokenType(Enum):
    NUMBER = auto()
    PLUS = auto()
    MINUS = auto()
    MULTIPLY = auto()
    DIVIDE = auto()
    LPAREN = auto()
    RPAREN = auto()
    EOF = auto()  # End of file


@dataclass
```

## Testing the Lexer

Let's test our lexer with a simple arithmetic expression:

```
def test_lexer():
    source = "3 + 4 * (2 - 1)"
    tokens = tokenize(source)

    expected = [
        Token(TokenType.NUMBER, "3"),
        Token(TokenType.PLUS),
        Token(TokenType.NUMBER, "4"),
        Token(TokenType.MULTIPLY),
        Token(TokenType.LPAREN),
        Token(TokenType.NUMBER, "2"),
        Token(TokenType.MINUS),
        Token(TokenType.NUMBER, "1"),
        Token(TokenType.RPAREN),
        Token(TokenType.EOF)
    ]
```

# Section 3: Parsing and Abstract Syntax Trees

Once we have tokens, we need to organize them into a structured representation of the program - an Abstract Syntax Tree (AST).

## Understanding Abstract Syntax Trees

An AST is a tree representation of the abstract syntactic structure of source code. Each node in the tree represents a construct in the source code.

For example, the expression `3 + 4 * 2` would be represented as:

```
    (+)
   /   \
  3    (*)
      /   \
     4     2
```

The tree captures the structure and precedence of operations.

# Defining AST Nodes

Let's define classes for our AST nodes:

```
from dataclasses import dataclass

from typing import Union, Optional


@dataclass

class Number:

    value: float


@dataclass

class BinaryOp:

    left: 'Expression'

    operator: str

    right: 'Expression'


# Define our Expression type

Expression = Union[Number, BinaryOp]
```

## Implementing a Recursive Descent Parser

A recursive descent parser is a top-down parser that uses a set of recursive procedures to process the input:

```
class Parser:
    def __init__(self, tokens: List[Token]):
        self.tokens = tokens
        self.current = 0

    def peek(self) -> Token:
        return self.tokens[self.current]

    def previous(self) -> Token:
        return self.tokens[self.current - 1]

    def advance(self) -> Token:
        if not self.is_at_end():
            self.current += 1
        return self.previous()
```

## Testing the Parser

Let's test our parser with the same expression:

```
def test_parser():
    tokens = tokenize("3 + 4 * 2")
    parser = Parser(tokens)
    ast = parser.parse()

    # Expected: BinaryOp(Number(3), "+", BinaryOp(Number(4), "*", Number(2)))
    expected = BinaryOp(
        Number(3),
        "+",
        BinaryOp(
            Number(4),
            "*",
            Number(2)
        )
    )
```

# 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

For our mini interpreter, we'll implement a simple evaluator that computes arithmetic expressions:

```
def evaluate(expr: Expression) -> float:
    """Evaluate an expression and return its value."""
    match expr:
        case Number(value):
            return value
        case BinaryOp(left, operator, right):
            left_value = evaluate(left)
            right_value = evaluate(right)

            match operator:
                case "+":
                    return left_value + right_value
                case "-":
                    return left_value - right_value
                case "*":
```

## Testing the Evaluator

Let's test our evaluator with a few expressions:

```
def test_evaluator():
    expressions = [
        ("3 + 4", 7),
        ("3 * 4", 12),
        ("10 - 2", 8),
        ("20 / 5", 4),
        ("3 + 4 * 2", 11),  # Tests operator precedence
        ("(3 + 4) * 2", 14),  # Tests parentheses
    ]

    for source, expected in expressions:
        tokens = tokenize(source)
        parser = Parser(tokens)
        ast = parser.parse()
        result = evaluate(ast)
```

# Section 5: Putting It All Together

Now we'll combine our lexer, parser, and evaluator into a complete mini interpreter.

## The Full Interpreter

```
def interpret(source: str) -> float:
    """Interpret a source string and return the result."""
    try:
        tokens = tokenize(source)
        parser = Parser(tokens)
        ast = parser.parse()
        return evaluate(ast)
    except Exception as e:
        print(f"Error: {e}")
        return float('nan')  # Return NaN for errors


def run_repl():
    """Run a simple Read-Eval-Print Loop."""
    print("Mini Interpreter REPL")
    print("Enter expressions to evaluate, or 'exit' to quit")

    while True:
```

# Example Usage

## Using our interpreter:

```
Mini Interpreter REPL

Enter expressions to evaluate, or 'exit' to quit

> 3 + 4

= 7.0

> 3 * (4 + 2)

= 18.0

> 10 / (2 - 2)

Error: Division by zero

> (3 + 4) * (5 - 2)

= 21.0

> exit

Goodbye!
```

# Section 6: Extending the Interpreter

Our mini interpreter is very basic, but we can extend it with more features.

## Adding Variables

To add variable support, we need:

1. An environment to store variable bindings
2. New AST nodes for variable references and assignments
3. Updated parsing rules
4. Updated evaluation logic

Let's implement a simple environment first:

```python
@dataclass
class Environment:
    values: dict[str, float] = None

    def __post_init__(self):
        if self.values is None:
            self.values = {}

    def define(self, name: str, value: float) -> None:
        """Define a variable with the given name and value."""
```

## New AST Nodes for Variables

```python
@dataclass
class Variable:
    name: str


@dataclass
class Assignment:
    name: str
    value: 'Expression'


# Update Expression type
Expression = Union[Number, BinaryOp, Variable, Assignment]
```

# Updating the Parser

```python
class Parser:
    # ... existing code ...

    def parse(self) -> Expression:
        return self.assignment()

    def assignment(self) -> Expression:
        expr = self.expression()

        if self.match(TokenType.EQUAL):
            if isinstance(expr, Variable):
                value = self.assignment()
                return Assignment(expr.name, value)
            raise Exception("Invalid assignment target")

        return expr
```

# Updating the Evaluator

```python
def evaluate(expr: Expression, env: Environment) -> float:
    """Evaluate an expression in the given environment."""
    match expr:
        case Number(value):
            return value
        case Variable(name):
            return env.get(name)
        case Assignment(name, value):
            result = evaluate(value, env)
            env.define(name, result)
            return result
        case BinaryOp(left, operator, right):
            # ... existing code ...
```

## Adding Control Flow

We can extend our language with if-expressions:

```
@dataclass
class If:
    condition: Expression
    then_branch: Expression
    else_branch: Expression

# Update Expression type
Expression = Union[Number, BinaryOp, Variable, Assignment, If]

# Update evaluator
def evaluate(expr: Expression, env: Environment) -> float:
    match expr:
        # ... existing cases ...
        case If(condition, then_branch, else_branch):
            if evaluate(condition, env) != 0:  # Non-zero is true
                return evaluate(then_branch, env)
```

# Section 7: Further Exploration

Here are some ways you could extend our mini interpreter further:

## Additional Features to Implement

1. **Functions**: Add function declarations and calls
2. **Loops**: Implement while or for loops
3. **More Operators**: Add comparison, logical, and bitwise operators
4. **Error Handling**: Improve error messages and recovery
5. **Type System**: Add a simple type system
6. **Standard Library**: Implement built-in functions for common operations

## Learning Resources

To learn more about interpreters and language implementation:

- **“Crafting Interpreters”** by Robert Nystrom: A comprehensive guide to implementing interpreters
- **“Programming Language Pragmatics”** by Michael Scott: Covers theoretical aspects of language design
- **“Structure and Interpretation of Computer Programs”** by Abelson and Sussman: A classic text on programming language concepts

# Conclusion

Building a mini interpreter helps understand how programming languages work under the hood.
We've seen how to:

1. Tokenize source code into lexical tokens
2. Parse tokens into an abstract syntax tree
3. Evaluate the AST to produce results
4. Extend the interpreter with new features

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