Interpreter Pattern

Dmytro Razvaliaiev
Vistula University

What Interpreter Pattern is about?

- Interpreter pattern provides a way to evaluate language grammar or expression.
- It helps to convert information from one language into another.
- The language can be anything such as words in a sentence, numerical formulas or even software code.
- This pattern involves implementing an expression interface which tells to interpret a particular context.
- This pattern is used in SQL parsing, symbol processing engine, software for HRs etc.

Definitions

A Non-Terminal expression is a combination of other Non-Terminal and/or Terminal expressions.

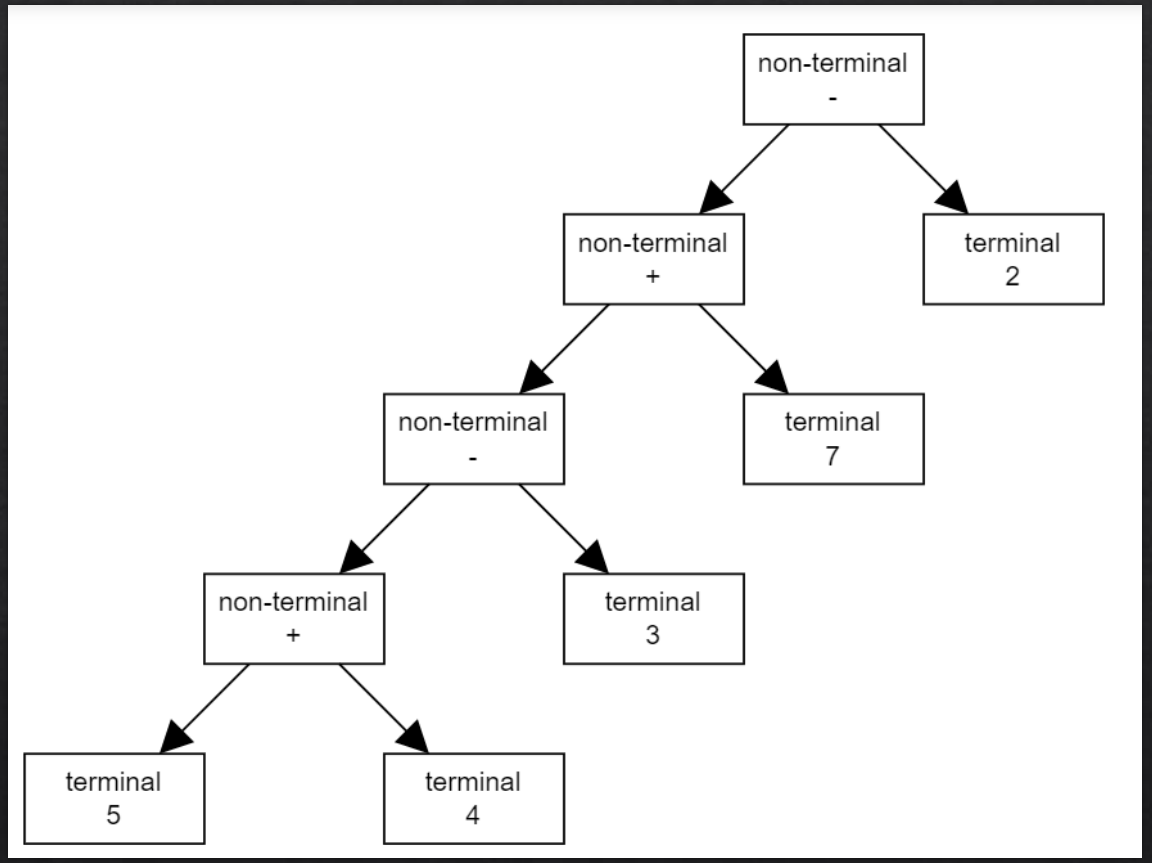
Terminal means terminated, i.e., there is no further processing involved.

An example

An example expression is **A + B**

The **A**and **B**are Terminal expressions and the **+** is Non-Terminal because it depends on the two other Terminal expressions.

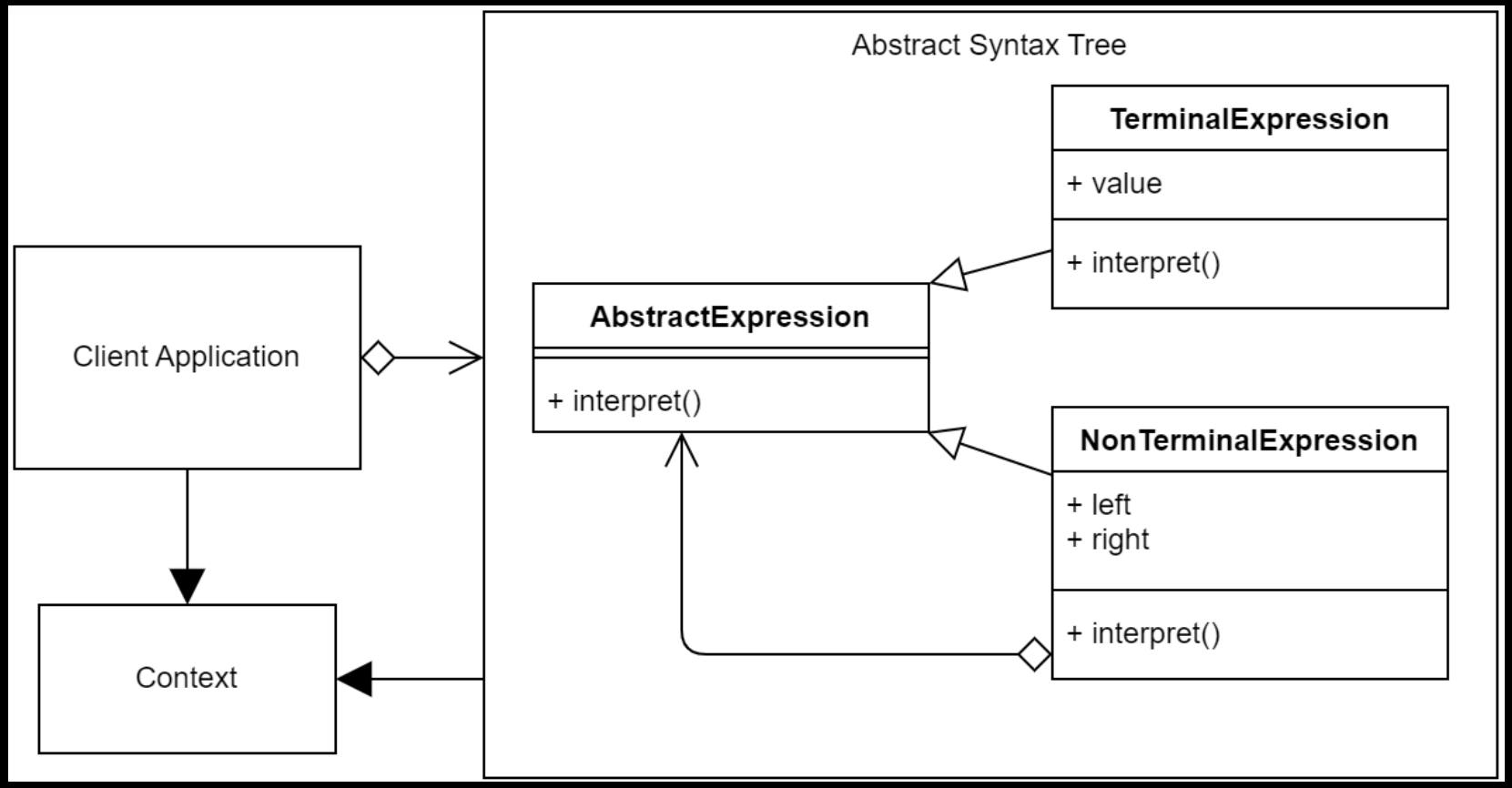
The Image below, is an AST for the expression: **5 + 4 – 3 + 7 - 2**



Terminology

- **Abstract Expression**: Describe the method(s) that Terminal and Non-Terminal expressions should implement.
- **Non-Terminal Expression**: A composite of Terminal and/or Non-Terminal expressions.
- **Terminal Expression**: A leaf node Expression.
- **Context**: Context is state that can be passed through interpret operations if necessary.
- **Client**: Builds or is given an Abstract Syntax Tree to interpret.

Interpreter UML Diagram



```python
class AbstractExpression():
    "All Terminal and Non-Terminal expressions will implement an 'interpret' method"
    @staticmethod
    def interpret():
        """
        The 'interpret' method gets called recursively for each
        AbstractExpression
        """

class Number(AbstractExpression):
    "Terminal Expression"

    def __init__(self, value):
        self.value = int(value)

    def interpret(self):
        return self.value

    def __repr__(self):
        return str(self.value)

class Add(AbstractExpression):
    "Non-Terminal Expression."

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() + self.right.interpret()

    def __repr__(self):
        return f"({self.left} Add {self.right})"
```

Code implementation

```
5 + 4 - 3 + 7 - 2
['5', '+', '4', '-', '3', '+', '7', '-', '2']
11
((((5 Add 4) Subtract 3) Add 7) Subtract 2)

Process finished with exit code 0
```

```python
class Subtract(AbstractExpression):
    "Non-Terminal Expression"

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() - self.right.interpret()

    def __repr__(self):
        return f"({self.left} Subtract {self.right})"

# The Client
# The sentence complies with a simple grammar of
# Number -> Operator -> Number -> etc,
SENTENCE = "5 + 4 - 3 + 7 - 2"
print(SENTENCE)

# Split the sentence into individual expressions that will be added to
# an Abstract Syntax Tree (AST) as Terminal and Non-Terminal expressions
TOKENS = SENTENCE.split(" ")
print(TOKENS)

# Manually Creating an Abstract Syntax Tree from the tokens
AST: list[AbstractExpression] = []  # Python 3.9
# AST = []  # Python 3.8 or earlier
AST.append(Add(Number(TOKENS[0]), Number(TOKENS[2])))  # 5 + 4
AST.append(Subtract(AST[0], Number(TOKENS[4])))        # ^ - 3
AST.append(Add(AST[1], Number(TOKENS[6])))             # ^ + 7
AST.append(Subtract(AST[2], Number(TOKENS[8])))        # ^ - 2

# Use the final AST row as the root node.
AST_ROOT = AST.pop()

# Interpret recursively through the full AST starting from the root.
print(AST_ROOT.interpret())

# Print out a representation of the AST_ROOT
print(AST_ROOT)
```