Design Patterns

Interpreter Pattern

not-matthias

Contents

1 Description

The interpreter pattern consists of different classes. There is one class for each symbol. These symbols can either be *terminal* or *nonterminal*.

1.1 Grammar

In order to understand the Interpreter Pattern, you also need to understand the basic concepts of grammar. A grammar is basically **the language of languages**. Every language must have a grammar to determine its structure. There's different notations and symbols used in a grammar. For example, there are multiple symbols in a grammar: *Nonterminal* and *terminal*. There are also different notations like: *Backus-Naur Form (BNF)* or *Extended Backus-Naur Form (EBNF)*.

1.1.1 Terminal Symbols

Terminal symbols cannot be replaced by production rules. All terminal symbols define an alphabet, of which sentences or words consist of.

To phrase it differently: Terminal symbols are basically the leaf nodes of a syntax tree. Here's an example of what this could look like:

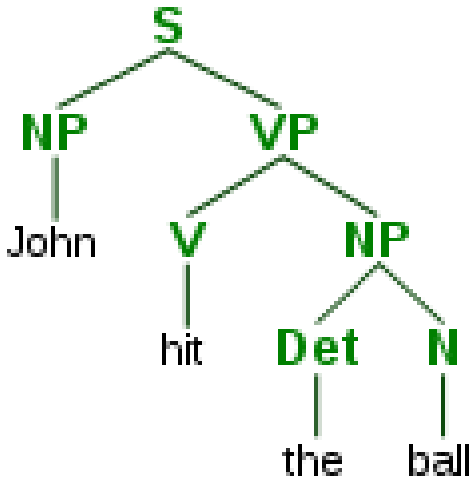


Figure 1.1: Parse Tree

The terminal symbols here are: *John*, *hit*, *the* and *ball*.

Terminal symbols are also often used for punctuation marks or keywords in programming languages. For example in C#, you could use: *for*, *while*, *var*, *if* and every other keyword.

1.1.2 Nonterminal Symbols

Nonterminal symbols are the opposite of terminal symbols. These are symbols which can be replaced. A formal grammar includes a *start symbol* (which is also a nonterminal symbol) and a set of nonterminal symbols. Then, with the production rules, every sentence in a language can be derived from them. Usually, terminal symbols will be used to replace the nonterminal ones.

Let's look at the example again:

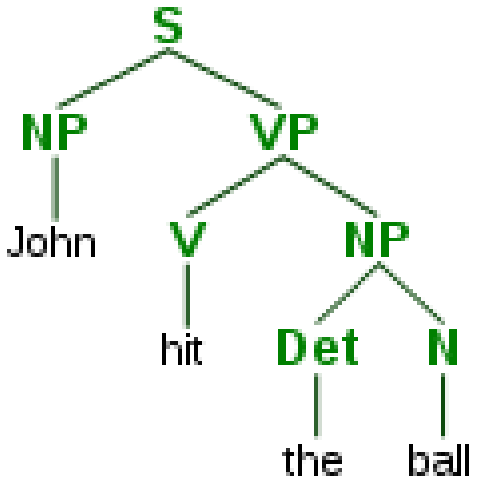


Figure 1.2: Parse Tree

The parse tree is the entire structure, starting from S and ending in each of the leaf nodes (John, hit, the, ball). The following abbreviations are used in the tree:

- S for sentence
- NP for noun phrase
- VP for verb phrase
- V for verb
- D for determiner
- N for noun

There is also a differentiation between root node, branch node and leaf node. A *root node*, is a node that doesn't any predecessors. In this example S is the root node. Within a sentence there can also only ever be one root node. A *branch node* is a parent node, that connects two or more child nodes, which can again be branch or leaf nodes. VP and NP are the branch nodes in this example. A *leaf node* does not have any other child nodes apart from the terminal symbols.

There are also multiple ways of visualizing this parse tree. See section 1.2 fore more information.

1.1.3 Production Rules

A grammar is defined by production rules, which specify, which symbols can be replaced with which symbols. Each rule has a *head*, or *left-hand-side*, and *body*, or *right-hand-side*.

Rules are often written in the $head \rightarrow body$ format. For example the rule $a \rightarrow b$ specifies that a can be replaced by b.

Example

Let's say you have your custom alphabet. You alphabet consists of the following:

- Nonterminal Symbols: A, B, C
- Terminal Symbols: D, E, F

So we defined the symbols, but we are missing something. We cannot define any sentences or words with this. This is were the production rules can be used. So we can for example define that the nonterminal symbol A can be replaced with the terminal symbols E and F. This can then also be done for every other nonterminal symbol and then you have your basic grammar.

```
A → E, F
B → D, E, F
C → F
```

1.2 Syntax Tree

Parse trees are also often known as *derivation trees* or *(concrete) syntax tree*. Parse trees is a tree-like data structure that represents a concrete representation of the input. Parse trees retain all of the information of the input. Here we again have a start symbol S, nonterminal and terminal symbols as mentioned in section 1.1.3.

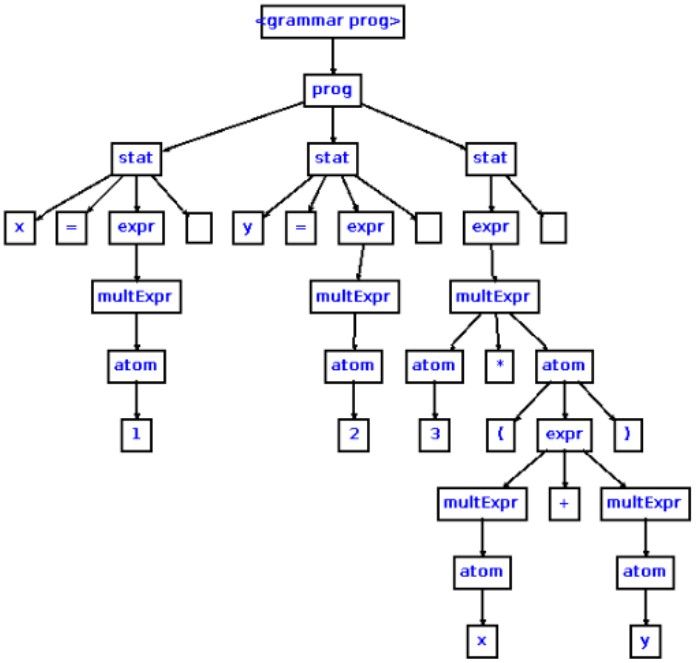


Figure 1.3: Parse Tree

As you can see, this parse tree contains all the meta data, which might not be needed. For example you could simply remove nonterminal symbols like *atom*, because they only have one terminal expression as child. This is where the Abstract Syntax Tree can be used.

1.3 Abstract Syntax Tree

The *Abstract Syntax Tree* (AST) is an abstract representation of the input. You'll notice that most of the nonterminal expressions are not present anymore, because unimportant translations have been removed. **In other words, an *Abstract Syntax Tree* highlights the structure of a language and not the grammar (like the syntax tree)**.

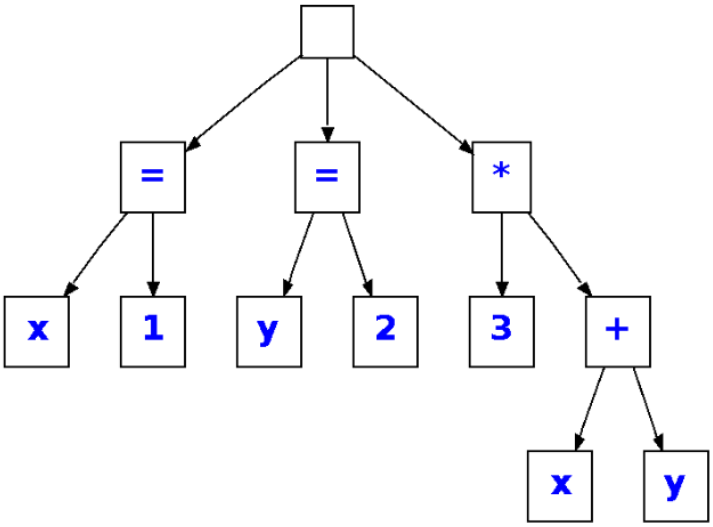


Figure 1.4: AST

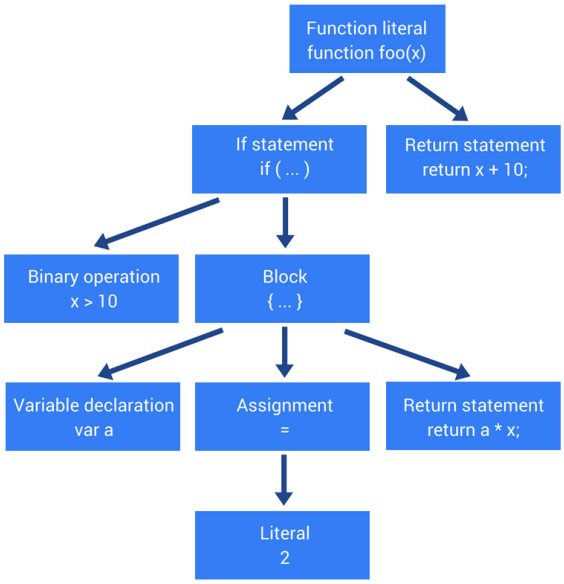
When you compare it to the parse tree, you can already see, that it's much easier to understand. Let's look at another example. You can also parse source code, like this simple JavaScript function, which just does a mathematical calculation based on the input.

```
function foo(x) {
    if (x > 10) {
        var a = 2;
        return a * x;
```

```
    }

    return x + 10;
}
```

The resulting *Abstract Syntax Tree* could look like this.



This AST has been simplified for visualization purposes. The actual AST would be much more complex and contain more data. There's a cool project, where you can show the actual AST of a JavaScript program: https://astexplorer.net/

1.3.1 Usage in Compilers

Abstract Syntax Trees are also often used, to represent the structure of the program code in various computer languages. An AST is usually the result of the analysis of the source code by the compiler. It does also often serve as an intermediate representation of the program, which is needed for the compiler.

For example the compiler LLVM uses their own intermediate language (or AST), which can then be used by other programs. This brings the big advantage, that other programs can convert their AST, to be able to compile their code. Renowned programming languages that use LLVM as their backend are RustLang, C++, C and even Java.

Motivation

An AST has several properties that aid the further steps of the compilation process:

- An AST can be edited and enhanced with information such as properties or annotations.
  - This cannot be done with source code, because then it would need to be changed.
- An AST doesn't include punctuation and delimiters (braces, semicolons, parentheses, ...), because they are not needed.

During the first stage, the syntax analysis, the compiler produces a parse tree. This parse tree will then be converted to an AST. This will almost always lead to a more efficient compiler. An AST has also the advantage that it has a smaller height and smaller number of elements.

Design

The design of the *Abstract Syntax Tree* is also a crucial part of the compiler design. This design is often closely linked with the design of a compiler and its expected features.

Core requirements must include the following:

- Variable types must be preserved, as well as the location of each declaration in source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

These items can help designing the data structure of the AST. You also want to support the verification of the source code via the compiler. This can be done by unparsing the AST into source code form. The resulting source code should be similar to the original.

1.4 Backus–Naur form

This is a notation technique, that is often used to describe the syntax of languages.

A BNF specification is a set of derivation rules, which can be written like this:

```
<symbol> ::= __expression__
```

Here the *symbol* is *nonterminal* and the *expression* consists of one or more *terminal symbols*. The *::=* means that the symbol on the left must be replaced with one of the symbols on the right. The BNF also uses different production rules to map the nonterminal to the terminal symbols. This has already been discussed in section 1.1.1 and 1.1.2.

1.4.1 Example

For example, this is how a simple terminal symbol could look like. A digit without zero, can either be 1, 2, 3 and so on.

```
<Digit without Zero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

You can also use a sequence, with multiple terminal and nonterminal symbols:

```
<Digit>              ::= 0 | <Digit without Zero>
<Twodigit Number>    ::= <Digit without Zero> <Digit>
<Ten to Nineteen>    ::= 1 <Digit>
<Fourtytwo>          ::= 42
```

Here *Digit* can either be 0 or a digit without zero. If you don't specify a separator, it'll be concatenated.

1.5 Reverse Polish Notation

This is another notation, which is also known as *polish postfix notation*. It's a mathematical notation where the operators follow their operands.

For example, a normal addition looks like this: **3 + 4**. In the polish postfix notation, you move the operator to the end. This will look like this: **3 4 +**. If you take a more advanced example like **3 - 4 + 5**, this can be written as **3 4 - 5 +**. 4 is first subtracted from 3, then 5 is added to it.

However, there is one big advantage: the *Reverse Polish Notation* removes the need for parentheses. While **3 - 4 × 5** can also be written as **3 - (4 × 5)**, that means something quite different than **(3 - 4) × 5**. In *Reverse Polish Notation*, the former could be written **3 4 5 -** × and the latter **3 4 - 5** ×.

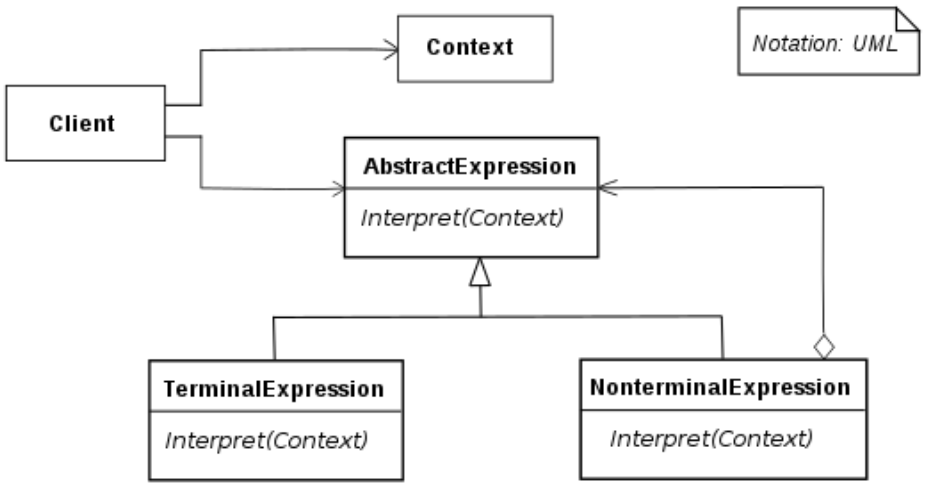
2 Purpose

2.1 When should it be used?

You should not use the interpreter pattern for every problem. It has, like the other design patterns, its use cases. However, you should always use it when:

- there's a language to interpret and you can represent the statements as an AST.
- the grammar is simple.
  - In large projects, the class hierarchy becomes large and unmaintainable.
  - Parser are a great alternative in such cases.
  - They don't use an AST, which can save space and possibly time.
- efficiency is not a critical concern.
  - It's usually more efficient, when translating the parse tree to another form.
  - For example, regular expressions are often transformed into state machines.
  - The interpreter pattern could there still be used to translate it.

3 UML



These are the classes that are represented in the UML class diagram:

- *AbstractExpression*: Declares an interpret() operation that all nodes (terminal and nonterminal) in the AST override.
- *TerminalExpression* (*NumberExpression*): Implements the interpret() operation for terminal expressions
- *NonTerminalExpression* (*PlusExpression, MinusExpression, DivideExpression, MultiplyExpression*): Implements the interpret() operation for all nonterminal expressions.

- *Context(String)*: Contains information that is global to the interpreter. It contains the String expression with the Postfix notation that has to be interpreted and parsed.

- *Client* : Builds the AST assembled from TerminalExpression and NonTerminal-Expression. The Client invokes the interpret() operation.

The nonterminal symbols forward interpret the *interpret* methods. This means, that you only need to call it once for the root node, because they will automatically call it on their list of child nodes. The *TerminalExpression* cannot have any children. Only the *NonTerminalExpression* can have one or more children. To see the interactions between the different object, look at the example in the next chapter.

4 Example

This example will cover a calculator that can parse mathematical expressions and print the result. The resulting code for this, can be found in the *examples* folder.

4.1 Define the classes

The first step should be the specify, which nonterminal and terminal symbols are needed. In our case, we have the following symbols:

- Nonterminal
  - Plus
  - Minus
  - Division
  - Multiplication
- Terminal
  - Number

The nonterminal symbols need, of course, two child nodes. This could for example be a terminal but also nonterminal symbol. With these symbols, we can basically define a simple syntax tree.

4.2 Implement the classes

Now that we have defined the classes, we can implement them. As already mentioned in the last chapter, each class derives from an abstract class, that has the *interpret* method.

```
package expressions;

public abstract class AbstractExpression {
    public abstract int interpret();

    @Override
    public abstract String toString();
}
```

You can see that this class also overrides the *toString* method. You'll see later why. The different expressions then derive the *AbstractExpression* class and implement the *interpret* method. This could look like this:

4.2.1 Nonterminal Expressions

```
package expressions;

import lombok.AllArgsConstructor;
```

```
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@AllArgsConstructor
public class MultiplyExpression extends AbstractExpression {
    private AbstractExpression lhs;
    private AbstractExpression rhs;

    @Override
    public int interpret() {
        return lhs.interpret() * rhs.interpret();
    }

    @Override
    public String toString() {
        return String.format("(%s * %s)", lhs.toString(), rhs.toString());
    }
}
```

I should probably mention, that I used *Lombok* to automatically generate the getter and setters for me so I don't have to write that much code and keep it simple. The only difference between the different nonterminal symbols is, that they have a different *interpret* method. For example the *PlusExpression* will just use the + instead of the * operator. As you have seen in the UML diagram, the nonterminal expression forward interprets the terminal expressions that it stores internally.

You can also see that I implemented the overridden *toString* method, which will later be used to print our custom parse tree. It just takes the left and right side of the

mathematical expression and prints it. Just like you would write it on paper.

4.2.2 Terminal Expressions

This is basically the same as the nonterminal expression above, with the only difference, that it does not have child nodes. It has only a simple number, which cannot be replaced.

```
package expressions;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
@AllArgsConstructor
public class NumberExpression extends AbstractExpression {
    private int number;

    public NumberExpression(String s) {
        this.number = Integer.parseInt(s);
    }

    @Override
    public int interpret() {
        return number;
    }
```

```
    @Override
    public String toString() {
        return String.format("%s", number);
    }
}
```

4.3 Define an input format

I decided to use the Reverse Polish Notation as input format. This remove the need of parentheses, which makes parsing the input easier. The input format for our calculator can also be described in the Backus–Naur form, which will look like this:

```
expression  ::= plus | minus | variable | number
plus        ::= expression expression '+'
minus       ::= expression expression '-'
variable    ::= 'a' | 'b' | 'c' | ... | 'z'
digit       = '0' | '1' | ... | '9'
number      ::= digit | digit number
```

This defines a language that contains Reverse Polish Notation like these:

```
a b +
a b c + -
a b + c a - -
```

4.4 Parse the input

The input can be parsed, by first splitting the string and then finding the correct expression for the token. Using a stack makes parsing the Reverse Polish Notation input really easy.

```
private static AbstractExpression createAbstractSyntaxTree (String tokenString) {
    var tokenList = tokenString.split(" ");
    var stack = new Stack<AbstractExpression>();

    // Iterate though the token list.
    for (var token : tokenList) {
        if (isOperator(token)) {
            var rhs = stack.pop();
            var lhs = stack.pop();

            var operator = getOperator(token, lhs, rhs);

            stack.push(operator);
        } else {
            var number = new NumberExpression(token);
            stack.push(number);
        }
    }

    return stack.pop();
}
```

4.5 Result

When you use all the functions, you can add the input string like **4 3 2 - 1 + *** in this example, generate the Abstract Syntax Tree and then either print it or call *interpret*. The *toString* method also forward calls *toString*, thus, if we print it, the entire Abstract Syntax Tree will get printed to the console. The output will be *(4 * ((3 - 2) + 1)) = 8* and *((12 * 2) + (64 / 2)) = 56*

```
public static void main(String[] args) {
    //
    // Create expression tree from string
    //
    var tokenString = "4 3 2 - 1 + *";
    var result = createAbstractSyntaxTree(tokenString);
    System.out.println(String.format("%s = %s",
        result, result.interpret()));

    //
    // Create expression tree manually
    //
    var customExpression = new PlusExpression(
            new MultiplyExpression(new NumberExpression(12),
                new NumberExpression(2)),
            new DivideExpression(new NumberExpression(64),
                new NumberExpression(2))
    );
    System.out.println(String.format("%s = %s",
        customExpression, customExpression.interpret()));
}
```

5 Usages

5.1 Frameworks

The interpreter pattern is also sometimes used in big frameworks.

- java.util.Pattern
- java.text.Normalizer
- javax.el.ELResolver
- All subclasses of java.text.Format
- Spring Expression Language (SpEL)
  - The developers of the Spring Framework used the Interpreter Pattern when they added the feature of SpEL. This is an ideal use case for the Interpreter Pattern.
  - For example there's a *ExpressionParser* which is an interface that parses expression strings into compiled expressions that can be evaluated. This implementation is responsible for parsing an input and returning instances of *Expression*.
  - These are some expressions: *CompositeStringExpression*, *LiteralExpression*, *SpelExpression*

  - Spring also provides the *EvaluationContext* interface, with the implementations *MethodBasedEvaluationContext* and *StandardEvaluationContext* classes to represent the context component of the Interpreter pattern.

5.2 General

The interpreter pattern can generally be used for a lot of things. Like already noted in section 2.1, it can be used for the following:

- Code Formatters
- Extensions for IDEs
- Database query languages (e.g. SQL)
  - Here's an example, where the interpreter pattern was used to implement a simple SQL-like language: Baeldung - Java Interpreter Pattern
- Computer languages