Interpreter

Kurt de Leon & Brian Guadalupe

April 19, 2018

Introduction

What is this?

What is this?

From Gamma, et al.

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

What is this?

From Gamma, et al.

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

- A behavioral pattern

What is this?

From Gamma, et al.

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

- A behavioral pattern
- Unfortunately, also ignored by programmers :(

When to use this?

The Interpreter design pattern works best when

- the grammar is simple
- efficiency is **not** a concern

When to use this?

The Interpreter design pattern works best when

- the grammar is simple
- efficiency is **not** a concern

Basically...

Use the Interpreter design pattern when you have **a language to interpret**.

Motivation

Consider this problem...

- Suppose you want to create your own personal assistant (maybe for your smartphone, laptop, fridge, etc.)
- Often takes a query written in plain English as input

Some things you can ask me:
How long do beagles live?
Play Monument Valley
Find Disney films
Enable Wi-Fi
When is my wife's birthday?
How's the weather this week?

Consider this problem...

- Suppose you want to create your own personal assistant (maybe for your smartphone, laptop, fridge, etc.)
- Often takes a query written in plain English as input
- How do you process such queries?

Some things you can ask me:

How long do beagles live?

Play Monument Valley

Find Disney films

Enable Wi-Fi

When is my wife's birthday?

How's the weather this week?

Consider this problem...

- Suppose you want to create your own personal assistant (maybe for your smartphone, laptop, fridge, etc.)
- Often takes a query written in plain English as input
- How do you process such queries?

Take note!

For now, only simple calculations are supported.

Some things you can ask me:
How long do beagles live?
Play Monument Valley
Find Disney films
Enable Wi-Fi
When is my wife's birthday?
How's the weather this week?

Defining the language

We can define the language recursively, as follows:

Defining the language

We can define the language recursively, as follows:

Backus-Naur form

$$\langle \textit{expression} \rangle ::= \langle \textit{plus} \rangle \mid \langle \textit{minus} \rangle \mid \langle \textit{integer} \rangle$$

$$\langle \textit{plus} \rangle ::= \langle \textit{expression} \rangle \text{ 'plus' } \langle \textit{expression} \rangle$$

$$\langle \textit{minus} \rangle ::= \langle \textit{expression} \rangle \text{ 'minus' } \langle \textit{expression} \rangle$$

$$\langle \textit{integer} \rangle ::= \langle \textit{unsigned-integer} \rangle \mid \text{'}+\text{'} \langle \textit{unsigned-integer} \rangle \mid \text{'}-\text{'} \langle \textit{unsigned-integer} \rangle$$

$$\langle \textit{unsigned-integer} \rangle ::= \langle \textit{digit} \rangle \mid \langle \textit{unsigned-integer} \rangle \langle \textit{digit} \rangle$$

$$\langle \textit{digit} \rangle ::= \text{'0'} \mid \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'}$$

Defining the language

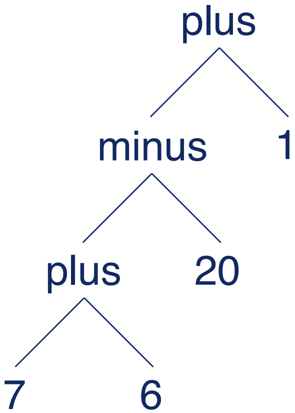


Figure 1: The abstract syntax tree for the query “7 plus 6 minus 20 plus 1”

Consequences

Consequences

Using the Interpreter pattern can lead into the following pros and cons:

Consequences

Using the Interpreter pattern can lead into the following pros and cons:

1. Easy to change and extend the grammar
    - Through inheritance

Consequences

Using the Interpreter pattern can lead into the following pros and cons:

1. Easy to change and extend the grammar
    - Through inheritance
2. Implementing the grammar is also easy
    - Classes defining nodes in the abstract syntax tree have similar implementations

Consequences

Using the Interpreter pattern can lead into the following pros and cons:

1. Easy to change and extend the grammar
   - Through inheritance
2. Implementing the grammar is also easy
   - Classes defining nodes in the abstract syntax tree have similar implementations
3. Complex grammars are hard to maintain
   - Because it defines at least one class for every rule in the grammar
   - Other techniques may be more appropriate to deal with complex grammars

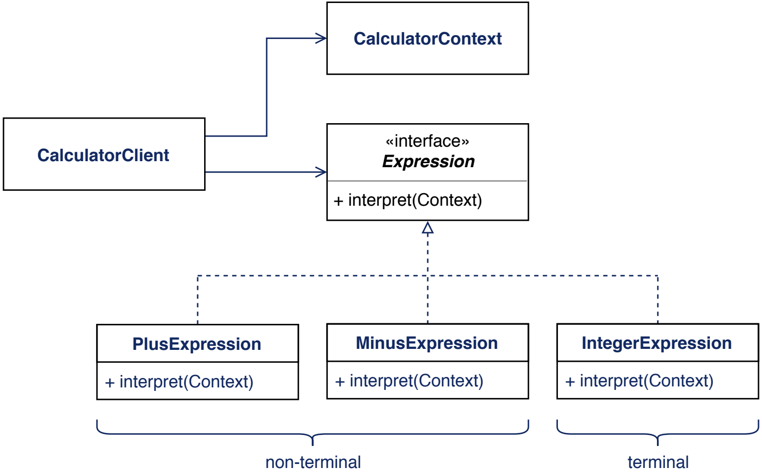
Consequences

Using the Interpreter pattern can lead into the following pros and cons:

1. Easy to change and extend the grammar
   - Through inheritance
2. Implementing the grammar is also easy
   - Classes defining nodes in the abstract syntax tree have similar implementations
3. Complex grammars are hard to maintain
   - Because it defines at least one class for every rule in the grammar
   - Other techniques may be more appropriate to deal with complex grammars
4. Support new ways to interpret expressions
   - Add pretty-printing and type checking functionality, for example
   - Consider using the Visitor pattern if you intend to provide more new ways to interpret an expression

Implementation

Structure


CalculatorContext
CalculatorClient
«interface»
Expression
+ interpret(Context)
PlusExpression
+ interpret(Context)
MinusExpression
+ interpret(Context)
IntegerExpression
+ interpret(Context)
non-terminal
terminal

Participants

1. CalculatorClient
   - builds an abstract syntax tree of the expression
   - invokes the `interpret()` method
2. CalculatorContext
   - contains information that is global to the interpreter (e.g., variables, current state)
3. Expression
   - declares an `interpret()` method that is common to all nodes in the AST
4. PlusExpression, MinusExpression
   - non-terminal expressions
5. IntegerExpression
   - terminal expression

CalculatorClient

```java
import java.util.Scanner;

public class CalculatorClient {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (true) {
            // get expr from standard input
            System.out.print("Enter an expression: ");
            String expr = sc.nextLine();
            // build the abstract syntax tree
            Expression ast = parse(expr);
            // interpret the AST
            String result = Integer.toString(ast.interpret());
            // output the result
            System.out.println("Result: " + result);
        }
    }

    // Note that the Interpreter pattern DOES NOT supply a parser of its own!
    public static Expression parse(String inputExpr) {
        // do stuff here
    }
}
```

CalculatorContext

```java
public class CalculatorContext {
    /*
     * The CalculatorContext class contains information that is global
     * to the interpreter.
     * For example, if we want to extend our simple calculator in order
     * to support variables, we could put a HashMap of variables, along
     * with theirrespective values, so that the interpreter can either
     * recall the value of a variable or store a new value to it.
     */
}
```

Expression

```java
public interface Expression {
    public int interpret();
}
```

PlusExpression

```java
public class PlusExpression implements Expression {

    private Expression leftExpr, rightExpr;

    public PlusExpression(Expression left, Expression right) {
        this.leftExpr = left;
        this.rightExpr = right;
    }

    @Override
    public int interpret() {
        return leftExpr.interpret() + rightExpr.interpret();
    }
}
```

MinusExpression

```java
public class MinusExpression implements Expression {

    private Expression leftExpr, rightExpr;

    public MinusExpression(Expression left, Expression right) {
        this.leftExpr = left;
        this.rightExpr = right;
    }

    @Override
    public int interpret() {
        return leftExpr.interpret() - rightExpr.interpret();
    }
}
```

IntegerExpression

```java
public class IntegerExpression implements Expression {

    private int number;

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

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

But there's a catch...

- The Gang of Four book did not address the issue of parsing the input (i.e. converting the language into an AST), because they said it's a separate problem altogether
- Very difficult to find an easy way to generate ASTs
- Without parsing, the Interpreter pattern is essentially useless!

Demo

References

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
  Design Patterns: Elements of Reusable Object-Oriented Software.
  Addison-Wesley, Reading, MA, 1995.

- Jeffrey Kegler.
  The Interpreter Design Pattern, March 2013.
  Available at `https://jeffreykegler.github.io/Ocean-of-Awareness-blog/individual/2013/03/interpreter.html`.