
INTERPRETER

Class Behavioral

Intent

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

Motivation

If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language. Then you can build an interpreter that solves the problem by interpreting these sentences.

For example, searching for strings that match a pattern is a common problem. Regular expressions are a standard language for specifying patterns of strings. Rather than building custom algorithms to match each pattern against strings, search algorithms could interpret a regular expression that specifies a set of strings to match.

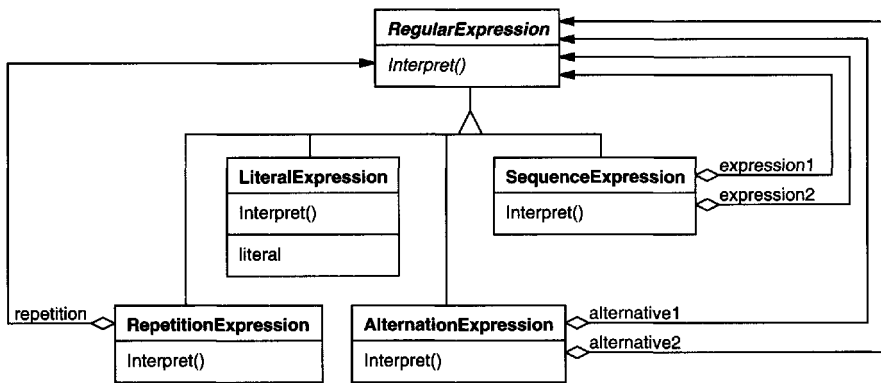
The Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences. In this example, the pattern describes how to define a grammar for regular expressions, represent a particular regular expression, and how to interpret that regular expression.

Suppose the following grammar defines the regular expressions:

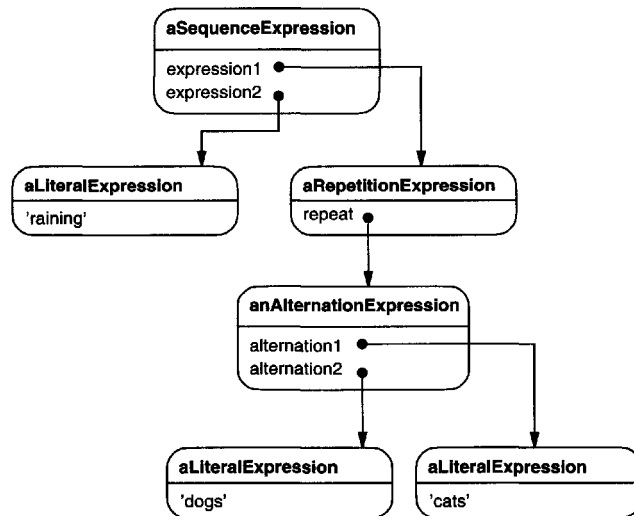
```
expression ::= literal | alternation | sequence | repetition |
              '(' expression ')'
alternation ::= expression '|' expression
sequence  ::= expression '&' expression
repetition ::= expression '**'
literal   ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

The symbol `expression` is the start symbol, and `literal` is a terminal symbol defining simple words

The Interpreter pattern uses a class to represent each grammar rule. Symbols on the right-hand side of the rule are instance variables of these classes. The grammar above is represented by five classes: an abstract class `RegularExpression` and its four subclasses `LiteralExpression`, `AlternationExpression`, `SequenceExpression`, and `RepetitionExpression`. The last three classes define variables that hold subexpressions.



Every regular expression defined by this grammar is represented by an abstract syntax tree made up of instances of these classes. For example, the abstract syntax tree



represents the regular expression

```
raining & (dogs | cats) *
```

We can create an interpreter for these regular expressions by defining the `Interpret` operation on each subclass of **RegularExpression**. `Interpret` takes as an argument the context in which to interpret the expression. The context contains the input string and information on how much of it has been matched so far. Each subclass of **RegularExpression** implements `Interpret` to match the next part of the input string based on the current context. For example,

- LiteralExpression will check if the input matches the literal it defines,
- AlternationExpression will check if the input matches any of its alternatives,
- RepetitionExpression will check if the input has multiple copies of expression it repeats,

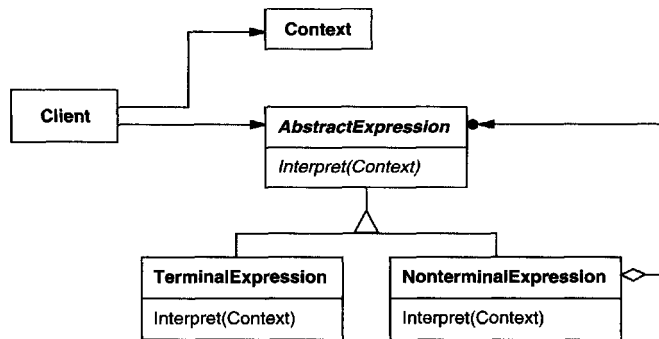
and so on.

Applicability

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when

- the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases. They can interpret expressions without building abstract syntax trees, which can save space and possibly time.
- efficiency is not a critical concern. The most efficient interpreters are usually *not* implemented by interpreting parse trees directly but by first translating them into another form. For example, regular expressions are often transformed into state machines. But even then, the *translator* can be implemented by the Interpreter pattern, so the pattern is still applicable.

Structure



Participants

- **AbstractExpression** (RegularExpression)
 - declares an abstract `Interpret` operation that is common to all nodes in the abstract syntax tree.

- **TerminalExpression** (LiteralExpression)
 - implements an Interpret operation associated with terminal symbols in the grammar.
 - an instance is required for every terminal symbol in a sentence.
- **NonterminalExpression** (AlternationExpression, RepetitionExpression, SequenceExpressions)
 - one such class is required for every rule $R ::= R_1 R_2 \dots R_n$ in the grammar.
 - maintains instance variables of type AbstractExpression for each of the symbols R_1 through R_n .
 - implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R_1 through R_n .
- **Context**
 - contains information that's global to the interpreter.
- **Client**
 - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.
 - invokes the Interpret operation.

Collaborations

- The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.
- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression defines the base case in the recursion.
- The Interpret operations at each node use the context to store and access the state of the interpreter.

Consequences

The Interpreter pattern has the following benefits and liabilities:

1. *It's easy to change and extend the grammar.* Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.

2. *Implementing the grammar is easy, too.* Classes defining nodes in the abstract syntax tree have similar implementations. These classes are easy to write, and often their generation can be automated with a compiler or parser generator.
3. *Complex grammars are hard to maintain.* The Interpreter pattern defines at least one class for every rule in the grammar (grammar rules defined using BNF may require multiple classes). Hence grammars containing many rules can be hard to manage and maintain. Other design patterns can be applied to mitigate the problem (see Implementation). But when the grammar is very complex, other techniques such as parser or compiler generators are more appropriate.
4. *Adding new ways to interpret expressions.* The Interpreter pattern makes it easier to evaluate an expression in a new way. For example, you can support pretty printing or type-checking an expression by defining a new operation on the expression classes. If you keep creating new ways of interpreting an expression, then consider using the Visitor (331) pattern to avoid changing the grammar classes.

[REDACTED]

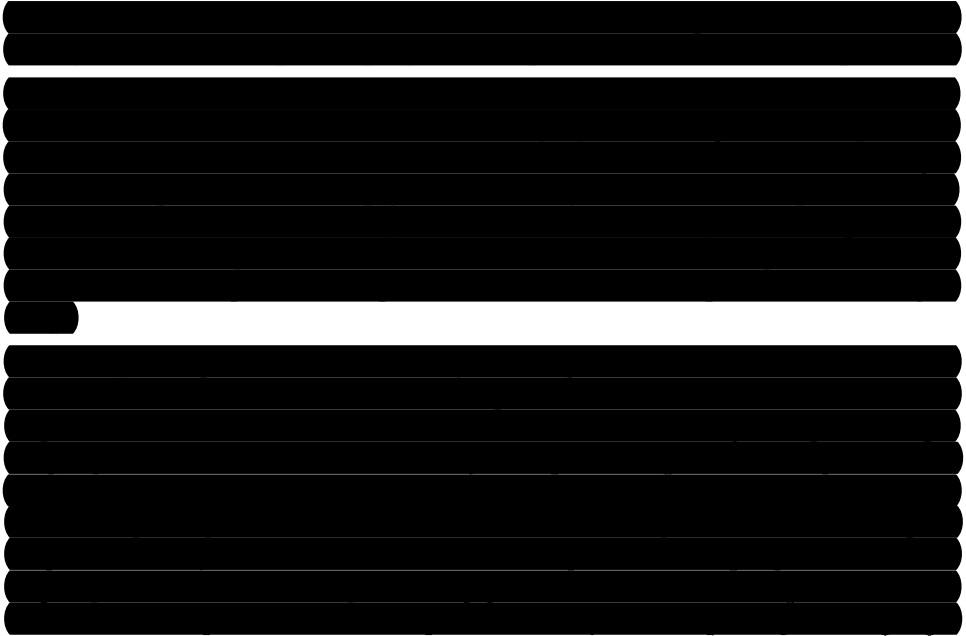
[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]



Known Uses

The Interpreter pattern is widely used in compilers implemented with object-oriented languages, as the Smalltalk compilers are. SPECTalk uses the pattern to interpret descriptions of input file formats [Sza92]. The QOCA constraint-solving toolkit uses it to evaluate constraints [HHMV92].

Considered in its most general form (i.e., an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern. But the Interpreter pattern should be reserved for those cases in which you want to think of the class hierarchy as defining a language.

Related Patterns

Composite (163): The abstract syntax tree is an instance of the Composite pattern. Flyweight (195) shows how to share terminal symbols within the abstract syntax tree.

Iterator (257): The interpreter can use an Iterator to traverse the structure.

Visitor (331) can be used to maintain the behavior in each node in the abstract syntax tree in one class.