**Interpreter（解释器）**

意图

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

动机

如果一种特定类型的问题发生的概率足够高，那么可能就值得将该问题的各个实例表述为一个简单语言中的句子。这样就可以构建一个解释器，该解释器通过解释这些句子来解决该问题。

例如，搜索匹配一个模式的字符串是一个常见问题。正则表达式是描述字符串模式的一种标准语言。与其为每一个模式都构造一个特定的算法，不如使用一种通用的搜索算法来解释执行一个正则表达式，该正则表达式定义了待匹配字符串的集合。

解释器模式描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。在上面的例子中，本设计模式描述了如何为正则表达式定义一个文法，如何表示一个特定的正则表达式，以及如何解释这个正则表达式。

考虑以下文法定义正则表达式：

expression ::= literal | alternation | sequence | repetition | ‘(’ expression ‘)’

alternation ::= expression ‘|’ expression

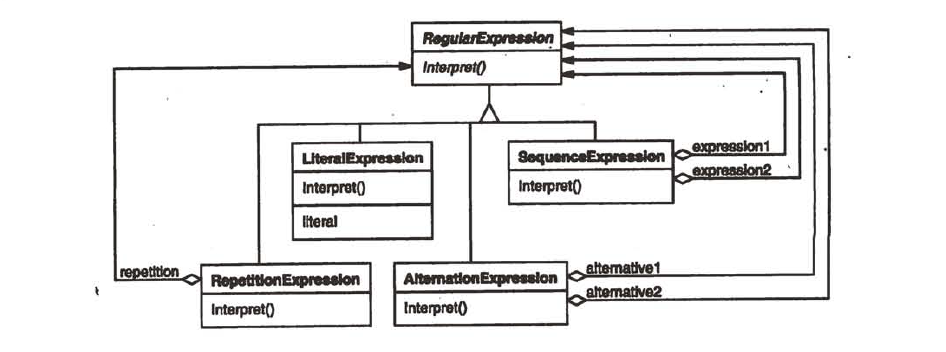
sequence ::= expression ‘&’ expression

repetition ::= expression ‘\*’

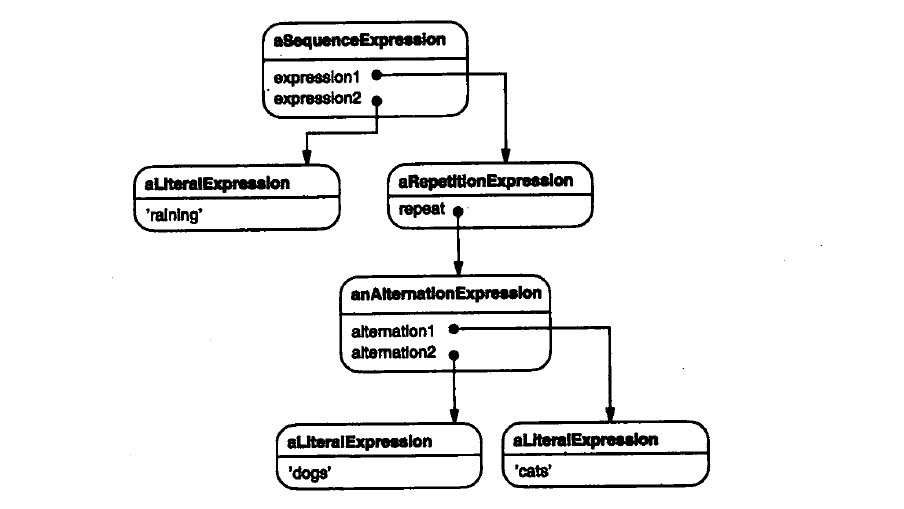
literal ::= ‘a’|’b’|’c’|..

符号expression是开始符号，literal是定义简单字的终结符。

解释器模式使用类来表示每一条文法规则。在规则右边的符号是这些类的实例变量。上面的文法用五个类表示：一个抽象类RegularExpression和它四个子类LiteralExpression、AlternationExpression、SequenceExpression和RepetitionExpression后三个类定义的变量代表子表达式。



每个用这个文法定义的正则表达式都被表示为一个由这些类的实例构成的抽象语法树。



表示正则表达式：

Raining & (dogs | cats) \*

如果我们为RegularExpression的每一个子类都定义解释操作，那么就得到了为这些正则表达式的一个解释器。解释器将该表达式的上下文作为一个参数。上下文包含输入字符串和关于目前它已有多少已经被匹配等信息。为匹配输入字符串的下一部分，每一个RegularExpression的子类都在当前上下文的基础上实现解释操作。例如，

LiterExpression将检查输入是否匹配它定义的字（Liter）

AlternationExpression将检查输入事发后匹配它的任意一个选择项

RepetitionExpression将检查输入是否含有多个它所重复的表达式

结构图

