34

INTERPRETER

This pattern was previously described in GoF95.

DESCRIPTION

In general, languages are made up of a set of grammar rules. Different sentences can be constructed by following these grammar rules. Sometimes an application may need to process repeated occurrences of similar requests that are a combination of a set of grammar rules. These requests are distinct but are similar in the sense that they are all composed using the same set of rules. A simple example of this sort would be the set of different arithmetic expressions submitted to a calculator program. Though each such expression is different, they are all constructed using the basic rules that make up the grammar for the language of arithmetic expressions.

In such cases, instead of treating every distinct combination of rules as a separate case, it may be beneficial for the application to have the ability to interpret a generic combination of rules. The Interpreter pattern can be used to design this ability in an application so that other applications and users can specify operations using a simple language defined by a set of grammar rules.

Applying the Interpreter pattern:

- A class hierarchy can be designed to represent the set of grammar rules with every class in the hierarchy representing a separate grammar rule.
- An `Interpreter` module can be designed to interpret the sentences constructed using the class hierarchy designed above and carry out the necessary operations.

Because a different class represents every grammar rule, the number of classes increases with the number of grammar rules. A language with extensive, complex grammar rules requires a large number of classes. The Interpreter pattern works best when the grammar is simple. Having a simple grammar avoids the need to have many classes corresponding to the complex set of rules involved, which are hard to manage and maintain.

EXAMPLE

Let us build a calculator application that evaluates a given arithmetic expression. For simplicity, let us consider only add, multiply and subtract operations. Instead of designing a custom algorithm for evaluating each arithmetic expression, the application could benefit from interpreting a generic arithmetic expression. The Interpreter pattern can be used to design the ability to understand a generic arithmetic expression and evaluate it.

The Interpreter pattern can be applied in two stages:

1. Define a representation for the set of rules that make up the grammar for arithmetic expressions.
2. Design an interpreter that makes use of the classes that represent different arithmetic grammar rules to understand and evaluate a given arithmetic expression.

The set of rules in Table 34.1 constitutes the grammar for arithmetic expressions.

Table 34.1 Grammar Rules for Arithmetic Expressions

Arithmetic Expressions – Grammar
`ArithmeticExpression::= ConstantExpression | AddExpression | MultiplyExpression | SubtractExpression`
`ConstantExpression::= Integer/Double Value`
`AddExpression::= ArithmeticExpression '+' ArithmeticExpression`
`MultiplyExpression::= ArithmeticExpression '*' ArithmeticExpression`
`SubtractExpression::= ArithmeticExpression '-' ArithmeticExpression`

From Table 34.1, it can be observed that arithmetic expressions are of two types — individual (e.g., `ConstantExpression`) or composite (e.g., `AddExpression`). These expressions can be arranged in the form of a tree structure, with composite expressions as nonterminal nodes and individual expressions as terminal nodes of the tree.

Let us define a class hierarchy as Figure 34.1 to represent the set of arithmetic grammar rules.

Each of the classes representing different rules implements the common `Expression` interface and provides implementation for the `evaluate` method (Listing 34.1 through Listing 34.5).

The `Context` is a common information repository that stores the values of different variables (Listing 34.6). For simplicity, values are hard-coded for variables in this example.

While each of the `NonTerminalExpression` classes performs the arithmetic operation it represents, the `TerminalExpression` class simply looks up the value of the variable it represents from the `Context`.

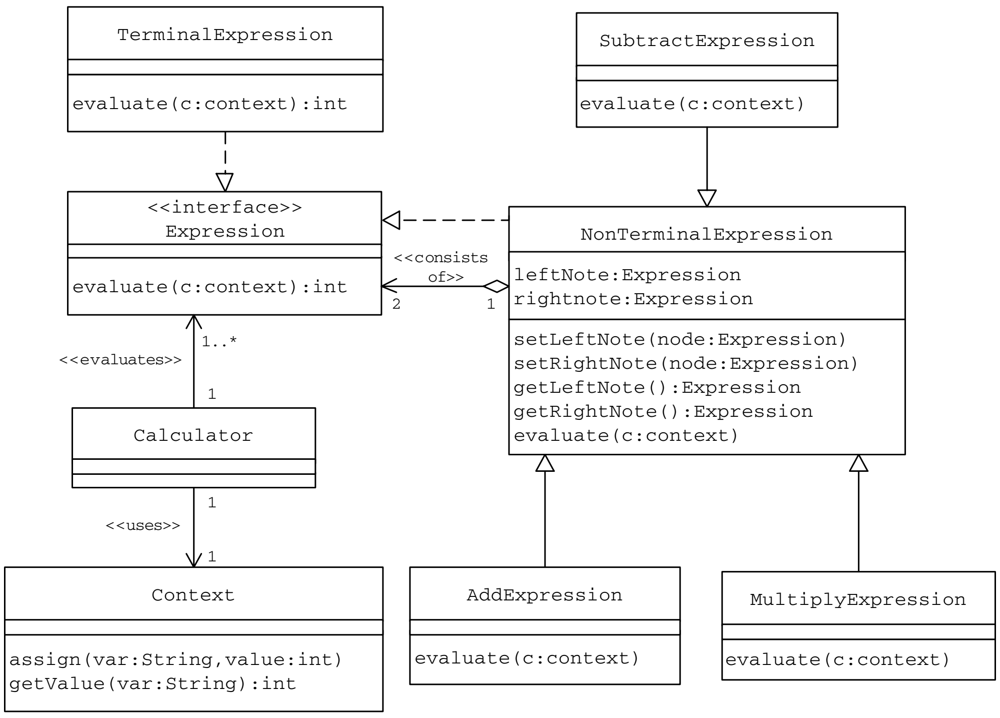

Figure 34.1 Class Hierarchy Representing Grammar Rules for Arithmetic Expressions

Listing 34.1 `Expression` Interface

```
public interface Expression {
  public int evaluate(Context c);
}
```

```
public class TerminalExpression implements Expression {
  private String var;
  public TerminalExpression(String v) {
    var = v;
  }
  public int evaluate(Context c) {
    return c.getValue(var);
  }
}
```

The application design can evaluate any expression. But for simplicity, the main `Calculator` (Listing 34.7) object uses a hard-coded arithmetic expression (a + b) * (c – d) as the expression to be interpreted and evaluated.

Listing 34.2 `NonTerminalExpression` Class

```
public abstract class NonTerminalExpression
  implements Expression {
  private Expression leftNode;
  private Expression rightNode;
  public NonTerminalExpression(Expression l, Expression r) {
    setLeftNode(l);
    setRightNode(r);
  }
  public void setLeftNode(Expression node) {
    leftNode = node;
  }
  public void setRightNode(Expression node) {
    rightNode = node;
  }
  public Expression getLeftNode() {
    return leftNode;
  }
  public Expression getRightNode() {
    return rightNode;
  }
}//NonTerminalExpression
```

Listing 34.3 `AddExpression` Class

```
class AddExpression extends NonTerminalExpression {
  public int evaluate(Context c) {
    return getLeftNode().evaluate(c) +
           getRightNode().evaluate(c);
  }
  public AddExpression(Expression l, Expression r) {
    super(l, r);
  }
}//AddExpression
```

The `Calculator` object carries out the interpretation and evaluation of the input expression in three stages:

Listing 34.4 `SubtractExpression` Class

```
class SubtractExpression extends NonTerminalExpression {
  public int evaluate(Context c) {
    return getLeftNode().evaluate(c) -
           getRightNode().evaluate(c);
  }
  public SubtractExpression(Expression l, Expression r) {
    super(l, r);
  }
}//SubtractExpression
```

Listing 34.5 `MultiplyExpression` Class

```
class MultiplyExpression extends NonTerminalExpression {
  public int evaluate(Context c) {
    return getLeftNode().evaluate(c) *
           getRightNode().evaluate(c);
  }
  public MultiplyExpression(Expression l, Expression r) {
    super(l, r);
  }
}//MultiplyExpression
```

1. *Infix-to-postfix conversion* — The input infix expression is first translated into an equivalent postfix expression.
2. *Construction of the tree structure* — The postfix expression is then scanned to build a tree structure.
3. *Postorder traversal of the tree* — The tree is then postorder traversed for evaluating the expression.

```
public class Calculator {
                         …
                         …
  public int evaluate() {
    //infix to Postfix
    String pfExpr = infixToPostFix(expression);
```

Listing 34.6 `Context` Class

```
class Context {
  private HashMap varList = new HashMap();
  public void assign(String var, int value) {
    varList.put(var, new Integer(value));
  }
  public int getValue(String var) {
    Integer objInt = (Integer) varList.get(var);
    return objInt.intValue();
  }
  public Context() {
    initialize();
  }
  //Values are hardcoded to keep the example simple
  private void initialize() {
    assign("a”,20);
    assign("b”,40);
    assign("c”,30);
    assign("d”,10);
  }
}
```

```
    //build the Binary Tree
    Expression rootNode = buildTree(pfExpr);
    //Evaluate the tree
    return rootNode.evaluate(ctx);
  }
                        …
                        …
}//End of class
```

Infix-to-Postfix Conversion (Listing 34.8)

An expression in the standard form is an infix expression.

Example: (a + b) * (c – d)

An infix expression is more easily understood by humans but is not suitable for evaluating expressions by computers. The usage of precedence rules and parentheses in the case of complex expressions makes it difficult for computer evaluation of

Listing 34.7 `Calculator` Class

```
public class Calculator {
  private String expression;
  private HashMap operators;
  private Context ctx;
  public static void main(String[] args) {
    Calculator calc = new Calculator();
    //instantiate the context
    Context ctx = new Context();
    //set the expression to evaluate
    calc.setExpression("(a+b)*(c-d)");
    //configure the calculator with the
    //Context
    calc.setContext(ctx);
    //Display the result
    System.out.println(" Variable Values: " +
                       "a=" + ctx.getValue("a") +
                       ″, b=" + ctx.getValue("b") +
                       ″, c=" + ctx.getValue("c") +
                       ″, d=" + ctx.getValue("d"));
    System.out.println(" Expression = (a+b)*(c-d)");
    System.out.println(" Result = " + calc.evaluate());
  }
  public Calculator() {
    operators = new HashMap();
    operators.put("+″,"1");
    operators.put("-″,"1");
    operators.put("/″,"2");
    operators.put("*″,"2");
    operators.put("(″,"0");
  }
         …
         …
}//End of class
```

these expressions. A postfix expression does not contain parentheses, does not involve precedence rules and is more suitable for evaluation by computers.

The postfix equivalent of the example expression above is ab+cd–*.

A detailed description of the process of converting an infix expression to its postfix form is provided in the Additional Notes section.

Listing 34.8 `Calculator` Class Performing the Infix-to-Postfix Conversion

```
public class Calculator {
      …
      …
  private String infixToPostFix(String str) {
    Stack s = new Stack();
    String pfExpr = "";
    String tempStr = "";
    String expr = str.trim();
    for (int i = 0; i < str.length(); i++) {
      String currChar = str.substring(i, i + 1);
      if ((isOperator(currChar) == false) &&
          (!currChar.equals("(")) &&
          (!currChar.equals(")"))) {
        pfExpr = pfExpr + currChar;
      }
      if (currChar.equals("(")) {
        s.push(currChar);
      }
      //for ')' pop all stack contents until '('
      if (currChar.equals(")")) {
        tempStr = (String) s.pop();
        while (!tempStr.equals("(")) {
          pfExpr = pfExpr + tempStr;
          tempStr = (String) s.pop();
        }
        tempStr = "";
      }
      //if the current character is an
      //operator
      if (isOperator(currChar)) {
        if (s.isEmpty() == false) {
          tempStr = (String) s.pop();
          String strVal1 =
```

(continued)

Listing 34.8 `Calculator` Class Performing the Infix-to-Postfix Conversion (Continued)

```
              (String) operators.get(tempStr);
            int val1 = new Integer(strVal1).intValue();
            String strVal2 =
              (String) operators.get(currChar);
            int val2 = new Integer(strVal2).intValue();
            while ((val1 >= val2)) {
              pfExpr = pfExpr + tempStr;
              val1 = -100;
              if (s.isEmpty() == false) {
                tempStr = (String) s.pop();
                strVal1 = (String) operators.get(
                             tempStr);
                val1 = new Integer(strVal1).intValue();
              }
            }
            if ((val1 < val2) && (val1 != -100))
              s.push(tempStr);
          }
          s.push(currChar);
        }//if
      }//for
      while (s.isEmpty() == false) {
        tempStr = (String) s.pop();
        pfExpr = pfExpr + tempStr;
      }
      return pfExpr;
    }
              …
              …
  }//End of class
```

Construction of the Tree Structure (Listing 34.9)

The postfix equivalent of the input infix expression is scanned from left to right and a tree structure is built using the following algorithm:

1. Initialize an empty stack.
2. Scan the postfix string from left to right.

Listing 34.9 `Calculator` Class Building a Tree with Operators as Nonterminal Nodes and Operands as Terminal Nodes

```
public class Calculator {
          …
          …
  public void setContext(Context c) {
    ctx = c;
  }
  public void setExpression(String expr) {
    expression = expr;
  }
          …
          …
  private Expression buildTree(String expr) {
    Stack s = new Stack();
    for (int i = 0; i < expr.length(); i++) {
      String currChar = expr.substring(i, i + 1);
      if (isOperator(currChar) == false) {
        Expression e = new TerminalExpression(currChar);
        s.push(e);
      } else {
        Expression r = (Expression) s.pop();
        Expression l = (Expression) s.pop();
        Expression n =
          getNonTerminalExpression(currChar, l, r);
        s.push(n);
      }
    }//for
    return (Expression) s.pop();
  }
          …
          …
}//End of class
```

3. If the scanned character is an operand:
   a. Create an instance of the `TerminalExpression` class by passing the scanned character as an argument.
   b. Push the `TerminalExpression` object to the stack.

4. If the scanned character is an operator:
   a. Pop two top elements from the stack.
   b. Create an instance of an appropriate `NonTerminalExpression` subclass by passing the two stack elements retrieved above as arguments.
5. Repeat Step 3 and Step 4 for all characters in the postfix string.
6. The only remaining element in the stack is the root of the tree structure.

The example postfix expression ab+cd–* results in the following tree structure as in Figure 34.2.

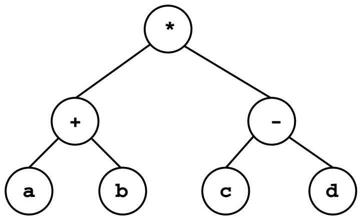

Figure 34.2 Example Expression: Tree Structure

Postorder Traversal of the Tree

The `Calculator` traverses the tree structure and evaluates different `Expression` objects in its postorder traversal path. There are four major tree traversal techniques. These techniques are discussed as part of the Additional Notes section. Because the binary tree in the current example is a representation of a postfix expression, the postorder traversal technique is followed for the expression evaluation. The `Calculator` object makes use of a helper `Context` object to share information with different `Expression` objects constituting the tree structure. In general, a `Context` object is used as a global repository of information. In the current example, the `Calculator` object stores the values of different variables in the `Context`, which are used by each of different `Expression` objects in evaluating the part of the expression it represents.

The postorder traversal of the tree structure in Figure 34.2 results in the evaluation of the leftmost subtree in a recursive manner, followed by the rightmost subtree, then the `NonTerminalExpression` node representing an operator.

ADDITIONAL NOTES

Infix-to-Postfix Conversion

Infix Expression

An expression in the standard form is an infix expression.

Example: a * b + c/d

Sometimes, an infix expression is also referred to as an in-order expression.

Postfix Expression

The postfix (postorder) form equivalent of the above example expression is ab*cd/+.

Conversion Algorithm

See Table 34.2 for the conversion algorithm.

Table 34.2 Conversion Algorithm

1.	*Define operator precedence rules* — In general arithmetic, the descending order of precedence is as shown in the rules below:

Precedence Rules	
*, /	Same precedence
+, –	Same precedence
Expressions are evaluated from left to right.	

2.	Initialize an empty stack.
3.	Initialize an empty postfix expression.
4.	Scan the infix string from left to right.
5.	If the scanned character is an operand, add it to the postfix string.
6.	If the scanned character is a left parenthesis, push it to the stack.
7.	If the scanned character is a right parenthesis:
	a. Pop elements from the stack and add to the postfix string until the stack element is a left parenthesis.
	b. Discard both the left and the right parenthesis characters.
8.	If the scanned character is an operator:
	a. If the stack is empty, push the character to the stack.
	b. If the stack is not empty:
	i. If the element on top of the stack is an operator:
	A. Compare the precedence of the character with the precedence of the element on top of the stack.
	B. If top element has higher or equal precedence over the scanned character, pop the stack element and add it to the Postfix string. Repeat this step as long as the stack is not empty and the element on top of the stack has equal or higher precedence over the scanned character.
	C. Push the scanned character to stack.
	ii. If the element on top of the stack is a left parenthesis, push the scanned character to the stack.
9.	Repeat Steps 5 through 8 above until all the characters are scanned.
10.	After all characters are scanned, continue to pop elements from the stack and add to the postfix string until the stack is empty.
11.	Return the postfix string.

Example

As an example, consider the infix expression (A + B) * (C – D). Let us apply the algorithm described above to convert this expression into its postfix form.

Initially the stack is empty and the postfix string has no characters. Table 34.3 shows the contents of the stack and the resulting postfix expression as each character in the input infix expression is processed.

Table 34.3 Infix-to-Postfix Conversion Algorithm Tracing

Infix Expression Character	*Observation and Action to Be Taken*	*Stack*	*Postfix String*
(	Push to the stack.	(	
A	Operand. Add to the postfix string.	(	A
+	Operator. The element on top of the stack is a left parenthesis and hence push + to the stack.	(+	A
B	Operand. Add to the postfix string.	(+	AB
)	Right parenthesis. Pop elements from the stack until a left parenthesis is found. Add these stack elements to the postfix string. Discard both left and right parentheses.		AB+
*	Operator. The element on top of the stack is +. The precedence of + is less than the precedence of *. Push the operator to the stack.	*	AB+
(	Push to the stack.	*(	AB+
C	Operand. Add to the postfix string.	*(	AB + C
–	Operator. The element on top of the stack is a left parenthesis and hence push + to the stack.	*(–	AB + C
D	Operand. Add to the Postfix string.	*(–	AB + CD
)	Right parenthesis. Pop elements from the stack until a left parenthesis is found. Add these stack elements to the postfix string. Discard both left and right parentheses.	*	AB + CD–
All characters in the infix expression are scanned	Add all remaining stack elements to the postfix string.		AB + CD–*

Binary Tree Traversal Techniques

There are four different tree traversal techniques — Preorder, In-Order, Postorder and Level-Order. Let us discuss each of these techniques by using the following binary tree in Figure 34.3 as an example.

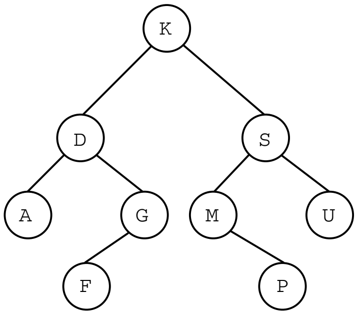

Figure 34.3 Example Sorted Tree Structure

Preorder (Node-Left-Right)

Start with the root node and follow the algorithm as follows:

- Visit the node first.
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder.

A preorder traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:

```
KDAGFSMPU
```

In-Order (Left-Node-Right)

Start with the root node and follow the algorithm as follows:

- Traverse the left subtree in in-order.
- Visit the node.
- Traverse the right subtree in in-order.

An in-order traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:

```
ADFGKMPSU
```

Postorder (Left-Right-Node)

Start with the root node and follow the algorithm as follows:

- Traverse the left subtree in in-order.
- Traverse the right subtree in in-order.
- Visit the node.

A postorder traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:

```
AFGDPMUSK
```

Level-Order

Start with the root node level and follow the algorithm as follows:

- Traverse different levels of the tree structure from top to bottom.
- Visit nodes from left to right with in each level.

A level-order traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:

```
KDSAGMUFP
```

PRACTICE QUESTIONS

1. Enhance the example application to include the division and the unary arithmetic negation operations.
2. Design an interpreter for the DOS `copy` command. The `copy` command can be used to create a new file with the contents of a single or multiple files:

- The `Copy a.txt c.txt` command copies the contents of the `a.txt` file to the new `c.txt` file.
- The `Copy a.txt + b.txt c.txt` command copies the contents of both the files `a.txt` and `b.txt` to the new `c.txt` file.

3. Design and develop an interpreter to display a given integer value in words.
4. Redesign the example application using the Visitor pattern.
   a. Design a `Visitor` with different `visit(ExpressionObjectType)` methods and a `getResult()` method.
   b. Convert the input infix expression to postfix expression.
   c. Scan the postfix expression from left to right.
      i. When an operand is found push to stack.
      ii. When an operator is found:
         A. Pop two operands from the stack.
         B. Create an appropriate `Expression` object.
         C. When the `Expression` object is created, it invokes an appropriate `visit` method on the `Visitor` instance by passing itself as an argument. The `Visitor` in turn calls the `evaluate` method on the `Expression` object. The integer result of the `evaluate` method call is then pushed to the stack.
         D. Once the postfix expression is scanned from left to right, the `getResult()` method can be invoked on the Visitor to get the final result. The Visitor can retrieve the only remaining stack element and return it.