

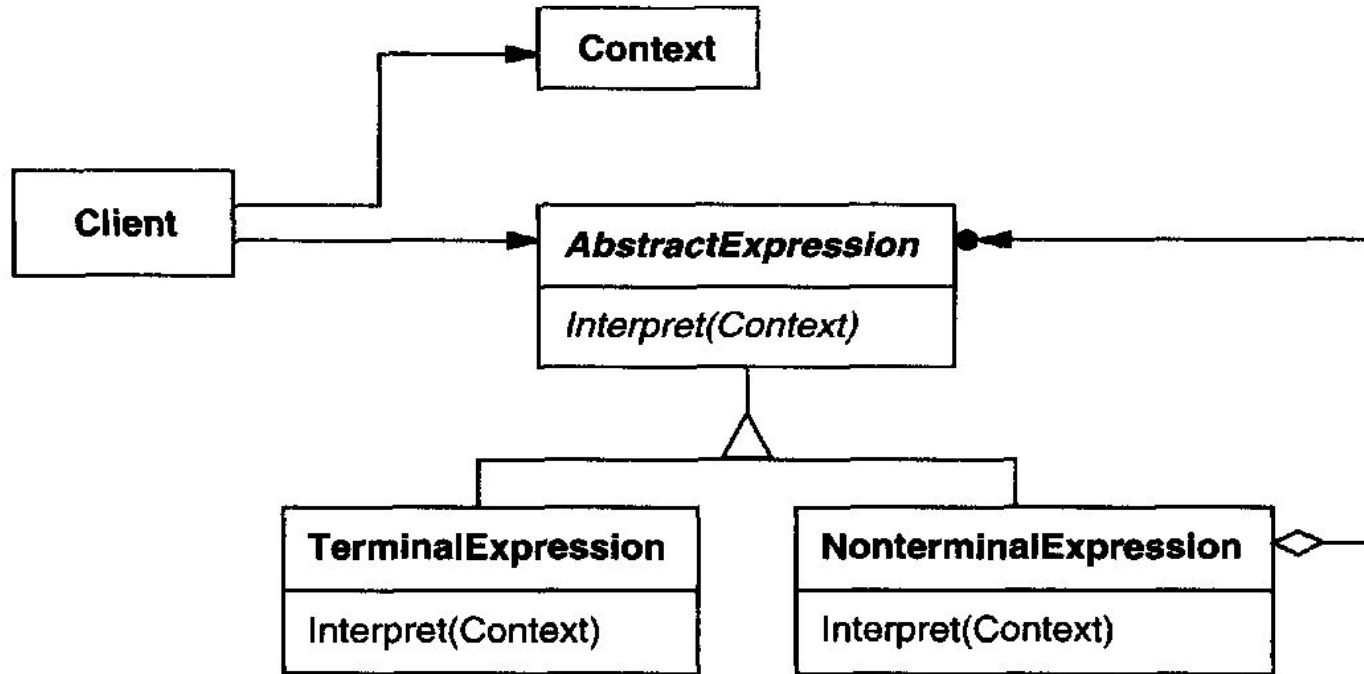
Interpreter

Akash Bhosale, Hao Zhang, Biao A

Intent

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

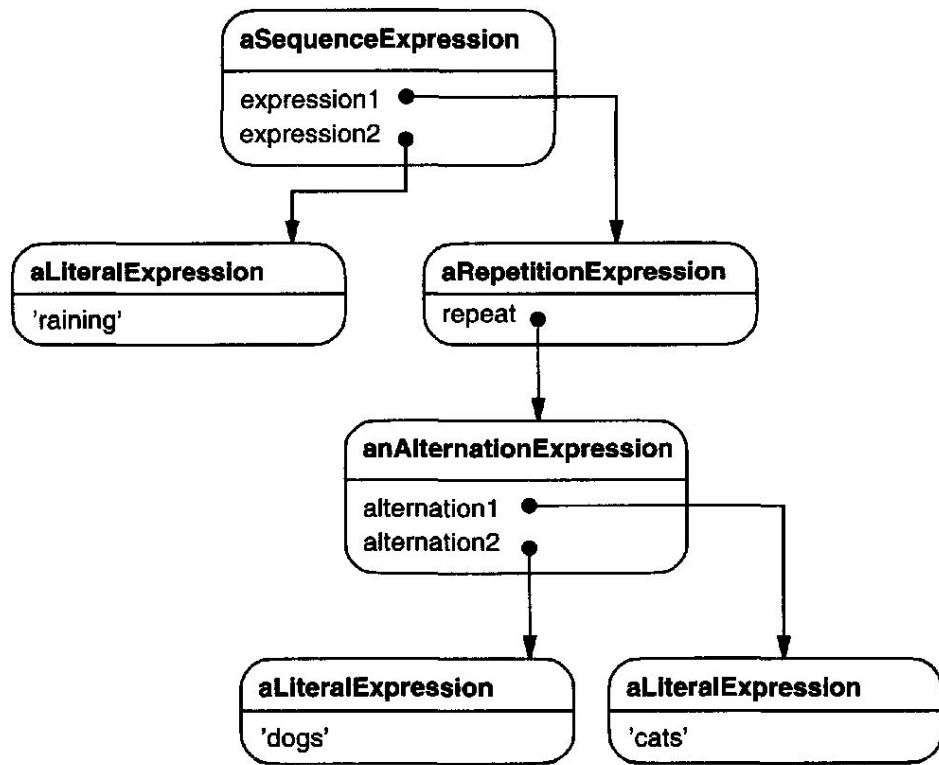
Structure



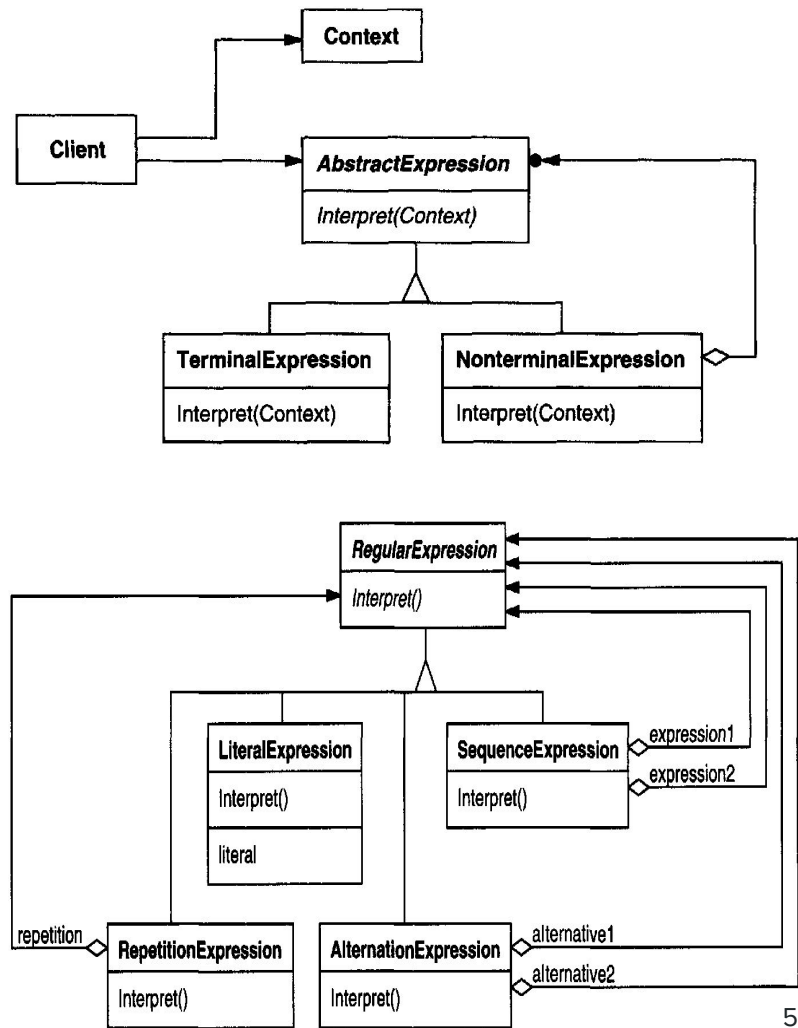
Motivation

- A common problem of searching for strings that match a pattern.
- Regular expressions are used as a standard language for specifying pattern of strings.
- Interpret a regular expression that specifies a set of strings to match rather than building custom algorithms for each pattern.
- Express instances of a commonly occurring problem as sentences in a simple language and let the interpreter solve the problem by interpreting sentences.

Example

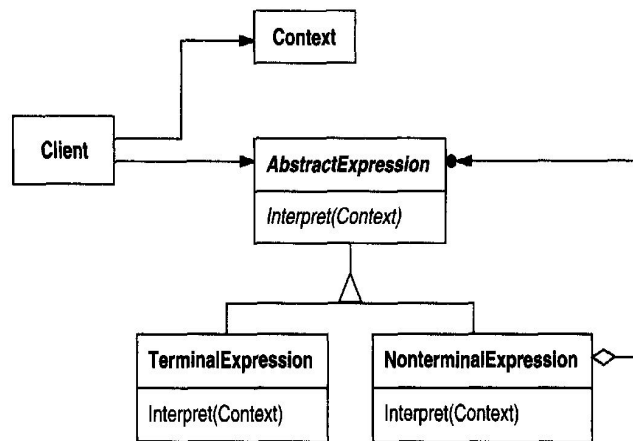


Represents the expression: `raining & (dogs | cats)*`



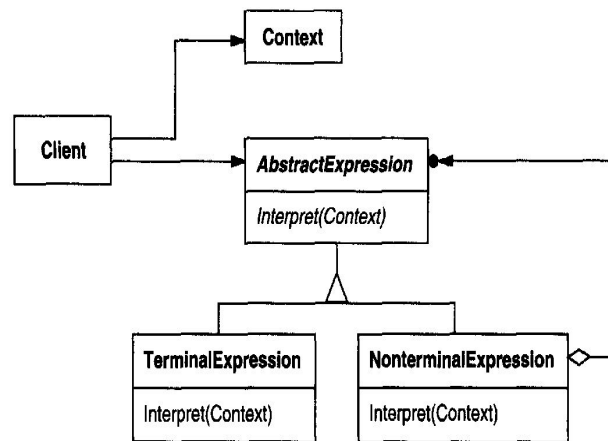
Participants

- **AbstractExpression**
 - declare an abstract interpret operation
- **TerminalExpression**
 - implements an interpret operation associated with terminal symbols in the grammar
- **NonterminalExpression** (Alternate Expression, Repetition Expression, Sequence Expression)
 - One class for every rule
 - implements an interpret operation for nonterminal symbols
- **Context**
 - contains information global to the interpreter
- **Client**
 - builds an AST representing a particular sentence in the language
 - invokes the interpret operation



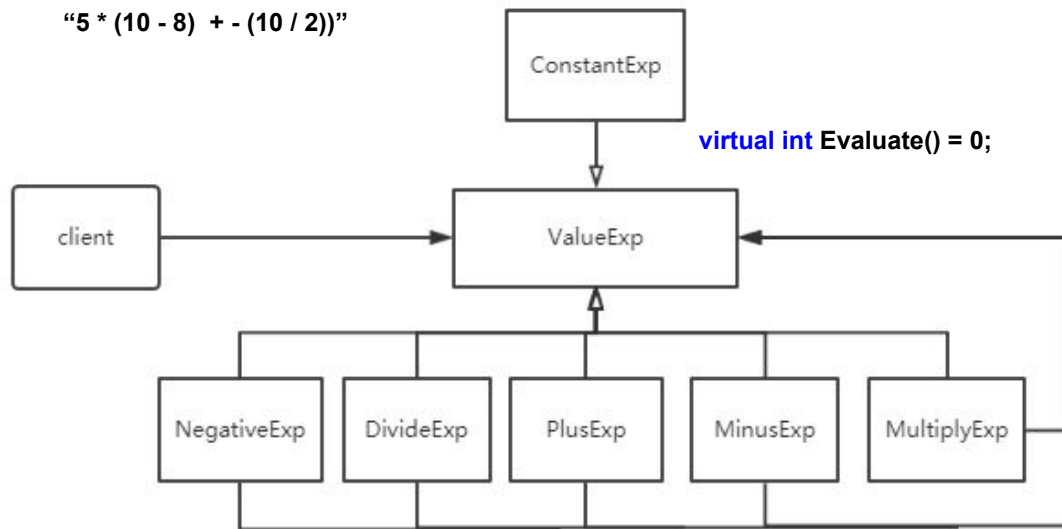
How to use this pattern

- Client builds a AST of terminal and nonterminal instances
- Initialize the context and invoke interpret operation
- Each nonterminal expression implements a interpret operation in terms of interpret on each subexpression
- The interpret operation of each terminal expression 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.



Sample Code

“5 * (10 - 8) + - (10 / 2)”



ValueExp (AbstractExpression):

Declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

ConstantExp (TerminalExpression):

Numbers in this case

Negative, Divide, Plus, Minus, Multiply (NonterminalExpression):

One such class is required for every rule in the grammar.

Client:

Builds (or is given) an abstract syntax tree and invokes the Interpret operation.

Sample Code

```
class ValueExp{
public:
    ValueExp() {}
    virtual ~ValueExp() {}
    virtual int Evaluate() = 0;
};
```

```
class constantExp : public ValueExp{
public:
    constantExp(int value) : value_(value) {}
    virtual ~constantExp() {}
    virtual int Evaluate() {
        return value_;
    }
private:
    int value_;
};
```

```
class NegativeExp : public ValueExp
{
public:
    NegativeExp(ValueExp *a) : a_(a) {}
    virtual ~NegativeExp() {}
    virtual int Evaluate() {
        return -(a_->Evaluate());
    }
private:
    ValueExp *a_;
};
```

```
class PlusExp : public ValueExp
{
public:
    PlusExp(ValueExp *a, ValueExp *b): a_(a), b_(b) {}
    virtual ~PlusExp() {}
    virtual int Evaluate() {
        return a_->Evaluate() + b_->Evaluate();
    }
private:
    ValueExp *a_;
    ValueExp *b_;
};
```

Sample Code

```
class MinusExp : public ValueExp
{
public:
    MinusExp(ValueExp *a, ValueExp *b): a_(a), b_(b) {

    }

    virtual ~MinusExp() {}

    virtual int Evaluate() {
        return a_>Evaluate() - b_>Evaluate();
    }
private:
    ValueExp *a_;
    ValueExp *b_;
};
```

```
class MultiplyExp : public ValueExp
{
public:
    MultiplyExp(ValueExp *a, ValueExp *b): a_(a)
        , b_(b) {

    }

    virtual ~MultiplyExp() {}

    virtual int Evaluate() {
        return a_>Evaluate() * b_>Evaluate();
    }
private:
    ValueExp *a_;
    ValueExp *b_;
};
```

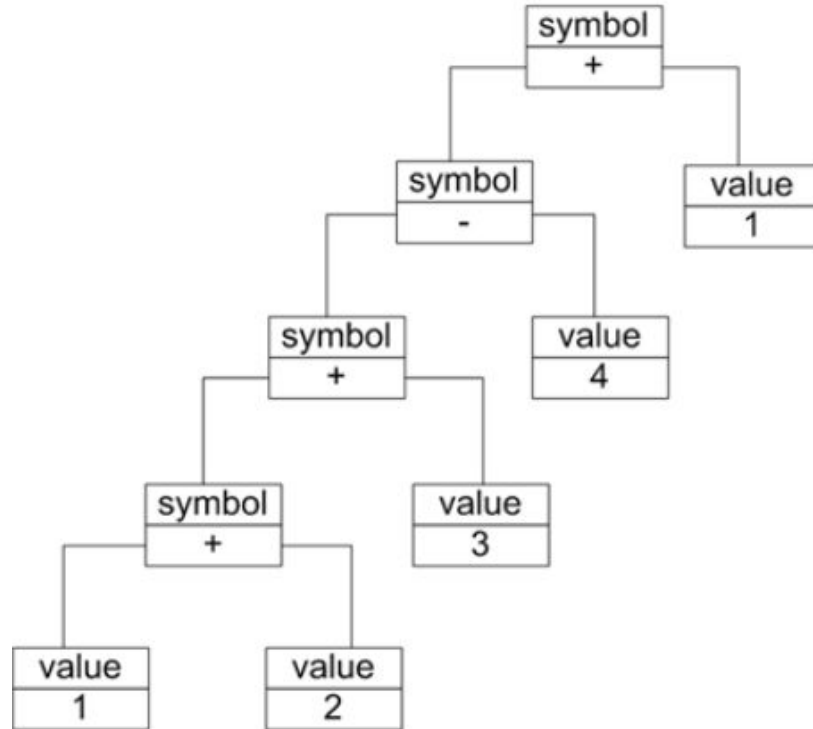
```
class DivideExp : public ValueExp
{
public:
    DivideExp(ValueExp *a, ValueExp *b)
        : a_(a)
        , b_(b) {

    }

    virtual ~DivideExp() {}

    virtual int Evaluate() {
        return a_>Evaluate() / b_>Evaluate();
    }
private:
    ValueExp *a_;
    ValueExp *b_;
};
```

Abstract Syntax Tree, AST



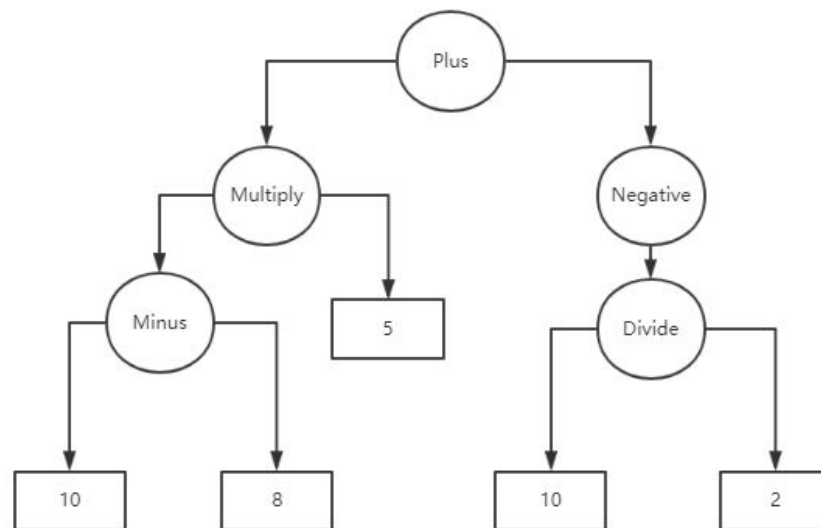
Sample Code

```
ValueExp *exp = new PlusExp(  
    new MultiplyExp(  
        new constantExp(5), new MinusExp(  
            new constantExp(10), new constantExp(8)  
        )  
    ),  
    new NegativeExp(  
        new DivideExp(new constantExp(10), new constantExp(2)))  
);  
  
printf("5 * (10 - 8) + -(10 / 2)) = %d.\n", exp->Evaluate());
```

C:\ Select C:\WINDOWS\system32\cmd.exe

```
5 * (10 - 8) + -(10 / 2)) = 5.  
Press any key to continue . . .
```

$$5 * (10 - 8) + -(10 / 2))$$



Sample Code

```
vector<string> toRPN(string expression) {...}
```

```
ValueExp* calculate(vector<string> tokens) {  
    stack<ValueExp*> stk;  
    for (string token : tokens) {  
        if ('0' <= token.back() && token.back() <= '9')  
            stk.push(new ConstantExp(stoi(token)));  
        else {  
            ValueExp* num1 = stk.top();  
            stk.pop();  
            ValueExp* num2 = stk.top();  
            stk.pop();  
            ValueExp* temp;  
            if (token == "+") temp = new PlusExp(num1, num2);  
            if (token == "-") temp = new MinusExp(num1, num2);  
            if (token == "*") temp = new MultiplyExp(num1, num2);  
            if (token == "/") temp = new DivideExp(num1, num2);  
            stk.push(temp);  
        }  
    }  
    return stk.top();  
}
```

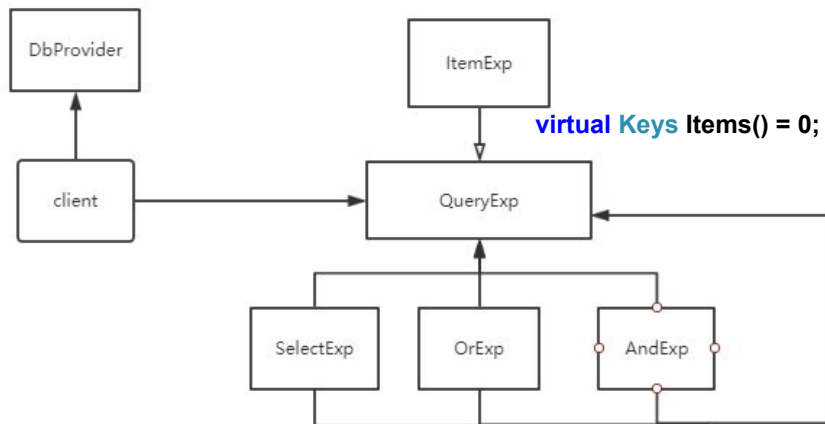
"5 * (10 - 8) - (10 / 2)"



10, 8, -, 5, *, 10, 2, /, -

```
vector<string> tokens = toRPN("5 * (10 - 8) + -(10 / 2)");  
cout << calculate(tokens)->Evaluate();
```

Sample Code



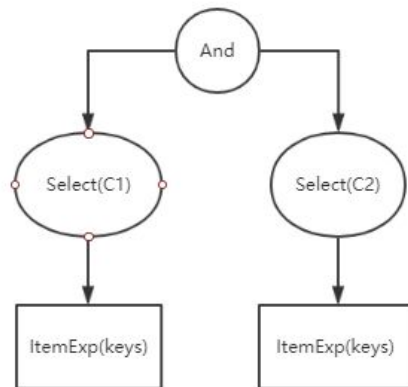
```
class QueryExp {
public:
    QueryExp() {}
    virtual ~QueryExp() {}
    virtual Keys Items() = 0;
};
```

```
class ItemExp :public QueryExp {
public:
    ItemExp(DbCore<PayLoad>& db): _keys(db.keys()){}
    virtual ~ItemExp() {}
    virtual Keys Items() {
        return _keys;
    }

private:
    Keys _keys;
};
```

Sample Code

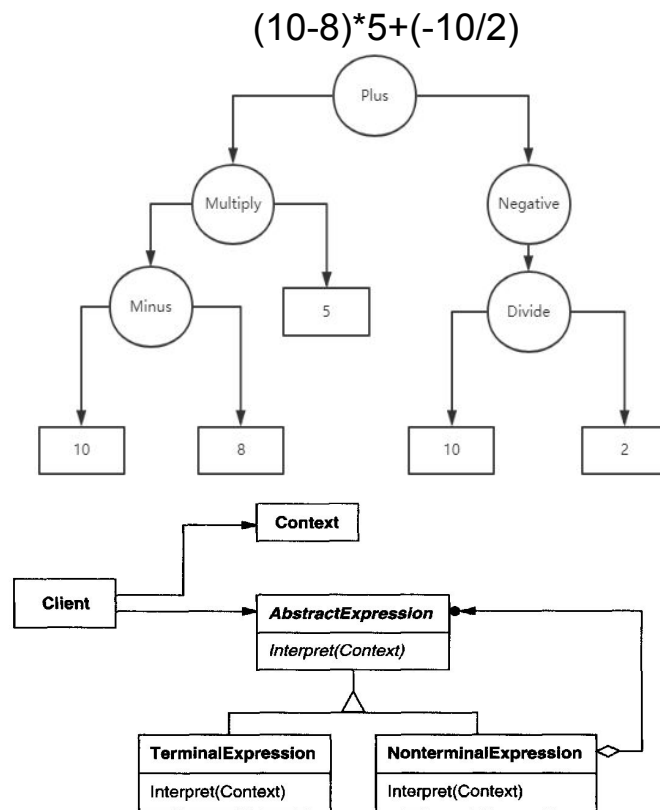
```
class SelectExp :public QueryExp {
public:
    SelectExp(QueryExp* QueryExp, Condition c) :
        _items(QueryExp), _c(c) {}
    virtual ~SelectExp() {}
    virtual Keys Items() {
        //...select the ikeys with condition c
        return keys;
    }
private:
    Condition _c;
    QueryExp* _items;
};
```



```
QueryExp* resultSet = new AndExp(
    new SelectExp(new ItemExp (db), c1), new SelectExp(new ItemExp (db), c2)
);
```

Tiny checklist for Using Interpreter pattern

1. Decide if a "little language" offers a justifiable return on investment.
2. Define a grammar for the language.
3. Map each part in the grammar to a class.
4. Organize the suite of classes into the structure of the Composite pattern to build an Abstract Syntax Tree(AST).
5. Define an interpreter(Context) method to interpret the AST.



Pros and cons

1. Implementing the grammar is easy.
2. It's easy to change and extend the grammar, too. (New operations like '%' '^')
3. Complex grammars are hard to implement.
4. Complex loop and recursion make it hard to debug and will make the program inefficiently.
5. Need a specialist.

Where to use?

1. Where the grammar is not complex.
2. Where the grammar is known pretty well.
3. Where the class hierarchy could be defined as a language.
4. Where the performance is not a big issue.

Other examples

1. Specialized database query(SQL) languages which support query in DB.
2. SMA and OOD course which use interpreter to store and load data from xml files.
3. A lot of programming languages like Java could be interpret first and then compile.
4. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.



Related patterns

Composite : The abstract syntax tree is an instance of the Composite pattern.

Flyweight : Shows how to share terminal symbols within the abstract syntax tree.

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

Visitor : The pattern can be used to maintain the behavior in each node in the abstract syntax tree in one class.