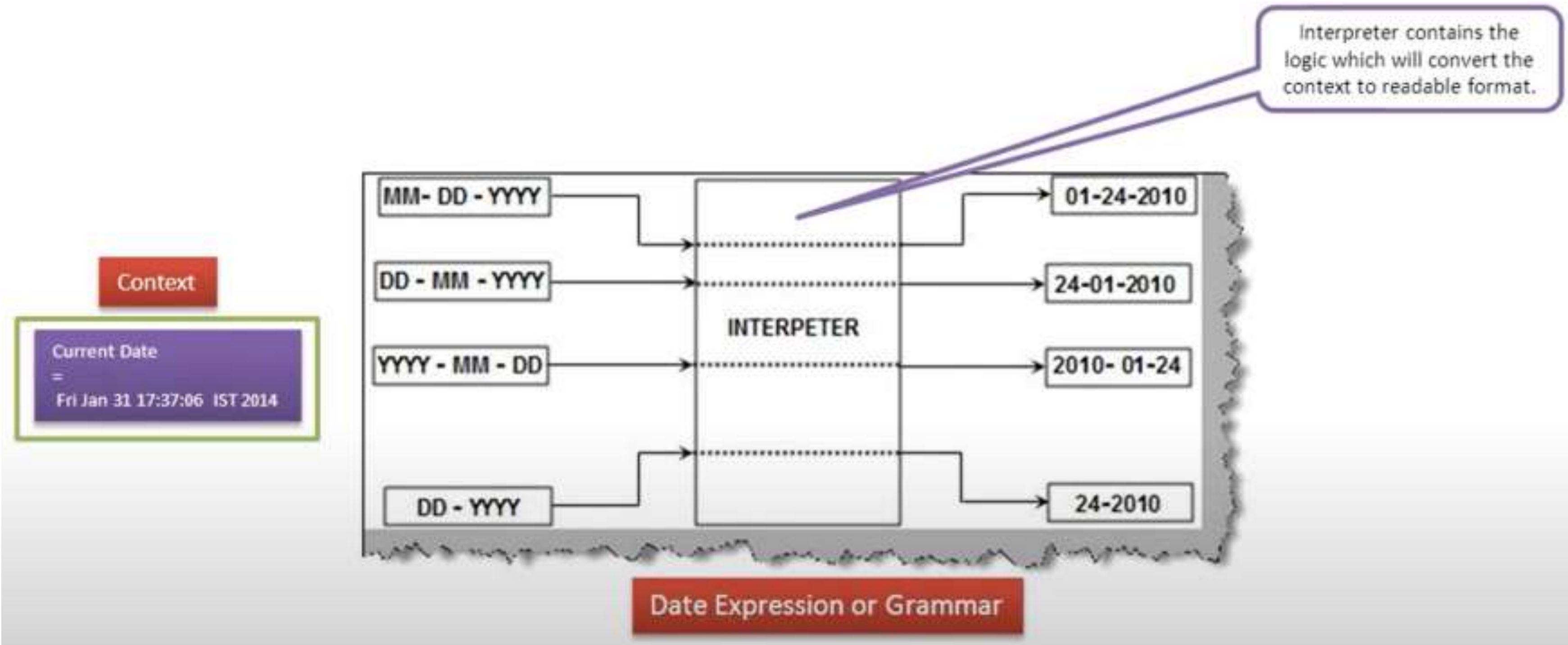
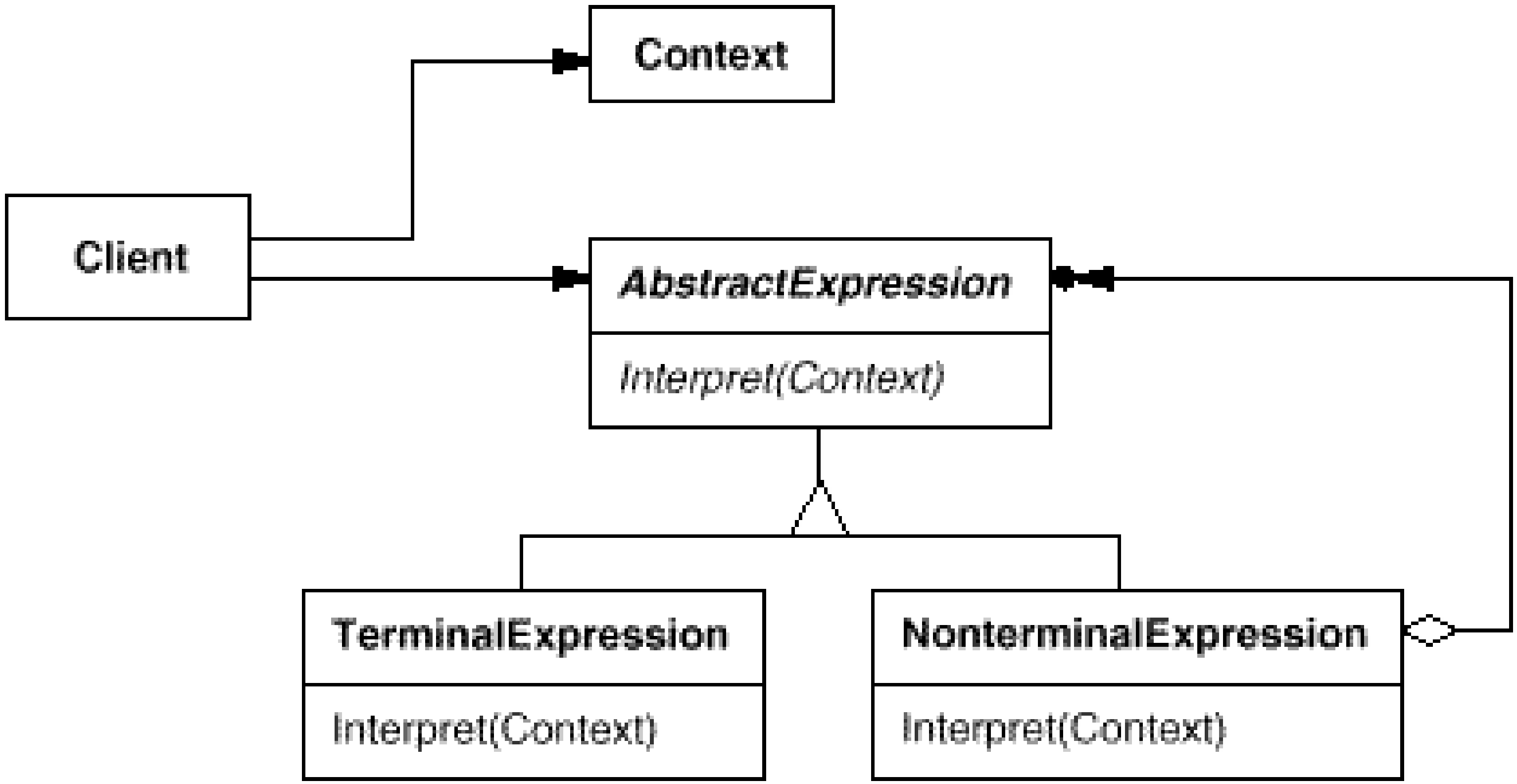
Interpreter

What is interpreter

- Behavioral pattern
- "Cursor"
- Provides way to evaluate language grammar or expression
- Motivation



Structure


Context
Client
AbstractExpression
Interpret(Context)
TerminalExpression
Interpret(Context)
NonterminalExpression
Interpret(Context)

Structure

- **AbstractExpression**
  - Declares abstract method Interpret()
  - Its implementation is responsible for interpretation of processed expression
- **TerminalExpression**
  - Implementation of method Interpret() associated with terminal of grammar
  - Instance for every terminal symbol in input
- **NonterminalExpression**
  - Implementation of method Interpret() of non-terminal
  - Class for every rule of grammar $R ::= R_1 R_2 \ldots R_N$
- **Context**
  - Global info (i.e. value we want to interprete)
- **Client**
  - Gets/creates abstract syntax tree representing sentence of language
    - Made from instances of NonterminalExpression and TerminalExpression
  - Calls method Interpret()

Example – regular expression

- **Grammar of regular expression**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

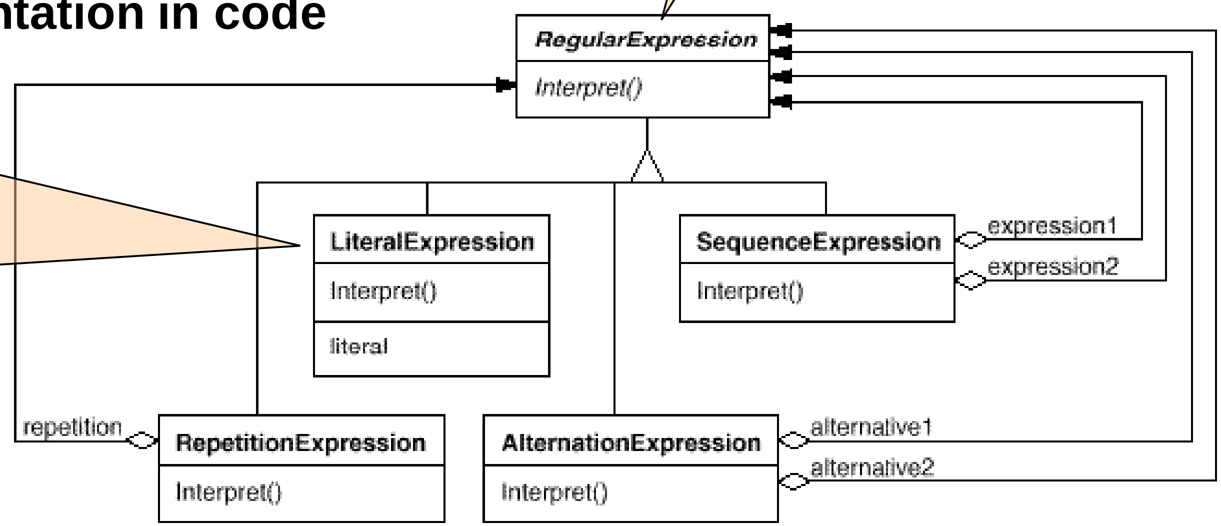
Example – regular expression

- **Grammar of regular expression**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
- **Grammar representation in code**

Abstract class

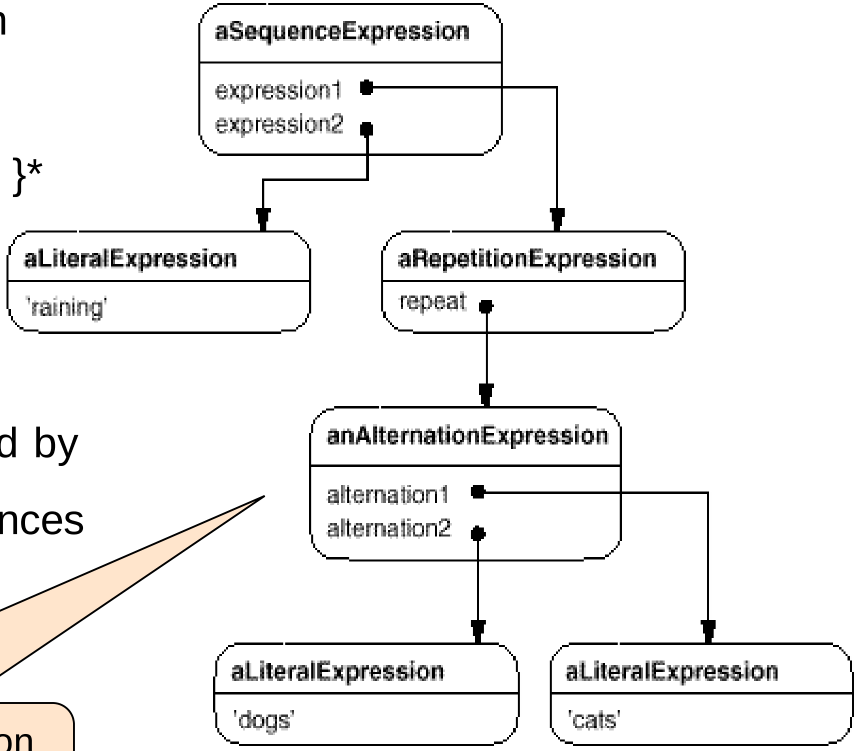
Class for every rule
(Instances maintain
subexpression)

Symbols in right side of
expressions are in
variables



Example – regular expression

- **Grammar of regular expression**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

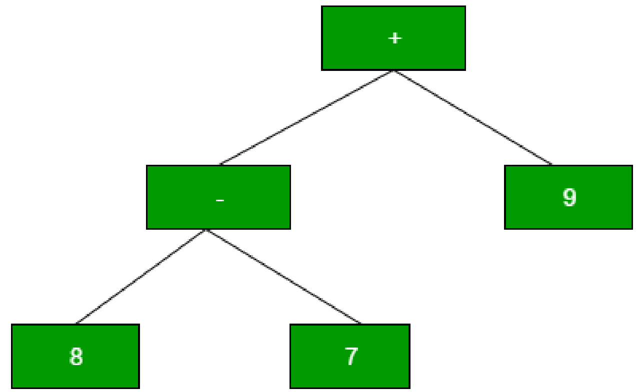


- **Abstract syntax tree**
  - Each regular expression is represented by abstract syntax tree made from instances of classes

Representation of regular expression
raining & (dog | cats) *

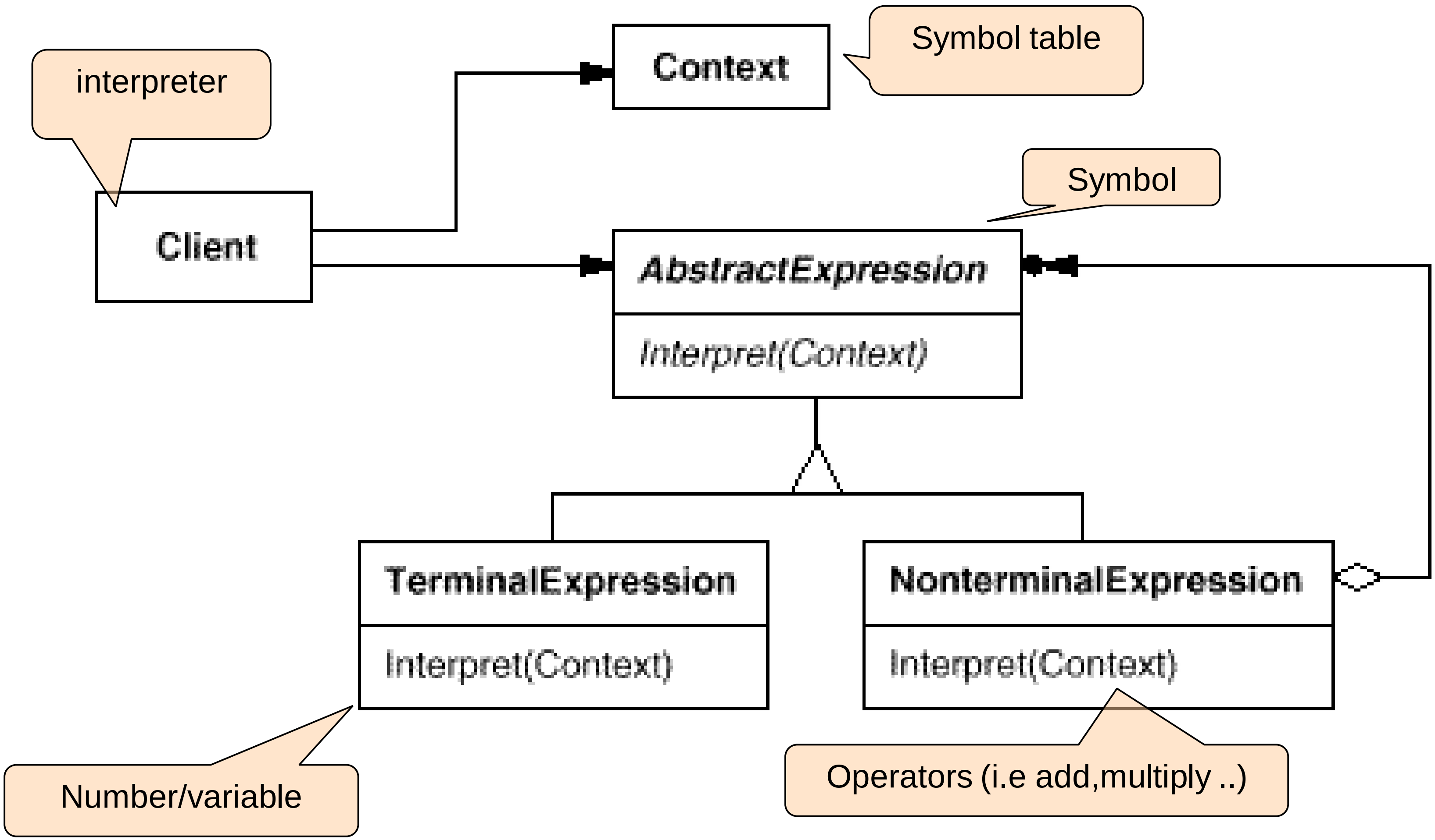
Example - postfix calculator

- **Expression**
  - “9 8 7 - +”
  - Tree representation

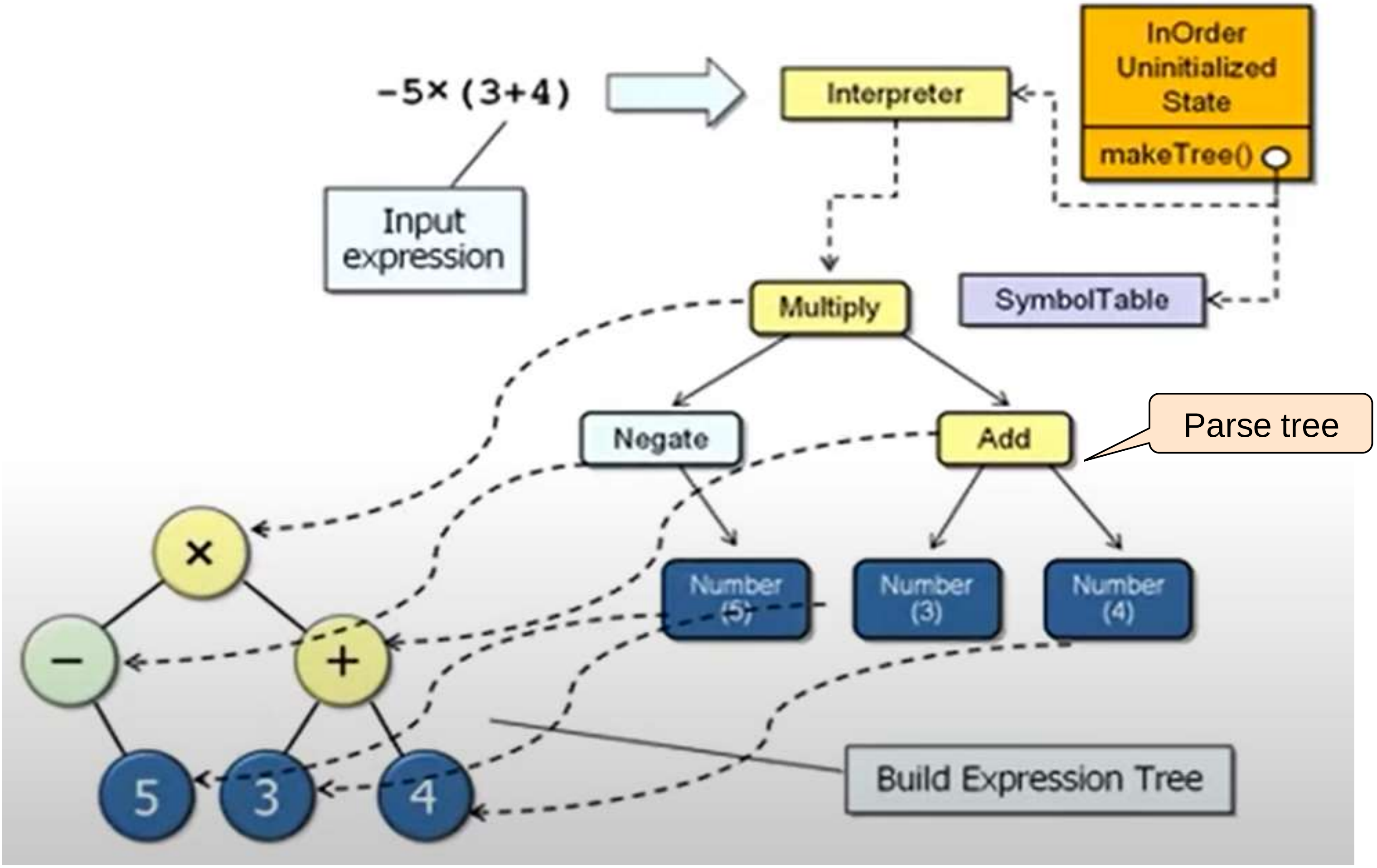


- **Algorithm**
  - Create empty stack
  - Create list of tokens from input expression separated by space
  - For every token
    - If symbol is number -> add number to stack
    - If symbol is operator -> take corresponding number of numbers from stack and apply operator, return output to stack
  - If the expression is read without mistakes and stack has only one value than that value is final output

Example - postfix calculator



Example - postfix calculator



Example - postfix calculator

```
public interface Expression {

    int interpret();
}
```

Example - postfix calculator

- Terminal Expression

```
public class NumberExpression implements Expression{

    private int number;

    public NumberExpression(int number){
        this.number=number;
    }

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

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

Example - postfix calculator

- **Nonterminal expression**

```
public class AdditionExpression implements Expression {

   private Expression firstExpression,secondExpression;

   public AdditionExpression(
           Expression firstExpression,
           Expression secondExpression){

      this.firstExpression=firstExpression;
      this.secondExpression=secondExpression;
   }

   @Override
   public int interpret(){
       return this.firstExpression.interpret() +
              this.secondExpression.interpret();
   }
}
```

Example - postfix calculator

```
public class ParserUtil {

   public static boolean isOperator(String symbol) {
       return (symbol.equals("+") ||
               symbol.equals("-") ||
               symbol.equals("*"));
   }

   public static Expression getExpressionObject(
    Expression firstExp,
    Expression secondExp,
    String symbol){
               if(symbol.equals("+"))
           return new AdditionExpression(firstExp,secondExp);
       else if(symbol.equals("-"))
           return new SubtractionExpression(firstExp,secondExp);
       else
           return new MultiplicationExpression(firstExp,secondExp);
   }
}
```

Example - postfix calculator

```
public class ExpressionParser {
    Stack stack = new Stack<>();
    public int parse(String str){
        String[] tokenList = str.split(" ");
        for (String symbol : tokenList) {
            if (!ParserUtil.isOperator(symbol)) {
                Expression numberExp = new NumberExpression(symbol);
                stack.push(numberExp);

            } else  if (ParserUtil.isOperator(symbol)) {
                Expression firstExp = stack.pop();
                Expression secondExp = stack.pop();
                Expression operator =
        ParserUtil.getExpressionObject(firstExp, secondExp, symbol);
                stack.push(new NumberExpression(operator.interpret()));
            }
        }
        int result = stack.pop().interpret();
        return result;
    }
}
```

Example - boolean expression

- **Boolean expressions**
  - BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp | '(' BooleanExp ')'
  - AndExp ::= BooleanExp 'and' BooleanExp
  - OrExp ::= BooleanExp 'or' BooleanExp
  - NotExp ::= 'not' BooleanExp
  - Constant ::= 'true' | 'false'
  - VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'

Interface for every class defining Boolean expression

```
interface BooleanExp {

  public bool interpret(Context context);
};
```

Context defines mapping of variables into Boolean values (constants true and false)

```
class Context {

  public bool lookup(String name);
  public void assign(VariableExp exp,
boolean bool);
};
```

Example - boolean expression

- Class for rule VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'‘

```
class VariableExp implements BooleanExp {
  private String name;

  VariableExp(String name){
    this.name = name;
  };

  public boolean interpret(Context context){
    return context.lookup(name);
  }
};
```

- Class for rule Constant ::= 'true' | 'false'

```
class Constant implements BooleanExp {
  private boolean bool;

  Constant(boolean bool){
    this.bool = bool;
  };

  public boolean interpret(Context context){
    return bool;
  }
};
```

Example - boolean expression

- Class for rule AndExp ::= BooleanExp 'and' BooleanExp

```
class AndExp implements BooleanExp {
  private BooleanExp operand1;
  private BooleanExp operand2;

  AndExp(BooleanExp op1, BooleanExp op2){
    operand1 = op1;
    operand2 = op2;
  };

  public boolean interpret(Context context){
    return operand1.interpret(context) && operand2.interpret(context);
  };

};
```

Same for rules
OrExp a **NotExp**

Example - boolean expression

- Creation of expression instance + its interpretation

```
BooleanExp expression;
Context context;

VariableExp x = new VariableExp("X");
VariableExp y = new VariableExp("Y");

expression = new OrExp(
  new AndExp(new Constant(true), x),
  new AndExp(y, new NotExp(x))
);

context.assign(x, false);
context.assign(y, true);

boolean result = expression.intepret(context);
```

Create abstract syntax tree for expression
(true and x) or (y and (not x))

Variables evaluation

Interpretes expression as *true*, *Possible to change evaluation and evaluate expression again*

Example – complex calculator

- **Kalkulačka**

```
expression ::= plus | minus | variable | number
plus ::= expression '+' expression
minus ::= expression '-' expression
variable ::= 'a' | 'b' | 'c' | ... | 'z'
digit ::= '0' | '1' | ... | '9'
number ::= digit | digit number
```

Example – complex calculator

```
import java.util.Map;

interface Expression {
    public int interpret(final Map<String, Expression> variables);
}

class Number implements Expression {
    private int number;
    public Number(final int number) {
                 this.number = number;
         }
    public int interpret(final Map<String, Expression> variables)  {
                 return number;
         }
}
```

Example – complex calculator

```
class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(final Expression left, final Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(final Map<String, Expression> variables) {
        return leftOperand.interpret(variables) +
                  rightOperand.interpret(variables);
    }
}

class Minus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Minus(final Expression left, final Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(final Map<String, Expression> variables) {
        return leftOperand.interpret(variables) -
                  rightOperand.interpret(variables);
    }
}
```

Example – complex calculator

```
class Variable implements Expression {
    private String name;
    public Variable(final String name) {
         this.name = name;
    }
    public int interpret(final Map<String, Expression> variables) {
        if (null == variables.get(name)) return 0;
        return variables.get(name).interpret(variables);
    }
}
```

Other use-cases

- **Compilers**
- **Parsers**
- **Query languages (SQL)**
- **Units conversion**
- **Web browsers, text editors**
  - To lay out documents
  - Check spelling

Pattern specifics

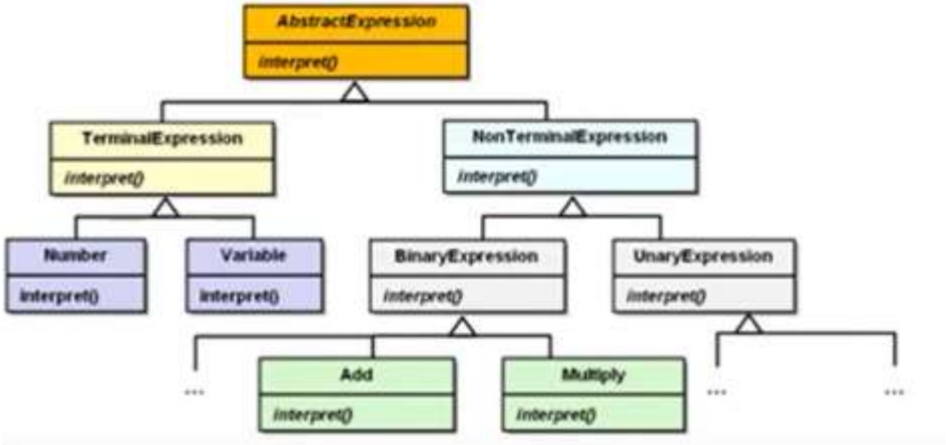
- **Only non-complex grammars**
- **Time/space are not important**
  - State machine instead of syntactic tree

- **Parser is not included in pattern**
- **Tree structure + traversal**
- **Defines grammar**
- **Each grammar rule = one class**

(dis)advantages

- **Advantages**
  - Easy to expand/change grammar
  - Easy implementation of grammar
  - Adding more methods for interpreting
  - Support various types of traversal

- **Disadvantages**
  - Often too complex and hard to maintain



Other patterns

- **Composite**
  - Tree structure is an implementation of Composite
- **Visitor**
  - Similar tree traversal
  - Separated functionality from data
  - Not only counting of value but can transform data as well
- **Iterator**
  - Structure traversal
  - Common abstract ancestor
- **Memento**
  - Capture state of iteration
- **Flyweight**
  - Common for compilers
  - Sharing of constant expressions from compile-time evaluation