Interpreter

Interpreter

- **Motivace**
  - Problém, jehož různé instance je třeba často řešit
  - Tyto instance lze vyjádřit větami v jednoduchém jazyce
    - Příklad: zda daný textový řetězec odpovídá nějakému vzoru (match)
- **Obecné řešení**
  - Vytvoříme interpret tohoto jazyka
  - Interpretace věty jazyka = řešení dané instance problému
    - Příklad: Jazyk pro popis vzorů – regulární výrazy
- **Interpreter specifikuje**
  - Pro daný jazyk:
    - Jak jeho gramatiku reprezentovat v kódu
    - Jak reprezentovat a interpretovat jednotlivé věty tohoto jazyka
- **A to tak, aby vše bylo průhledné a snadno rozšířitelné**

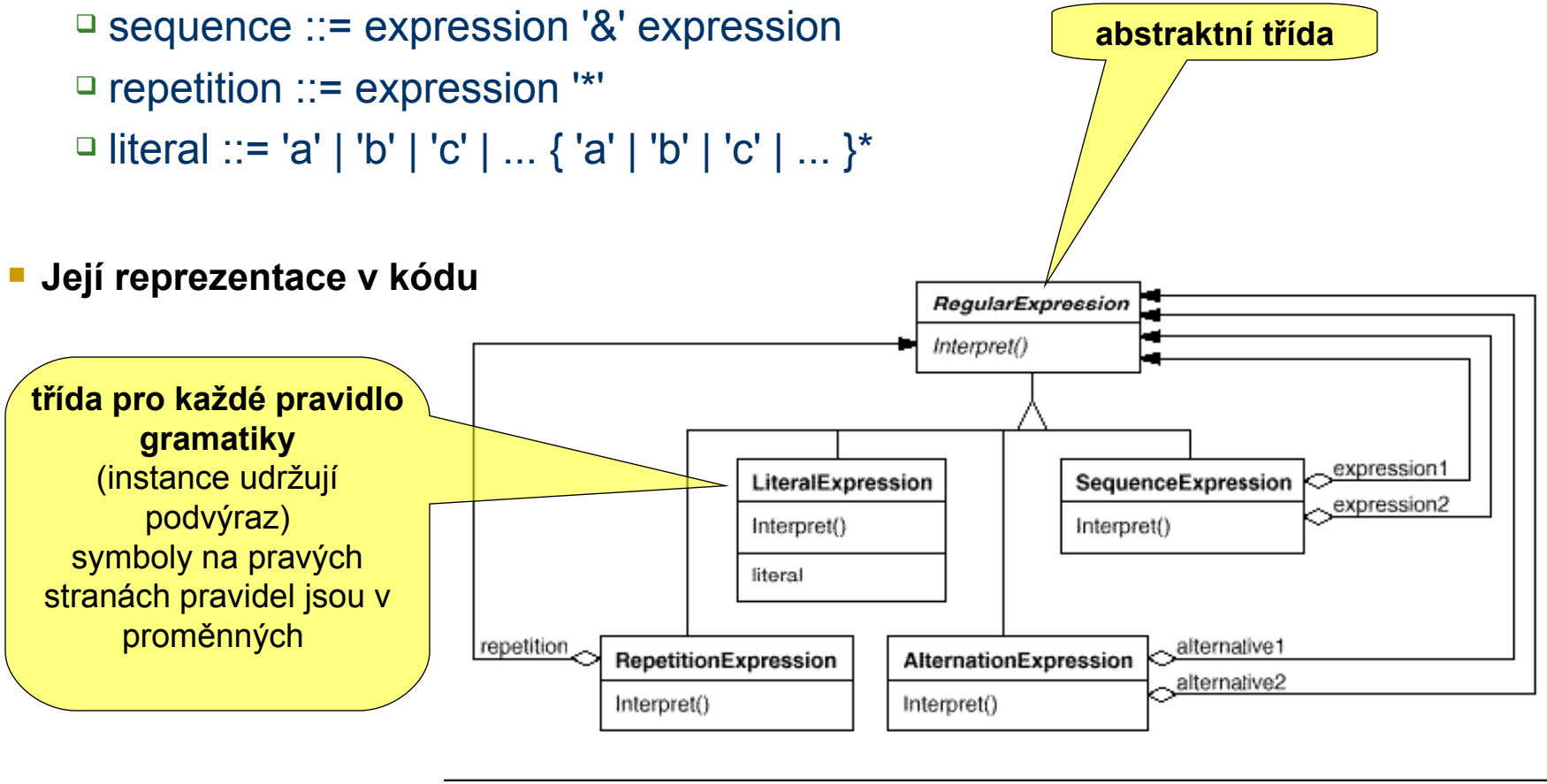
Interpreter – reprezentace gramatiky

- **Příklad: gramatika regulárního výrazu**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

Interpreter – reprezentace gramatiky

- **Příklad: gramatika regulárního výrazu**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

- **Její reprezentace v kódu**

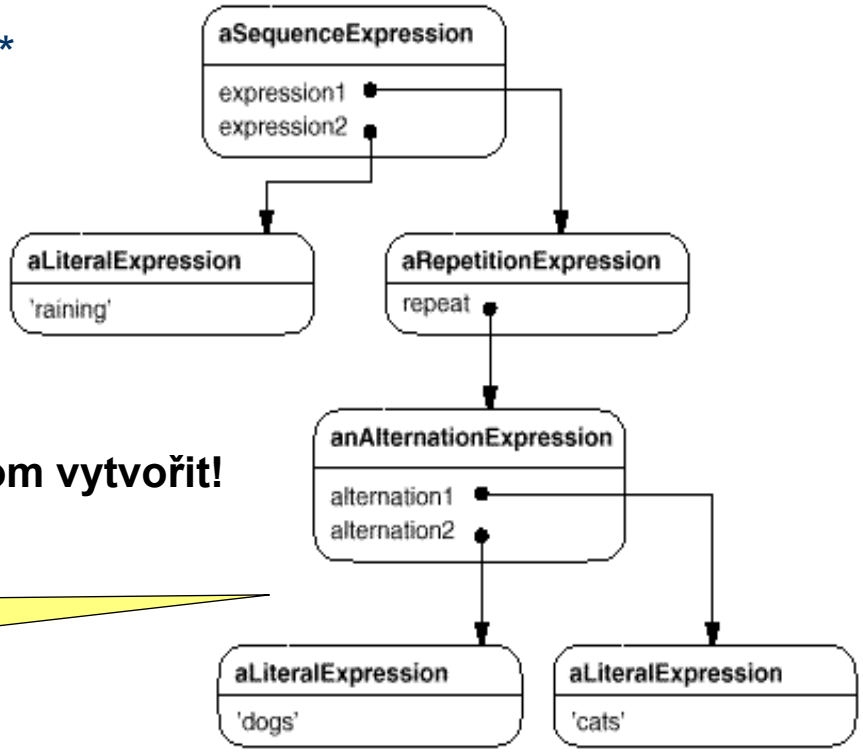


Interpreter – reprezentace vět

- **Příklad: gramatika regulárního výrazu**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

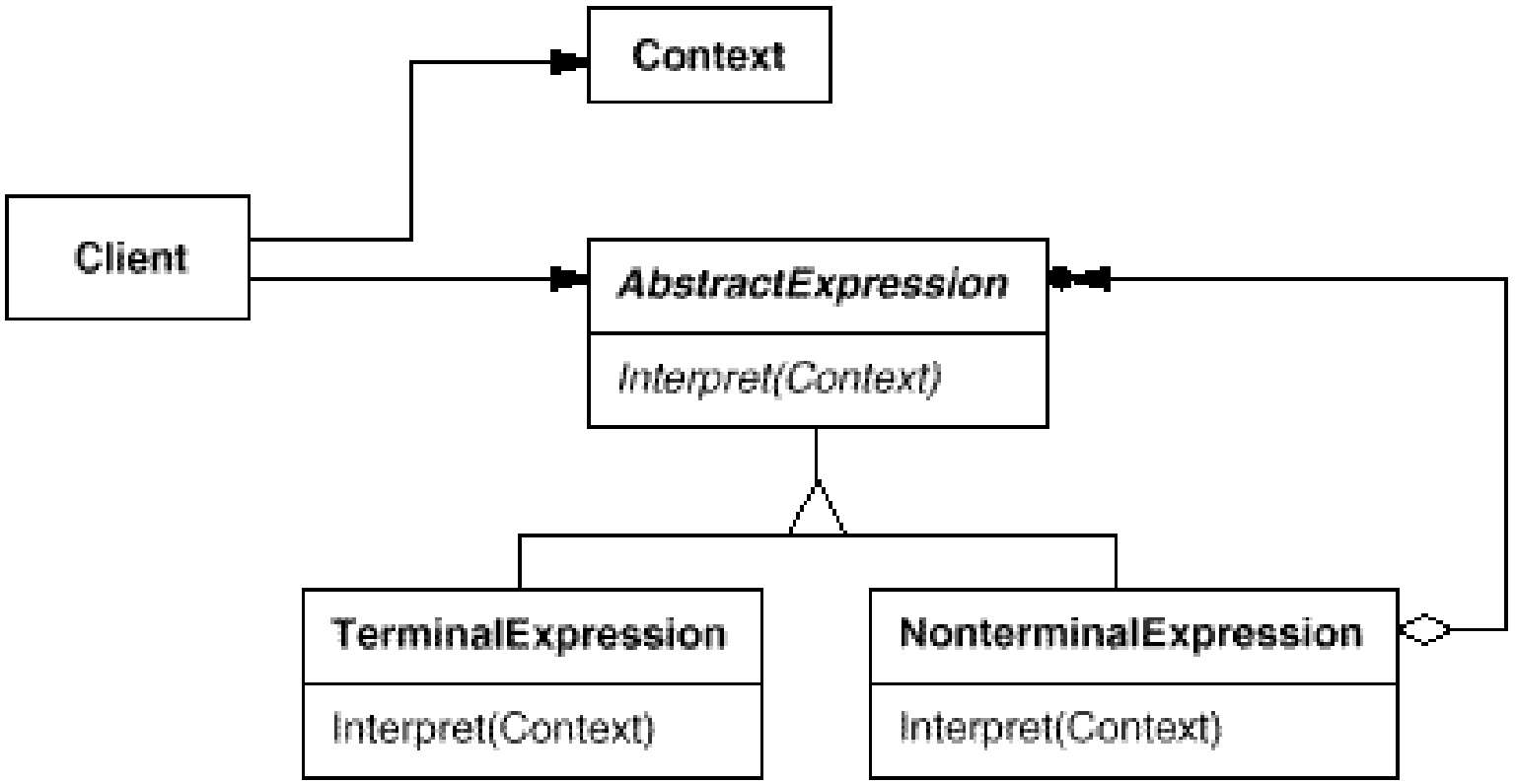
- **Abstraktní syntaktický strom**
  - Každý regulární výraz je reprezentován **abstraktním syntaktickým stromem, tvořeným instancemi zmíněných tříd**

- **Interpreter ovšem nespecifikuje, jak tento strom vytvořit!**



Reprezentace regulárního výrazu
raining & (dog | cats) *

Interpreter – struktura obecně



Interpreter – účastníci

- **AbstractExpression** (RegularExpression)
  - Deklaruje abstraktní metodu *Interpret()*
  - Implementace zajišťuje interpretaci zpracovávaného pojmu
- **TerminalExpression** (LiteralExpression)
  - Implementuje metodu *Interpret()* asociovanou s terminálem gramatiky
  - Instance pro každý terminální symbol ve vstupu (větě)
- **NonterminalExpression** (AlternationExpr, RepetitionExpr, SequenceExpr)
  - Implementuje metodu *Interpret()* neterminálu gramatiky
  - Třída pro každé pravidlo $R::=R_1R_2\ldots R_N$ gramatiky
  - Udržuje instance proměnných typu AbstractExpression pro každý symbol $R_1\ldots R_N$
- **Context**
  - Udržuje globální informace
- **Client**
  - Dostane (vytvoří) abstraktní syntaktický strom reprezentující konkrétní větu jazyka
    - složený z instancí NonterminalExpression a TerminalExpression
  - Volá metodu *Interpret()*

Interpreter - příklad

- **Práce s booleovskými výrazy**
  - 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 pro všechny třídy definující booleovský výraz

```
class BooleanExp {
public:
  BooleanExp();
  virtual ~BooleanExp();
  virtual bool Evaluate(Context&) = 0;
  virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
  virtual BooleanExp* Copy() const = 0;
};
```

kontext definuje mapování proměnných na booleovské hodnoty tj. konstanty ‘true’ a ‘false’

```
class Context {
public:
  bool Lookup(const char*) const;
  void Assign(VariableExp*, bool);
};
```

Interpreter - příklad

```
class VariableExp : public BooleanExp {
public:
  VariableExp(const char*);
  virtual ~VariableExp();
  virtual bool Evaluate(Context&);
  virtual BooleanExp* Replace(const char*, BooleanExp&);
  virtual BooleanExp* Copy() const;
private:
  char* _name;
};
VariableExp::VariableExp (const char* name) {
  _name = strdup(name);
}
bool VariableExp::Evaluate (Context& aContext) {
  return aContext.Lookup(_name);
}
BooleanExp* VariableExp::Copy () const {
  return new VariableExp(_name);
}
BooleanExp* VariableExp::Replace (const char* name, BooleanExp& exp) {
  if (strcmp(name, _name) == 0)
    return exp.Copy();
  else
    return new VariableExp(_name);
}
```

VariableExp reprezentuje pojmenovanou proměnnou

Interpreter - příklad

```
class AndExp : public BooleanExp {
public:
  AndExp(BooleanExp*, BooleanExp*);
  virtual ~ AndExp();
  virtual bool Evaluate(Context&);
  virtual BooleanExp* Replace(const char*, BooleanExp&);
  virtual BooleanExp* Copy() const;
private:
  BooleanExp* _operand1;
  BooleanExp* _operand2;
};
AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
  _operand1 = op1;
  _operand2 = op2;
}
bool AndExp::Evaluate (Context& aContext) {
 return _operand1->Evaluate(aContext) && _operand2->Evaluate(aContext);
}
BooleanExp* AndExp::Copy () const {
  return new AndExp(_operand1->Copy(), _operand2->Copy());
}
BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {
  return new AndExp(
    _operand1->Replace(name, exp),
    _operand2->Replace(name, exp)
  );
}
```

obdobně také operace **OR** a **NOT**

Interpreter - příklad

```
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);

bool result =
        expression->Evaluate(context);
```

vytvoření abstraktního syntaktického stromu pro výraz **(true and x) or (y and (not x))**

ohodnocení proměnných

interpretuje výsledek (true) můžeme změnit ohodnocení a znovu provést interpretaci

jiná interpretace

```
VariableExp* z = new VariableExp("Z");
NotExp not_z(z);

BooleanExp* replacement =
  expression->Replace("Y", not_z);

context.Assign(z, true);

result = replacement->Evaluate(context);
```

Interpreter - shrnutí

- **Použitelnost**
  - Interpretace jazyka, jehož věty lze vyjádřit abstraktním syntaktickým stromem
  - Gramatika jazyka je jednoduchá
    - Složitější gramatiky → nepřehledný kód, lépe použít parser generatory (Bison, Coco/R)
  - Efektivita není kriticky důležitá
    - Jinak lépe nekonstruovat syntaktický strom → stavový automat
- **Typické použití**
  - Parsery a kompilátory
- **Související návrhové vzory**
  - **Composite**
    - Abstraktní syntaktický strom je instancí NV Composite
  - **Flyweight**
    - Sdílení terminálových symbolů uvnitř syntaktického stromu
    - Typické u programovacích jazyků (častý výskyt té samé proměnné)
  - **Iterator**
    - Využití k průchodu strukturou
  - **Visitor**
    - Definice/změna chování všech uzlů abstraktního syntaktického stromu jednou třídou