Cours/TP IUT 2012

Introduction to Abstract Syntaxic Tree
for Math Expression

Design Pattern (Composite, Visitor, ...)

Arnaud Nauwynck
arnaud.nauwynck@gmail.com

Outline

- Introduction to simple Math Expressions
- In-memory Tree representation
- Abstract Syntaxic Tree : Class Hierarchy
- Simple Treatments: pretty print, constant folding, algebraic transform, numeric evaluation, etc...
- Visitor Design Pattern

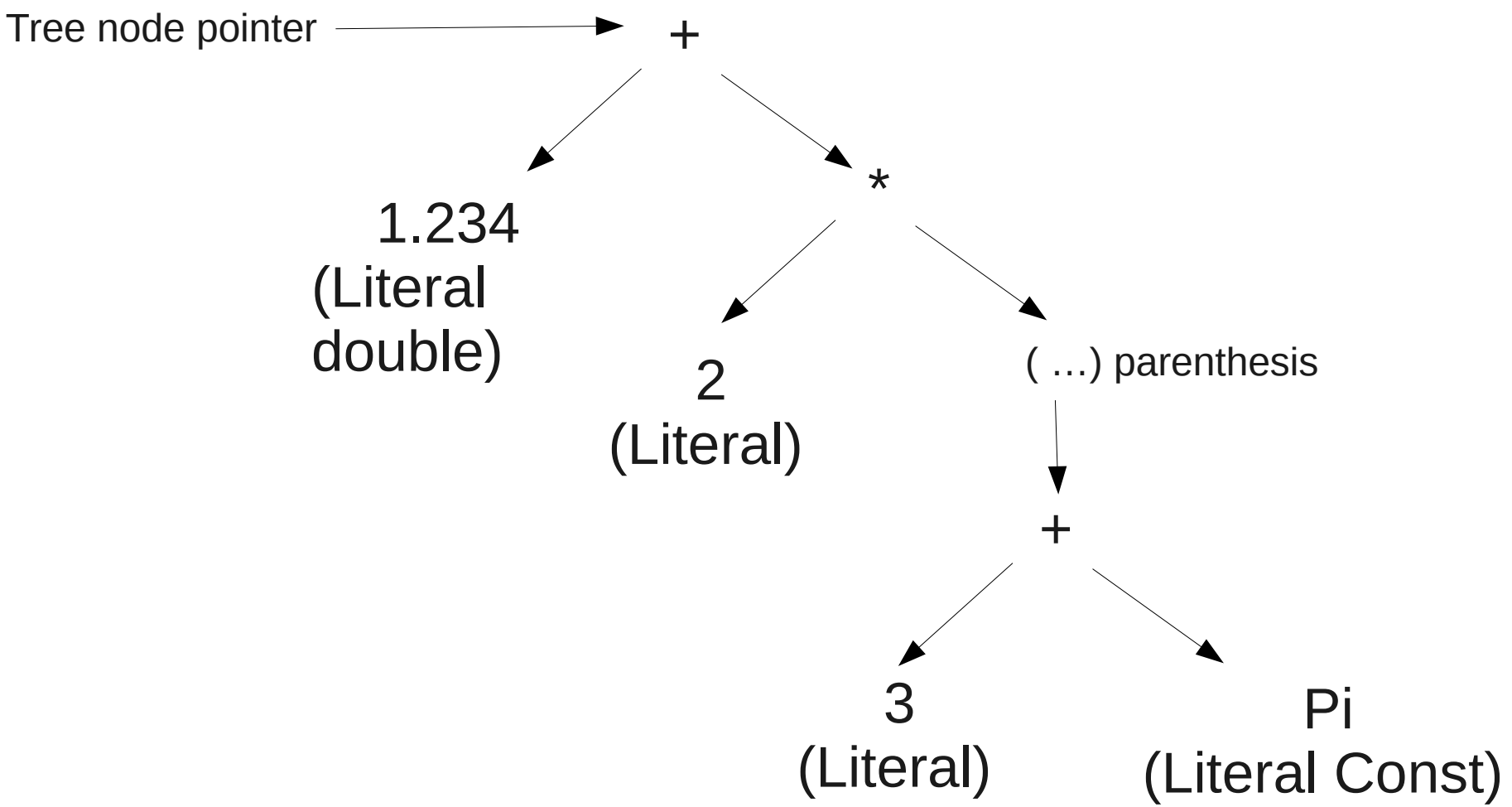
- Annexes: Grammar, scanner, parser

Simple Math Expression

- Literal Values, Constants:
  - Integer, Double, Complex, predefined constants
  - Ex: 1, 123.456, 1+i, pi, e
- Unary Op: (value), -value, value! ...
  - Ex: 12!, -(pi)
- Binary Op: +, -, *, /
  - Ex: 1+1, 2*pi, …
- =>
  - 1+2*3, (1+2)*3+pi, ...

Expression as In-Memory Tree

1.234 + 2 * (3 + pi)

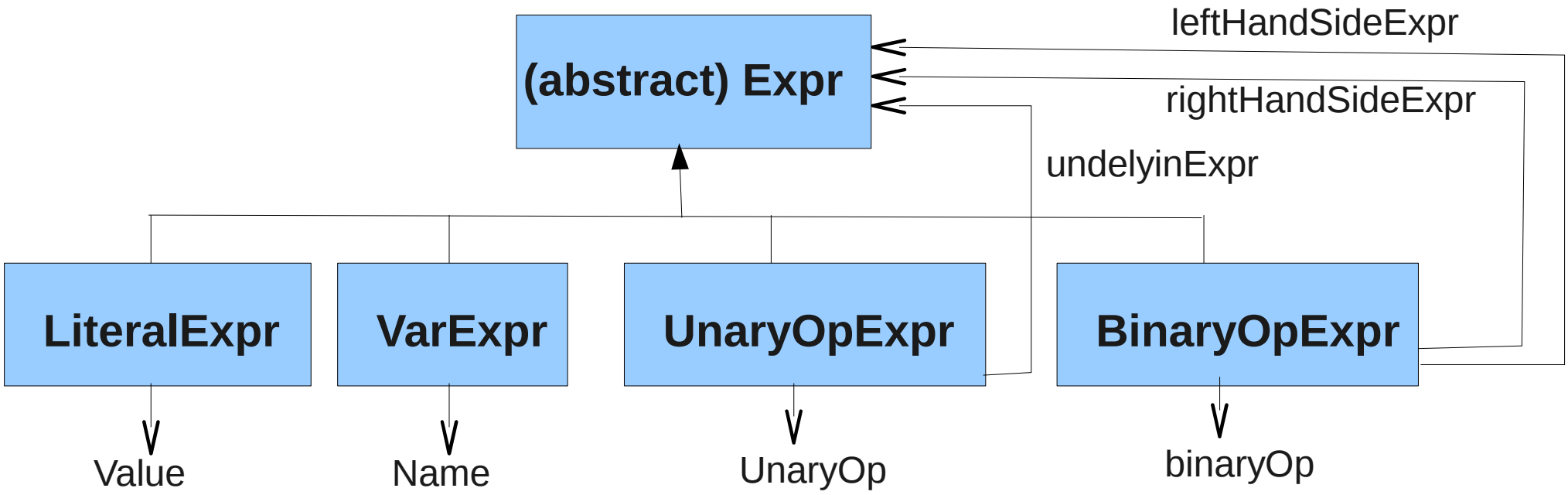


AST Class Hierarchy

- Equivalence Principle with Grammar rules (cf annexes)

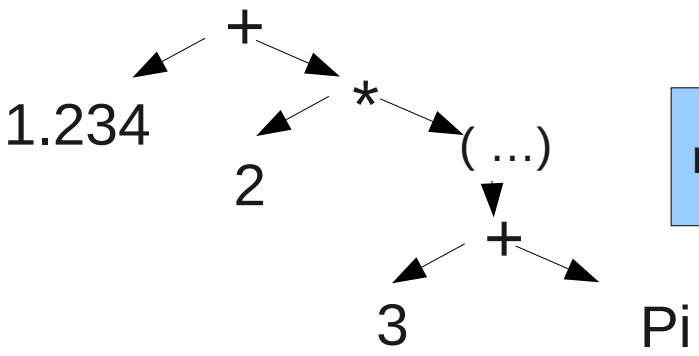
Grammar choice rule = abstract class

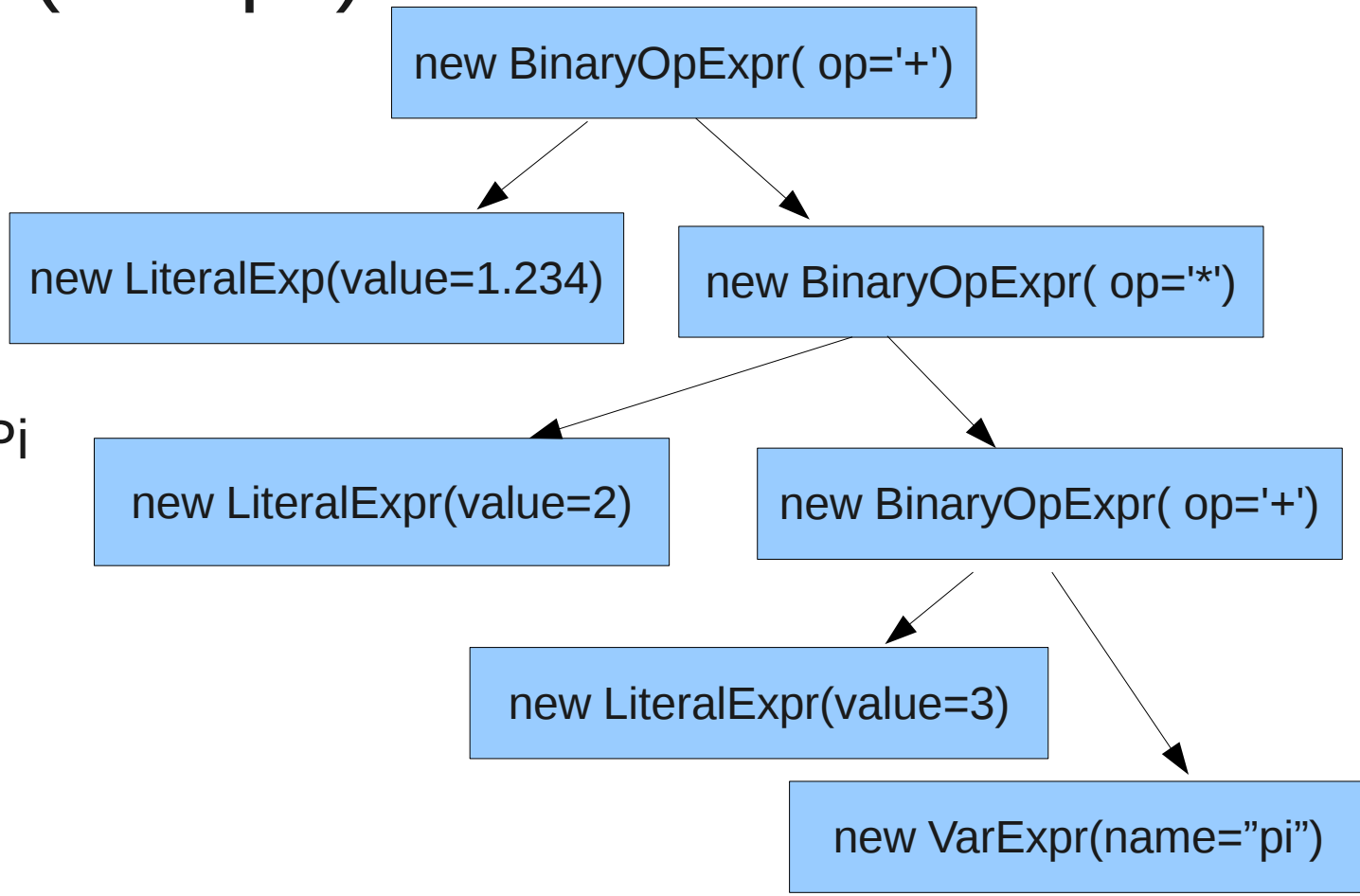
Grammar Rule (with N terms)= concrete sub-Class in class hierarchy (with N fields)



Expression as In-memory Object Instance Tree

1.234 + 2 * (3 + pi)





Theory => TP Step 1 ….

1) Create an Eclipse(opt: Maven) Java Project
2) Write the AST java classes for simple Math Expr (choose package name “fr.iut.tps.algexpr”)
3) Add Junit Dependency
4) Write simple Junit test for creating object instances

More On ASTs

- AST = Abstract Syntaxic Tree

- Abstract (!= CST Concrete Syntaxic Tree)
- the result Tree is no more dependent of syntaxic details...
  - Syntax in Java / C / Python => same AST...
  - No more parenthesis, no ',' ';' decorators …

- Purely functional code for 1.234 + 2 * (3 + pi) :

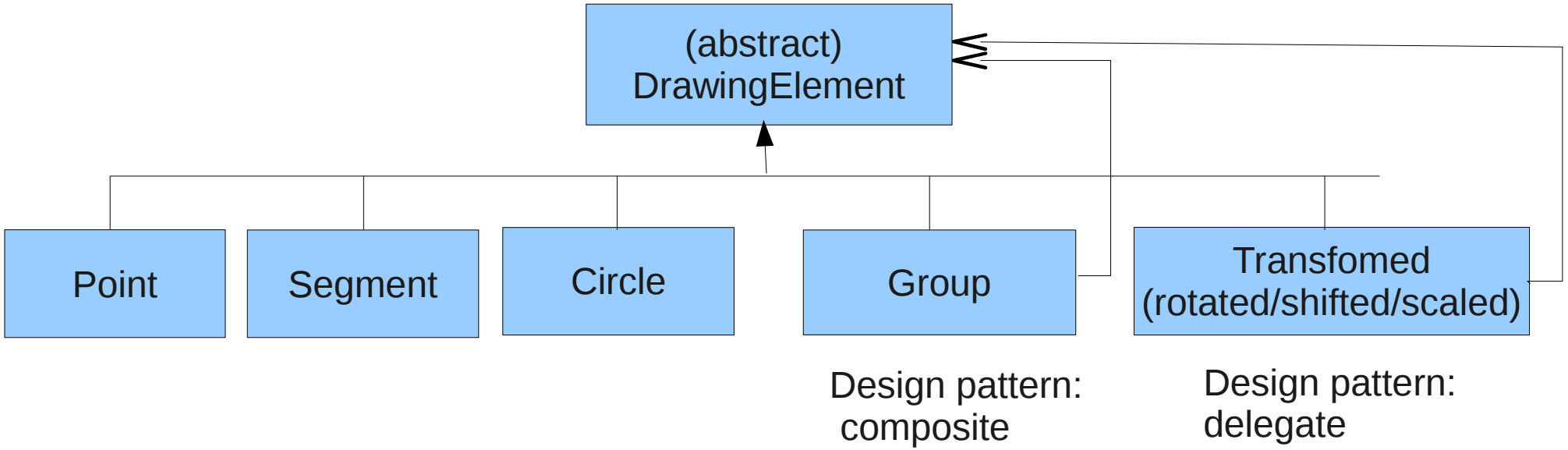
```
plus(  literal(1.234),
       mult(  literal(2),
              plus(  literal(3), var(“pi”) ) ) ) )
```

AST … Domain Driven Design

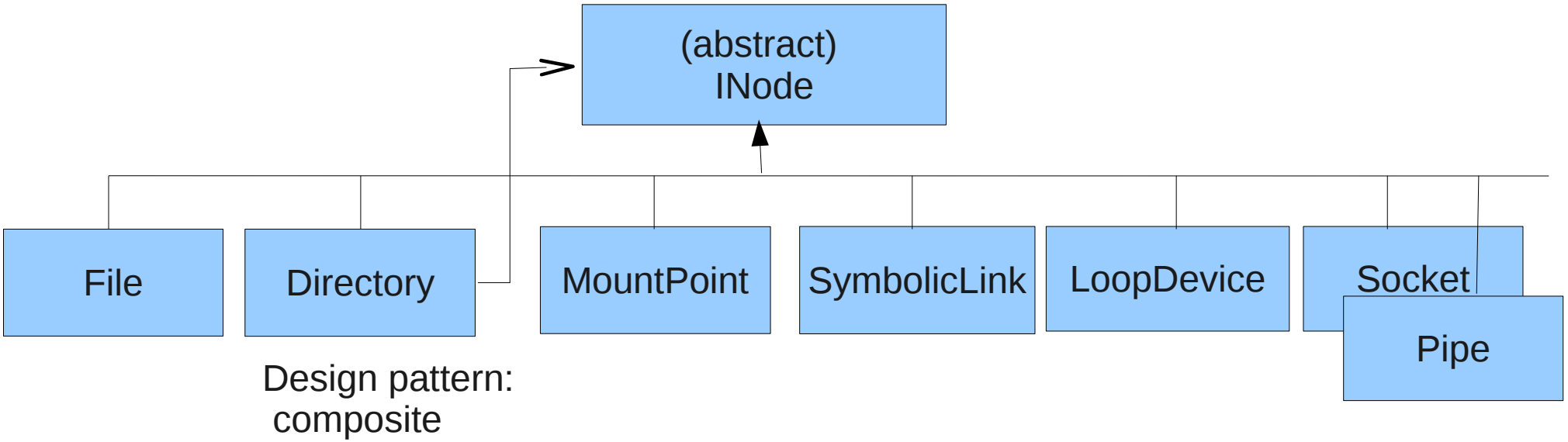
- The AST is always the kernel of the program
- It defines a vocabulary / dictionary of terms
- It is understandable by functional users
- It defines ALL the possible things, all the extensibility features of the program.. but does not contains code itself
- It must be well designed...

Example of AST in other domains

- Langage (geometrical langage) for Drawings...
- Points, Segments, Circle, Figures, …
- Example: SVG, Dia, Xfig, PowerPoint, ...

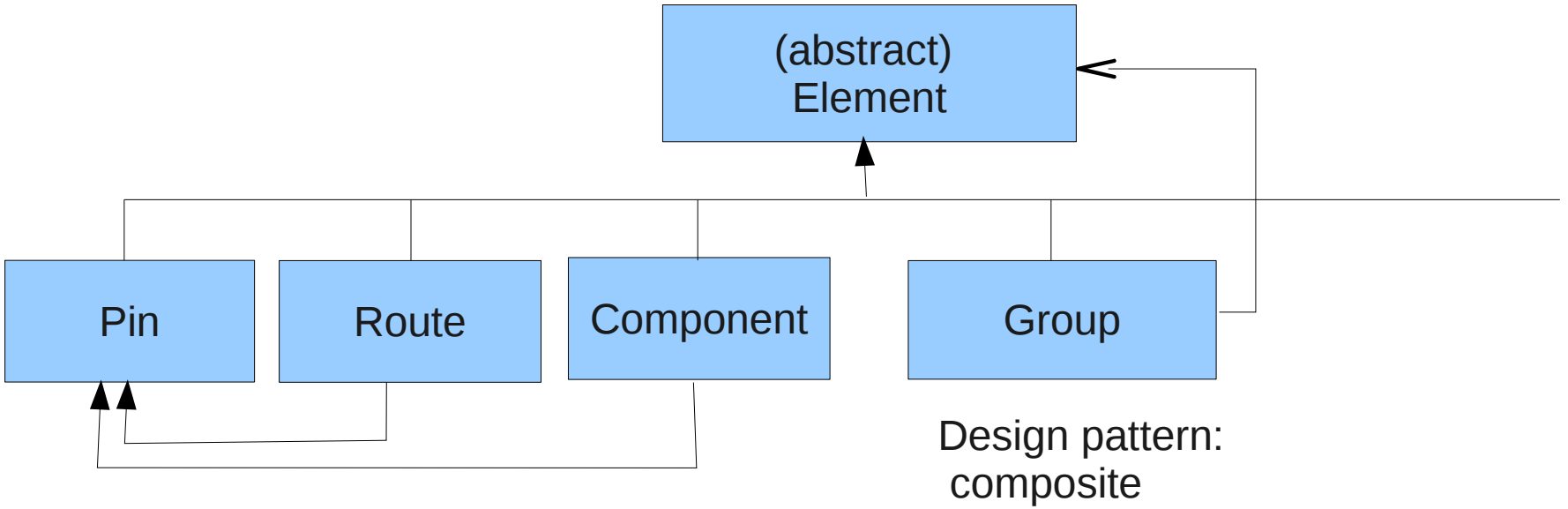


AST in (Unix) FileSystem



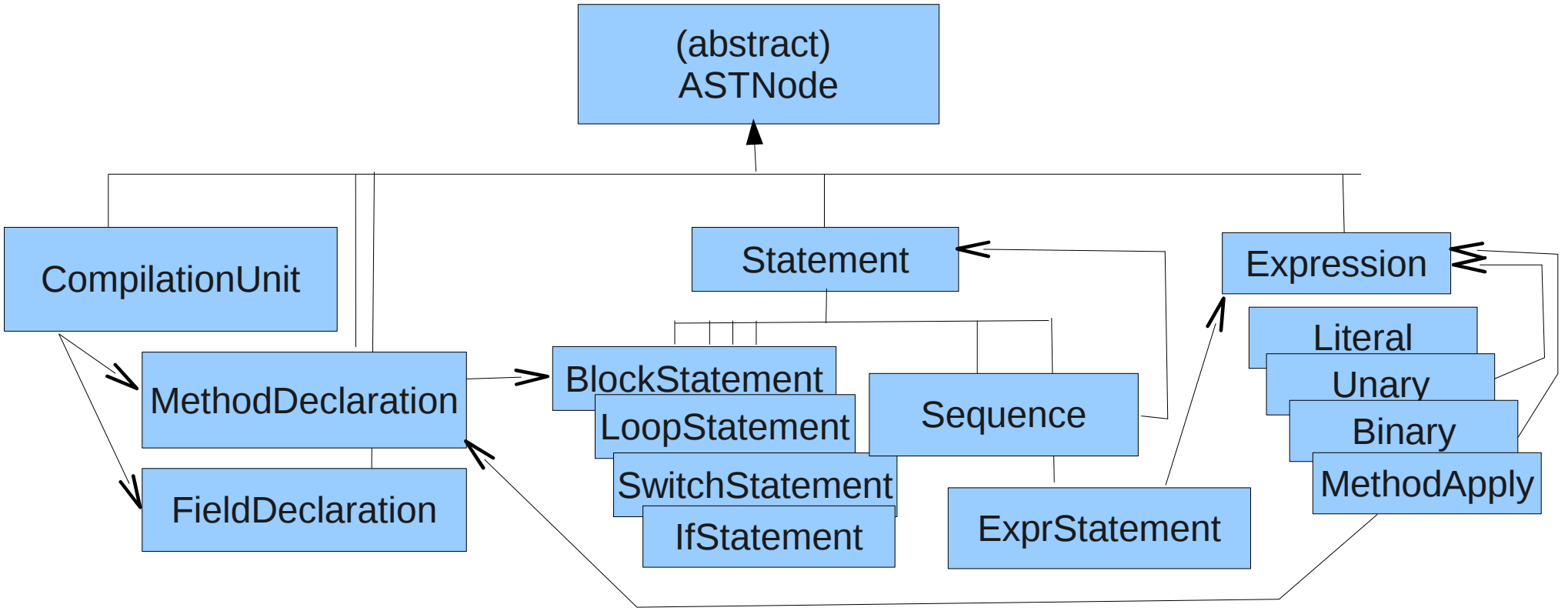
Design pattern:
composite

AST Example in Electronics



Design pattern:
composite

AST Example in Compilers

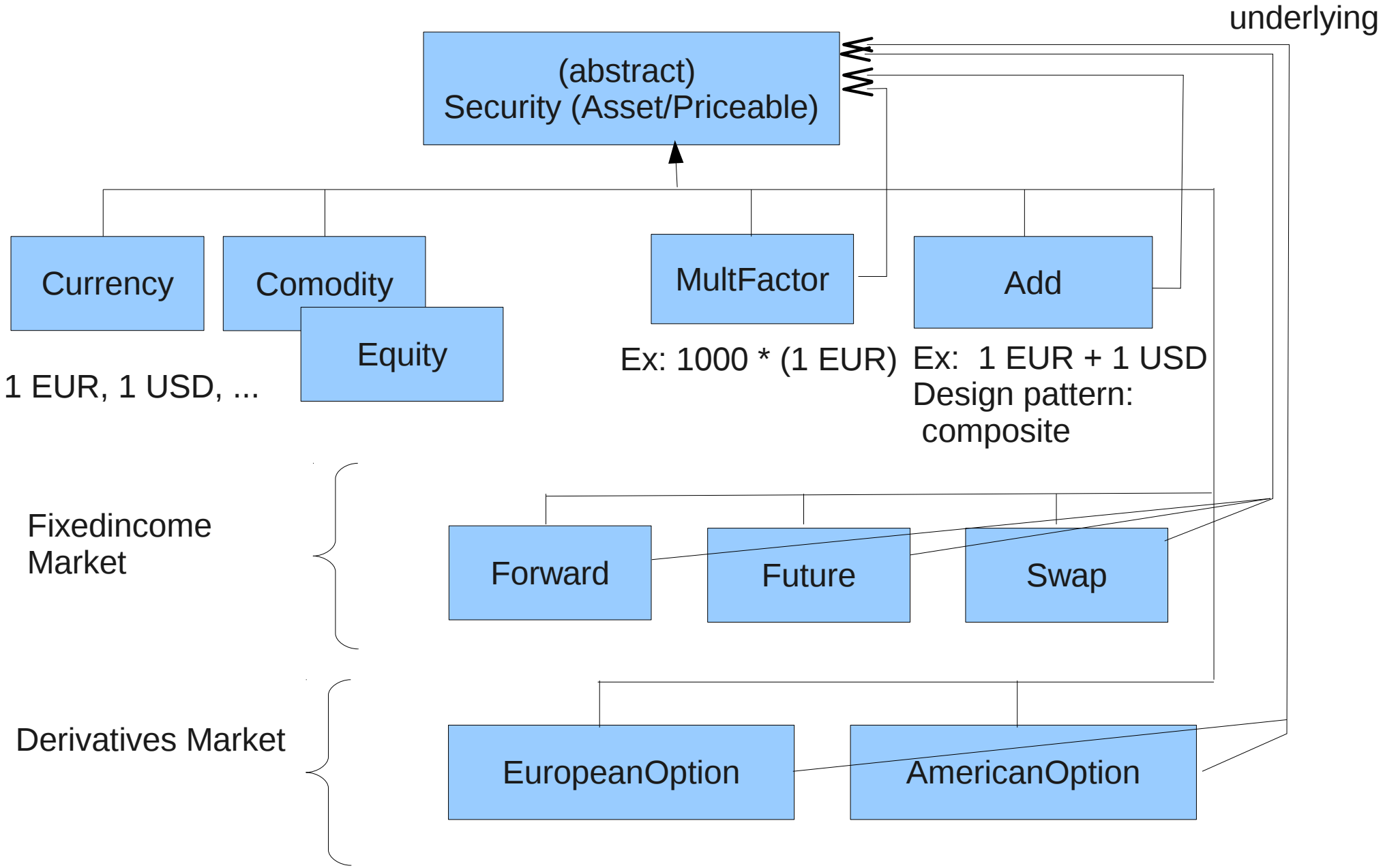


Declaration parts
(structure, symbols)

Statements parts => workflow
things ending with ';'
Wrappable with {}

Expression parts
things with value type
Wrappable with (..)

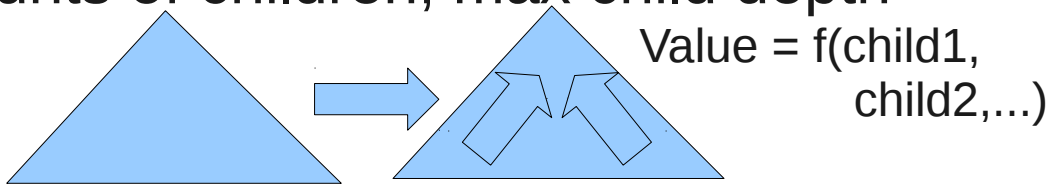
AST Example in Finance



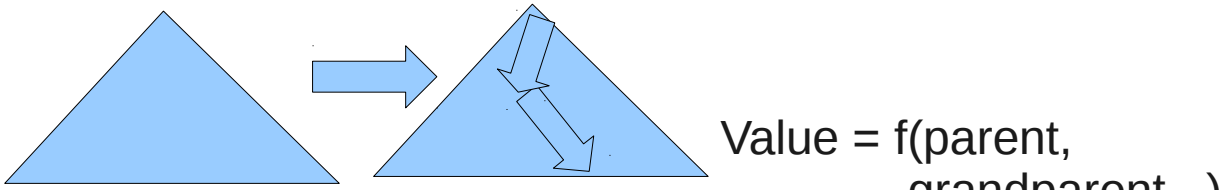
Types of Treatments on AST Trees

- AAST = Attributed – AST
  - **Synthetised** attributes =

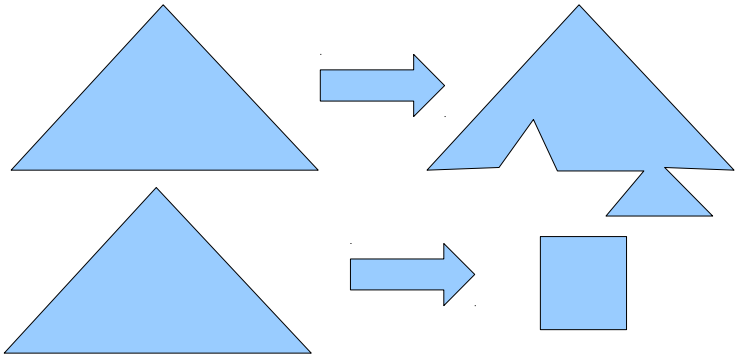
    example: recursive counts of children, max child depth



  - **Inherited** Attributes =
    example: level from root, index in parent, path, ...



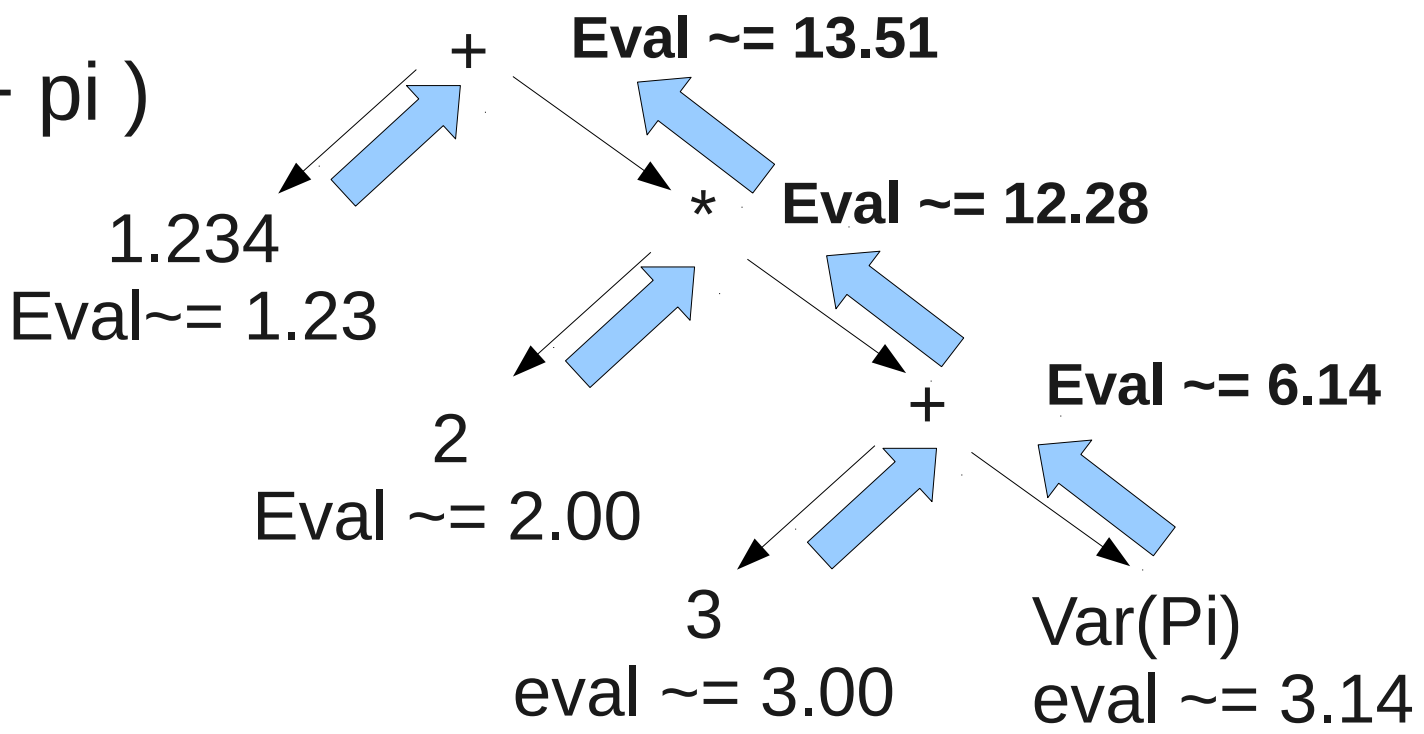
- **Transformations**
- **Front-End / Back-End** Transformations



Example of Synthetic Attribution value = Numerical Evaluation

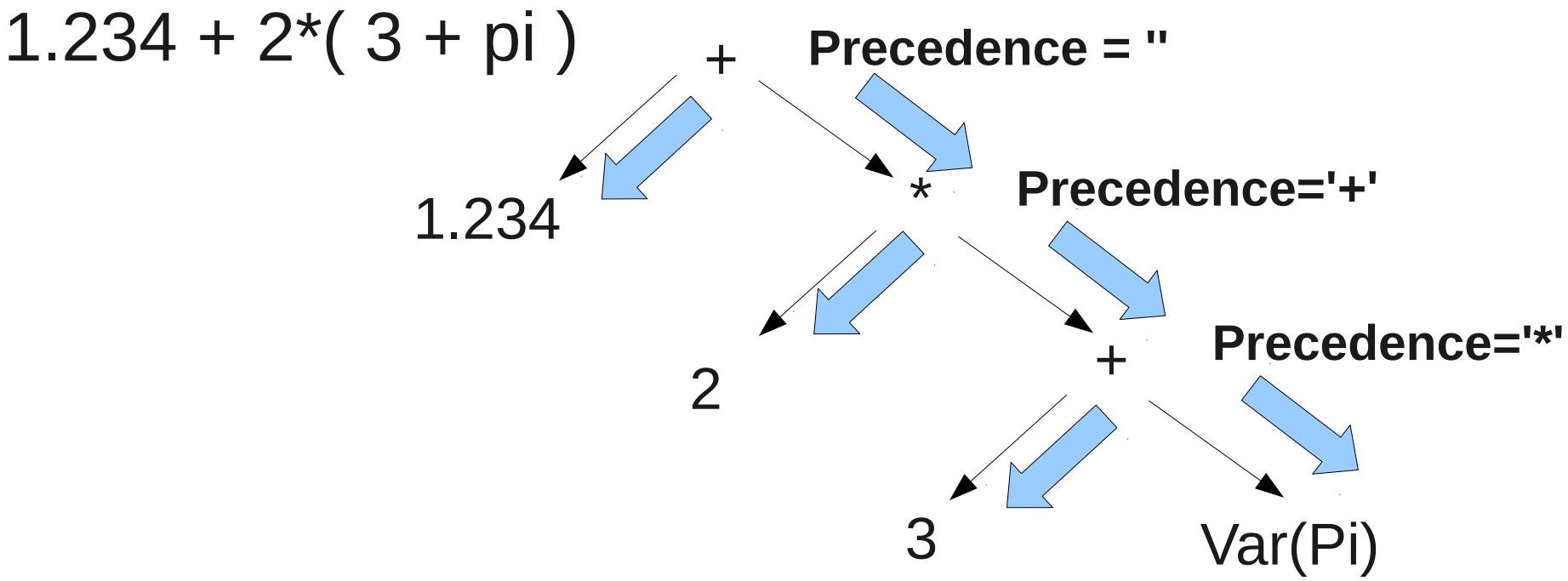
- Recursive child->parent traversal
- Synthetize from Child

1.234
+ 2 * (3 + pi)



Example of Inherited Attribution precedence = Parenthesis Order

- need for parenthesis?? compared to parent node and operator precedence : 1+ 2*3 != (1+2)*3
- Recursive parent->child traversal... Inherited from parent



Compiler Type-Checking

- Type – Checking in compilers :
  - Compute **inherited type**
    example : 1+2.3
    is of type (int)+(double) ==> double)
  - Compute expected **synthetized type**
    example : if(<<expr>>) { ..}
    <<expr>> is expected to be as boolean
  - **Check** that inherited type is **cohercible** with expected synthetized type

Treatments on AST

- Example of Transformation for Math-Expr:
  - Exact Constant Folding
    1+1 = 2 ==> replace x*(1+1)) by x*2
  - Partial Numerical Evaluation

    pi ~= 3.14 and => replace x*(1+pi) by ~x*4.14
  - Algebraic operation (distribute * over +, commute +, remove '+0', '*1' …)

- Pretty Printer … = Tree to Text representation

TP Step 2: Simple AST Treatments

1) Declare “public abstract double evalNumeric();” in abstract class Expression

2) Write Junit tests for numeric eval

3) Implement in sub-classes

4) Idem for “public abstract String prettyPrint();”

5) Write Junit for prettyPrint

6) Implements prettyPrint in sub-classes
(do not use parenthesis precedence
... over-parenthesis all)

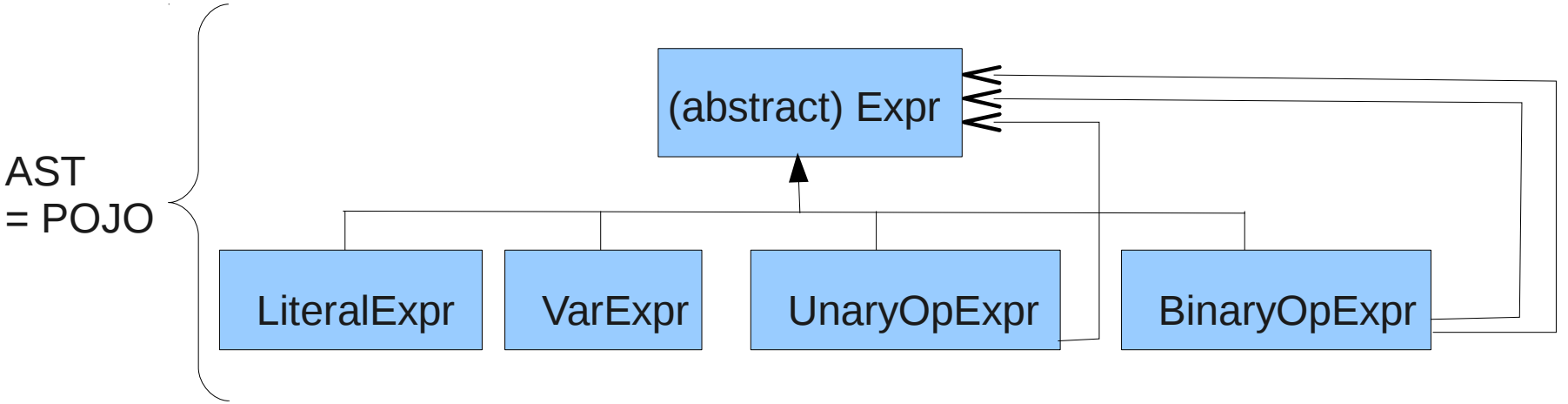
Extends for Adding Treatments... **Bad Design**

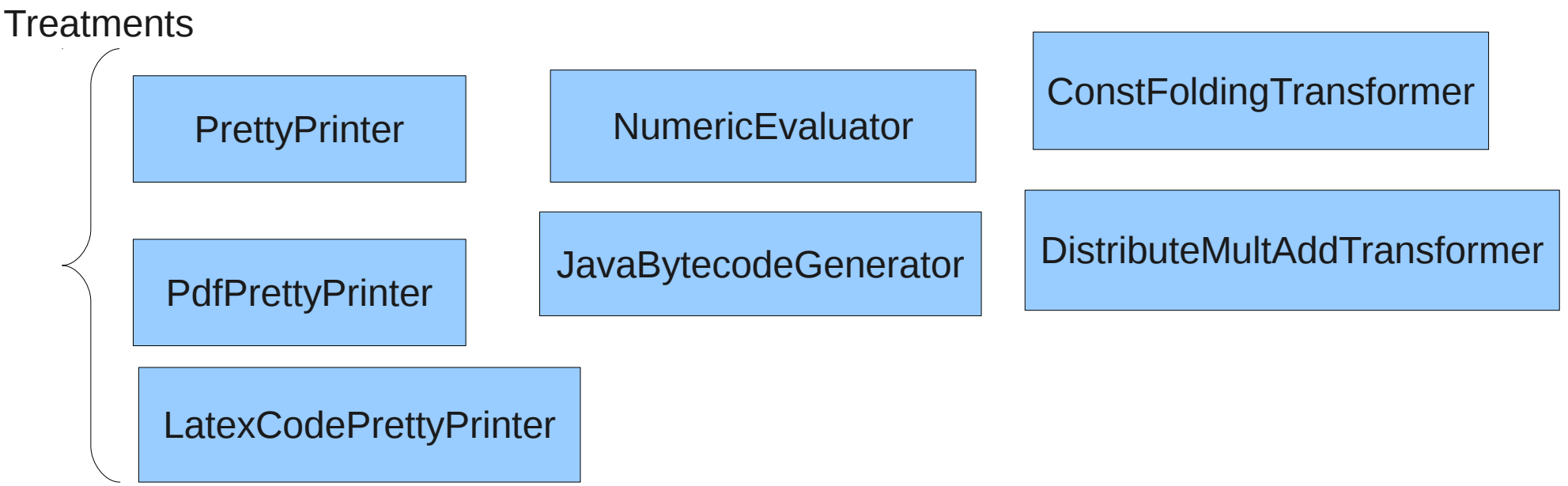
- KISS Principle: Keep It Simple Stupid
- GoF Patterns: implements interface, not extends class

- NEVER add Implementation codes in AST classes

- AST Classes = POJO (Plain Old Java Object)
  - Empty Constructor (+ optionaly full ctor)
  - Field with Getter + Setter
  - List with Add/Remove/iter
  - May add parent-child relationship / read-only support

Good Design : Write Treatment In Separate Classes





Problem: Object-Oriented Switch-Case

- Typical naïve code might look like:

```
class PrettyPrinter {
  public void print(Expression e) {
     if (e instanceof LiteralExpr) { …}
     else if (e instanceof UnaryOpExpr) { …}
     else if (e instanceof BinaryOpExpr) { …}
     else ...
  }
}
```

Solution : Design Pattern Visitor

- This Design Pattern is extremely Standard (Gof)
- Result in very generic tree traversal code
  (=> therefore its name “Visitor”)
- Might be also seen as an object-oriented
  “Switch / Case”
  - “Switch” lookup table in the abstract AST method
  - Concrete “case” in specialized sub-classes

Visitor Design Pattern

```
Interface ASTVisitor {
   public void caseA(A node);
   public void caseB(B node);
   public void caseC(C node);
   // … one type-checked method per sub-class in AST class hierarchy
}

Abstract class Expression {
   public void abstract accept(Visitor v);
   // **** Visitor Pattern MAGIC : abstract untyped switch
   // => call corresponding  type-check case method in concrete sub-classes ****
}


class XYZExpression  extends Expression {
   @Override public void accept(Visitor visitor) {
      visitor.caseXYZ(this);
   }
}
```

Sample PrettyPrinter Visitor

```
Public class ExprPrettyPrinter implements Visitor {

 @Override public void caseLiteral(Literal n) {
    print(n.getValue());
 }
 @Override public void caseUnaryExpr(UnaryOpExpression n) {
    print(n.getOperator() + “(“);
    n.getExpr().accept(this);
    print(“)”);
 }
 @Override public void caseBinaryExpr(Literal n) {
    print(“(“);
    n.getLefhHandSideExpr().accept(this);
    print(“)” + n.getOperator() + “(“);
    n.getRightHandSideExpr().accept(this);
    print(“)”);
 }
}
```

Theory => TP Steps 3 !!

1) Define the Visitor interface,
   and accept()/case() methods

2) Refactor code for Visitor pattern
   move methods “evalNumeric()”
   => in NumericEvalVisitor class

3) Also refactor the PrettyPrinter visitor
   optional: enhance to use Parenthesis Precedence

4) Zip project and send me as email

ANNEXES

Expression as Text

- Expression as text = sequence of chars
  - String text = “1 + 2 * (3+pi)”;
  - char[] textChars = new char[] { '1', ' ', '+', ' ', '2' ...}
- Readable?
  - String charabia = “unrecognized symbols”;
  - String badExpr = “1+++2***”; // not well-formed
  - String ambiguous = “1+2*5”; // op-precedence?
  - String semanticErr = “1 / 0”; // divided by 0

Text Grammar (LALR Grammar, LL)

- Sample grammar definition:

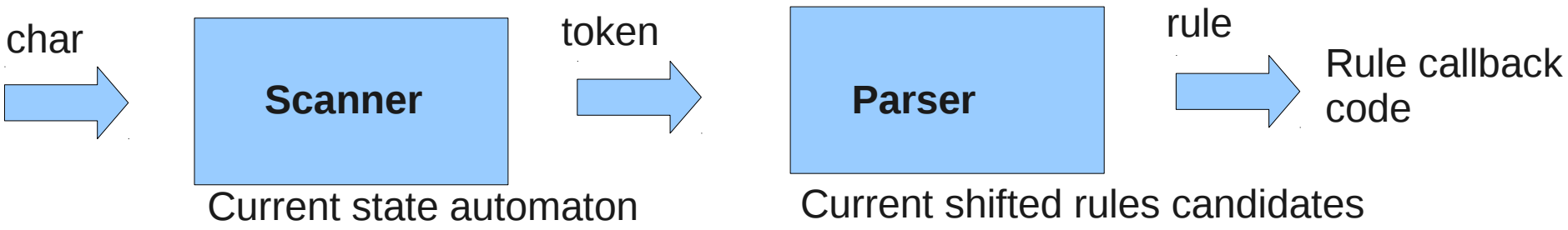
  tokens:= <intToken>, <doubleToken>,
  <plus>, <minus>, <mult>, , <exclamMark>...

  expr := literalExpr | varExpr | unaryExpr | binaryExpr;

  literalExpr := <intToken> | <doubleToken>;
  varExpr := <<name>>
  unaryExpr := <minus> expr | expr <exclamMark> | parentExpr;
  parentExpr := <leftParent> expr <rightParent>;
  binaryExpr := expr binaryOp expr;
  binaryOp := <plus> | <minus> | <mult> | ;

Text => Scanner => Parser => Code

- Historically: Lex & Yacc
- Lex = lexer = split chars into lexem (token/symbols)
- Yacc = Parse => recognize (shift/reduce) rules in sequence of lexems
- In Java: Javacc, ...



Sample Scanner/Parser Implementation

- Import library Javac, write “mygrammar.jj”
  Rules have a java code block:
  binaryExpr:= expr binaryOp expr {
  println(“parsed binaryExpr rule: ”
  + “left:”+ $1 + “operator:“+ $2 + “right:”+ $3);
  }
- JavaCC Code generator => “MyParser.java”
- usage:

  Scanner scanner = new Scanner(mytext);
  Parser parser = new MyParser(scanner);
  result = parser.parse();

Conclusion … Beautiful ASTs

Questions ?

arnaud.nauwynck@gmail.com