Lecture 3. ArithLang

September 7, 2018

ArithLang - Arithmetics

- Prefix, Infix, Postfix
- Arithlang Syntax and Semantics
- Interpreter (ANTLR, visit pattern, course project)

Prefix, Infix, Postfix

prefix: + 1 2 postfix: 1 2 +

infix: 1 + 2

the operands occur in the same order, and just the operators have to be moved to keep the meaning correct.

Prefix: Operators are written before their operands. Operators act on the two nearest values on the right. (Lisp, Scheme, The lambda-calculus uses prefix form and so we rewrite the body in prefix form)

Infix: Operators are written in-between their operands.

Postfix: Operators are written after their operands. Operators act on values immediately to the left of them.

conversion: You can convert directly between these bracketed forms simply by moving the operator within the brackets

5+(6*3)

(5+6) *3

*+563

3.9. Infix, Prefix and Postfix Expressions

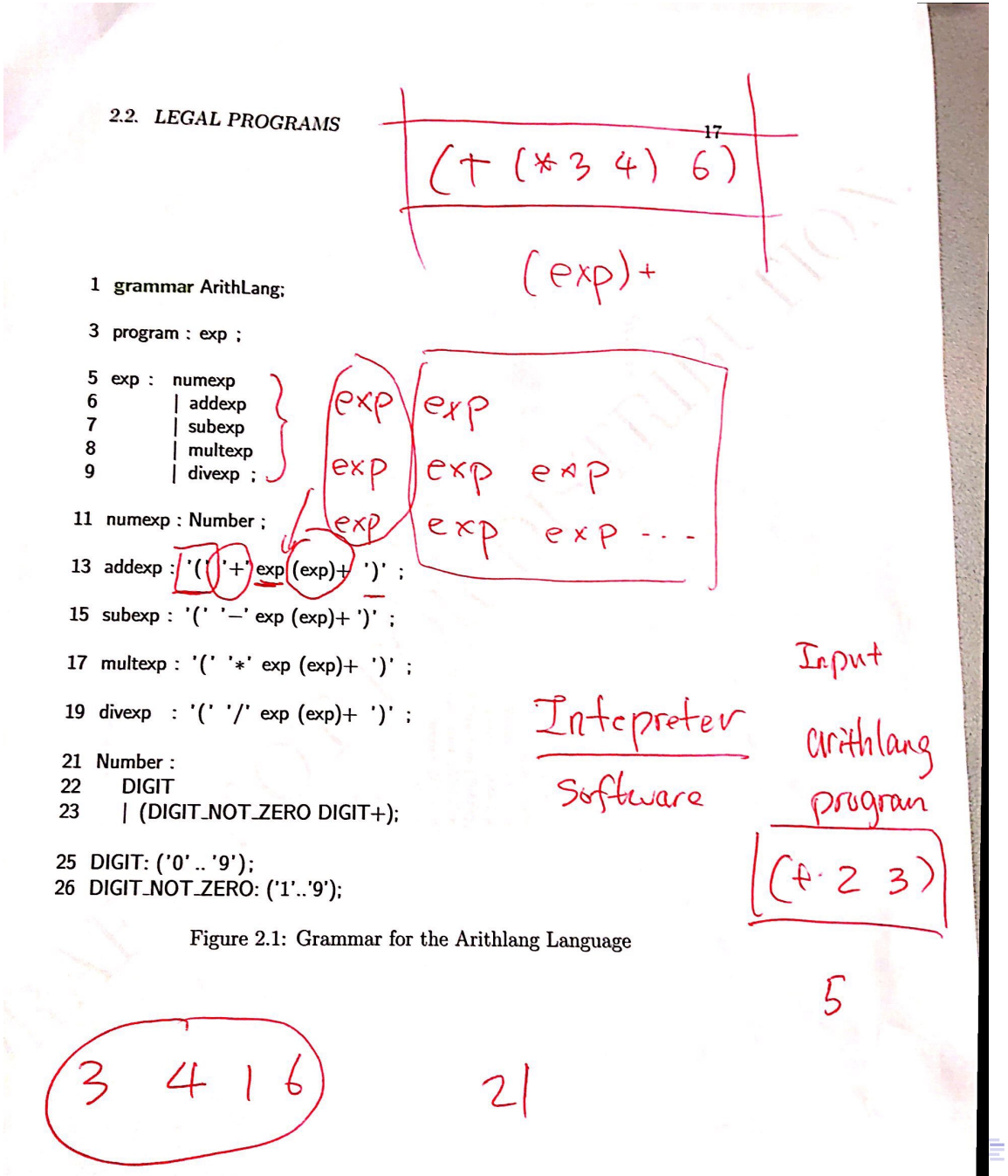
Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Prefix, Infix, Postfix

Both pre- and postfix have basically the same advantages over infix notation. The most important of these are:

- entirely unambiguous. Infix notation requires **precedence and associativity rules** to disambiguate it, or **addition of extra parentheses** that are not usually considered part of the notation. As long as the number of arguments to each operator are known in advance, both prefix and postfix notation are entirely unambiguous: "* + 5 6 3" is (5+6)*3, and cannot be interpreted as 5+(6*3), whereas parenthesis is required to achieve with infix.
- supports operators with different numbers of arguments without variation of syntax. **"unary-op 5" and "ternary-op 1 2 3"** both work fine, but need special syntax to make them work in infix.
- much easier to translate to a format that is suitable for direct execution

Syntax



2.2. LEGAL PROGRAMS 17

```
1  grammar ArithLang;

3  program : exp ;

5  exp :   numexp
6        | addexp
7        | subexp
8        | multexp
9        | divexp ;

11 numexp : Number ;

13 addexp : '(' '+' exp (exp)+ ')' ;

15 subexp : '(' '-' exp (exp)+ ')' ;

17 multexp : '(' '*' exp (exp)+ ')' ;

19 divexp  : '(' '/' exp (exp)+ ')' ;

21 Number :
22    DIGIT
23    | (DIGIT_NOT_ZERO DIGIT+);

25 DIGIT: ('0' .. '9');
26 DIGIT_NOT_ZERO: ('1'..'9');
```

Figure 2.1: Grammar for the Arithlang Language

(+ (* 3 4) 6)

(exp)+

exp exp

exp exp exp

exp exp exp ...

Interpreter

Software

Input

Arithlang program

(+ 2 3)

5

3 4 1 6

21

Syntax

Rajan's textbook

`Program`	::=	`Exp`	*Program*
`Exp`	::=		*Expressions*
		`Number`	*Number (NumExp)*
	|	`(+ Exp Exp`$^+$`)`	*Addition (AddExp)*
	|	`(- Exp Exp`$^+$`)`	*Subtraction (SubExp)*
	|	`(* Exp Exp`$^+$`)`	*Multiplication (MultExp)*
	|	`(/ Exp Exp`$^+$`)`	*Division (DivExp)*
`Number`	::=	`Digit`	*Number*
	|	`DigitNotZero Digit`$^+$	
`Digit`	::=	`0` | `DigitNotZero`	*Digits*
`DigitNotZero`	::=	`1` | `2` | `3` | `4` | `5`	*Non-zero Digits*
		| `6` | `7` | `8` | `9`	

Interpreter – Read Phase

1. lexical analysis,
2. parsing and constructing parse tree,
3. abstracting parse tree and store it in **(abstract syntax tree (AST)**
   An Example

Interpreter – Evaluate Phase

- Objective: convert AST to value
- Process like recursive tree traversal: To find value of program (* 3 (+ 4 2)) find value of sub-expressions 3 and (+ 4 2)
- PL semantics is essentially a systematically defined recursive traversal strategy:
  - What to do at a AST node?
  - How to traverse sub AST nodes?
- Completeness: Traversal must be defined for each kind of AST nodes, otherwise programs will get stuck.

Interpreter – Print phase

Value to String

Implementation of Interpreter

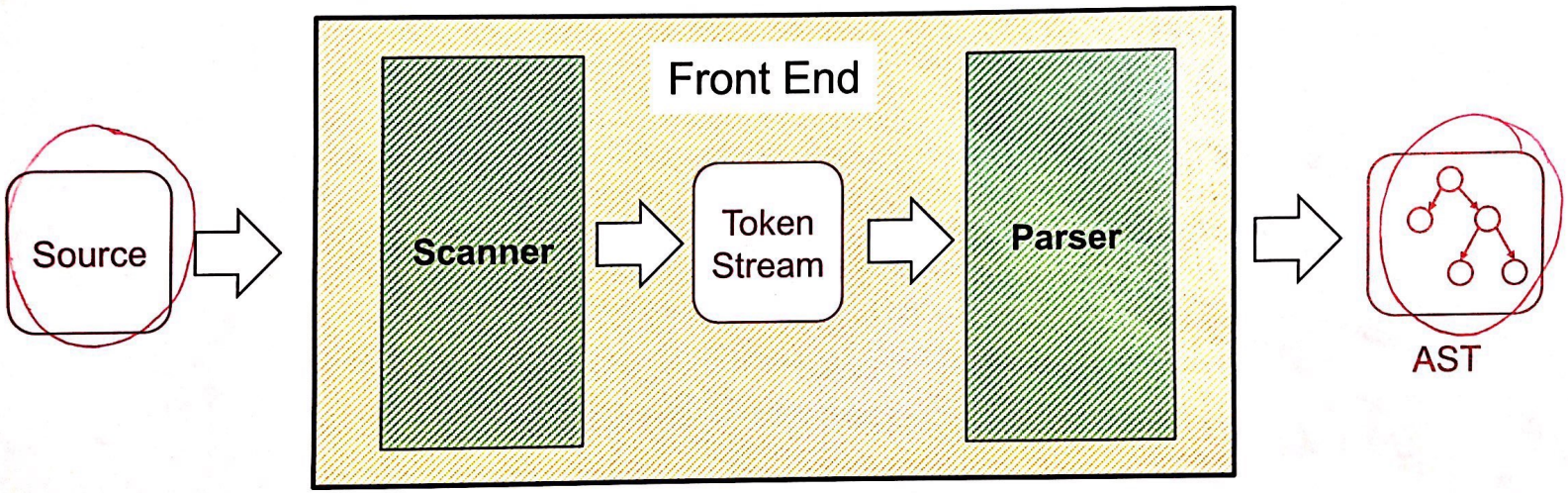
```
public class Interpreter {
  public static void main(String[] args) {
    Reader reader = new Reader();
    Evaluator eval = new Evaluator();
    Printer printer = new Printer();
    try {
      while (true) { // Read-Eval-Print-Loop (also known as REPL)
        Program p = reader.read();
        Value val = eval.valueOf(p);
        printer.print(val);
      }
    } catch (IOException e) {
      System.out.println("Error reading program.");
    }
  }
}
```

Compiler and Interpreter

1. Both Compiler and Interpreter need to parse code
2. Compiler: translate the program in one programming language to binary code, it does not produce values
3. Interpreter: evaluate a program to its value or error (python notebook demo)
   - input: a program, output: a value or error
   - read, evaluate and print three steps

read phase

Front End – Scanner and Parser



- Scanner / lexer converts program source into tokens (keywords, variable names, operators, numbers, etc.) using regular expressions // Context-insensitive matching
- Parser converts tokens into an AST (abstract syntax tree) using context free grammars

Parsing with CFGs

- CFGs formally define languages, but they do not define an *algorithm* for accepting strings
- Several styles of algorithm; each works only for less expressive forms of CFG
  - LL(k) parsing
  - LR(k) parsing
  - LALR(k) parsing
  - SLR(k) parsing

Antlr

Antlr

- Tools exist for building parsers from grammars
  - JavaCC, Yacc, etc.

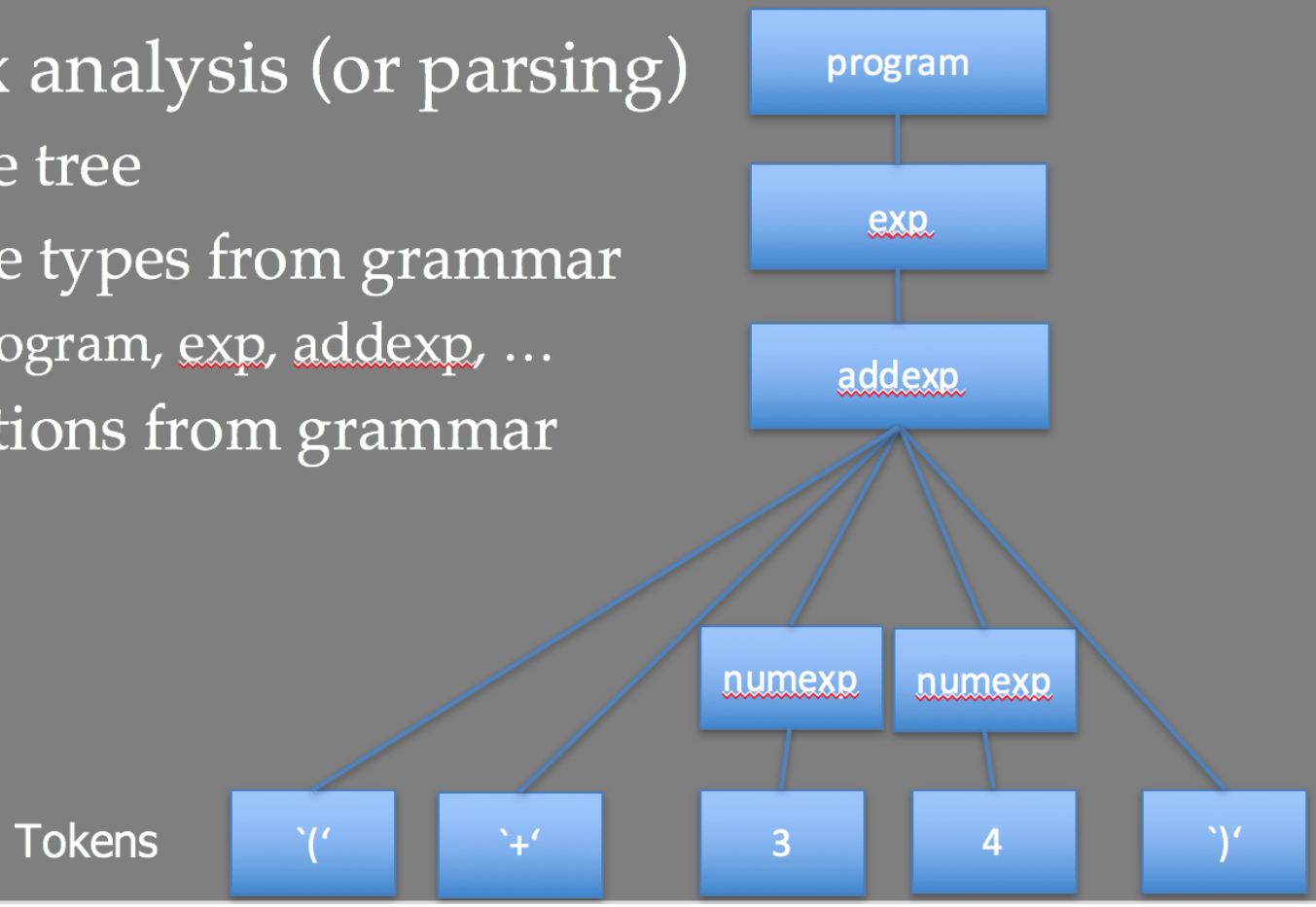
Arithlang Interpreter

Read Phase Example

- Objective: convert program (string) to AST
  - Example program: "(+ 3 4)"
- Lexical analysis
  - Divide the program into symbols (tokens)
  - `(', `+', DIGIT:3, DIGIT:4, and `)'
- Syntax analysis (or parsing)
  - Organize tokens into a tree (parse tree)
  - Structure of parse tree dictated by grammar.
- Abstraction
  - Convert parse tree to Abstract Syntax Tree (AST) by ignoring irrelevant tokens.

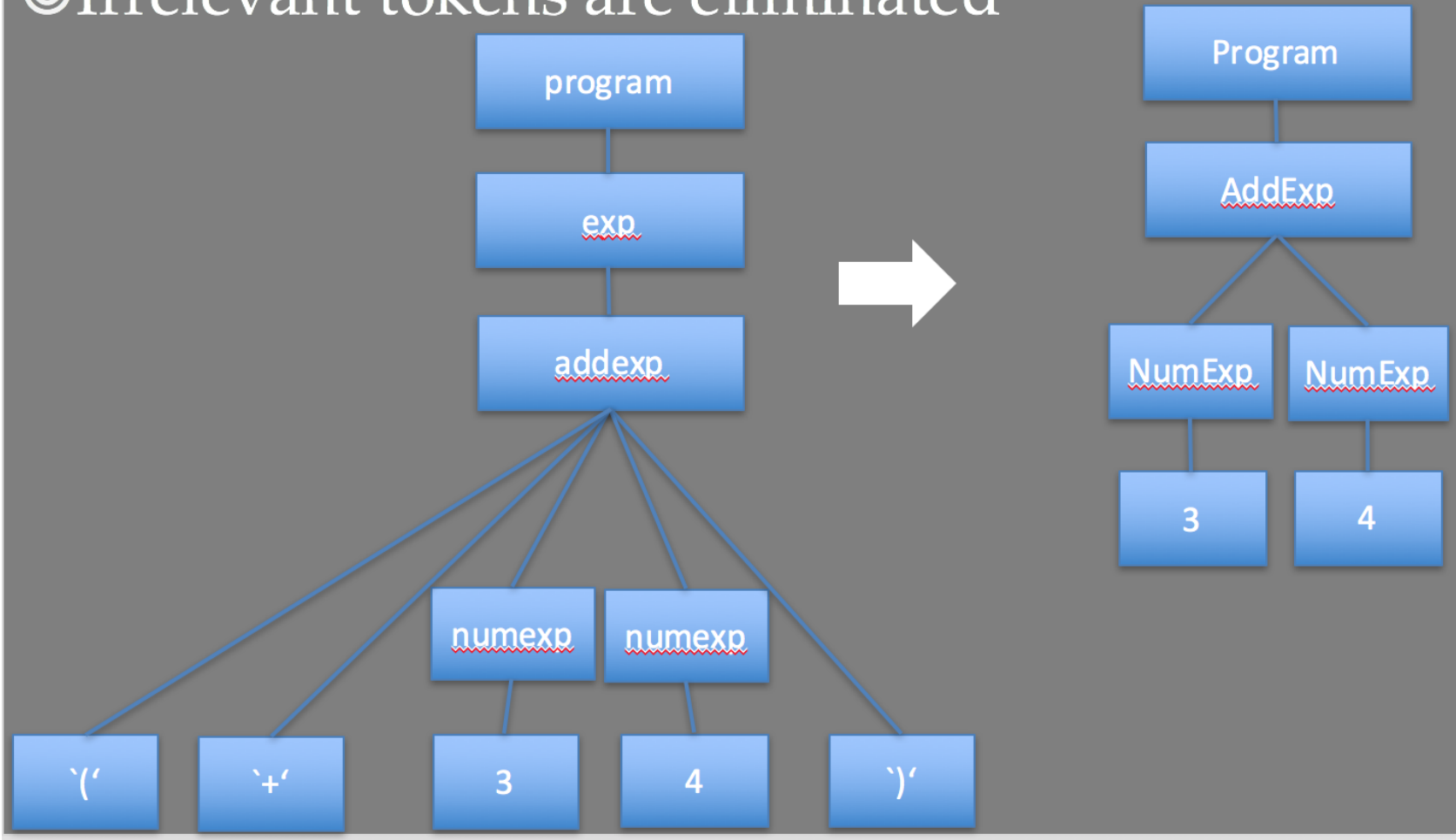
Read Phase Example

- Example program: "(+ 3 4)"
- Tokens: `(', `+', DIGIT:3, DIGIT:4, and `)'
- Syntax analysis (or parsing)
  - Parse tree
  - Node types from grammar
    - program, exp, addexp, …
  - Relations from grammar



Read Phase Example

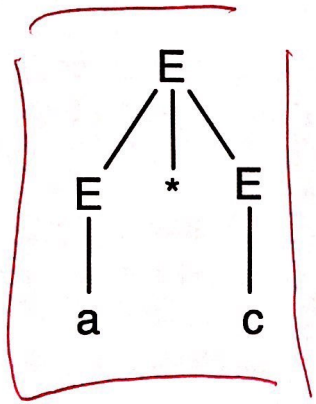
- Parse tree to AST
- Irrelevant tokens are eliminated



Abstract Syntax Trees

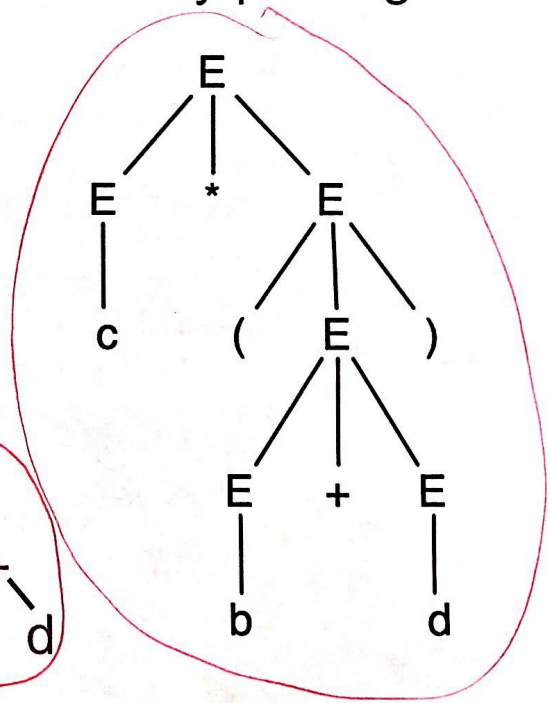
- A parse tree and an AST are not the same thing
  - The latter is a data structure produced by parsing

a*c



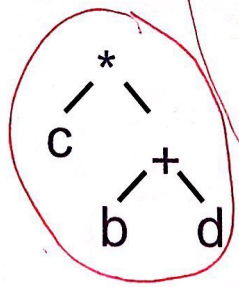
c*(b+d)

Parse trees



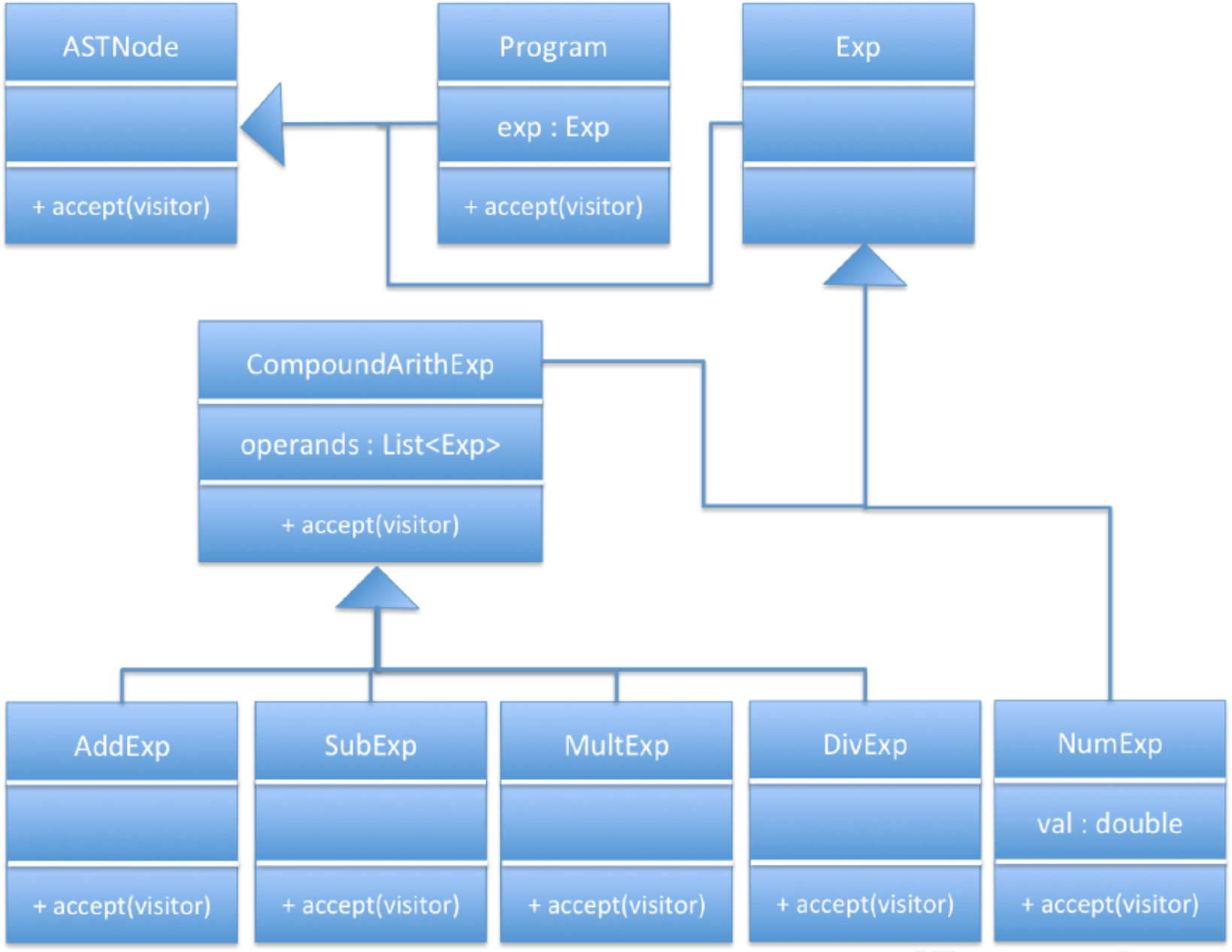
ASTs


```
Mult(Var("a"),Var("c"))
```




```
Mult(Var("c"),Plus(Var("b"),Var("d")))
```

Object-oriented Form of AST



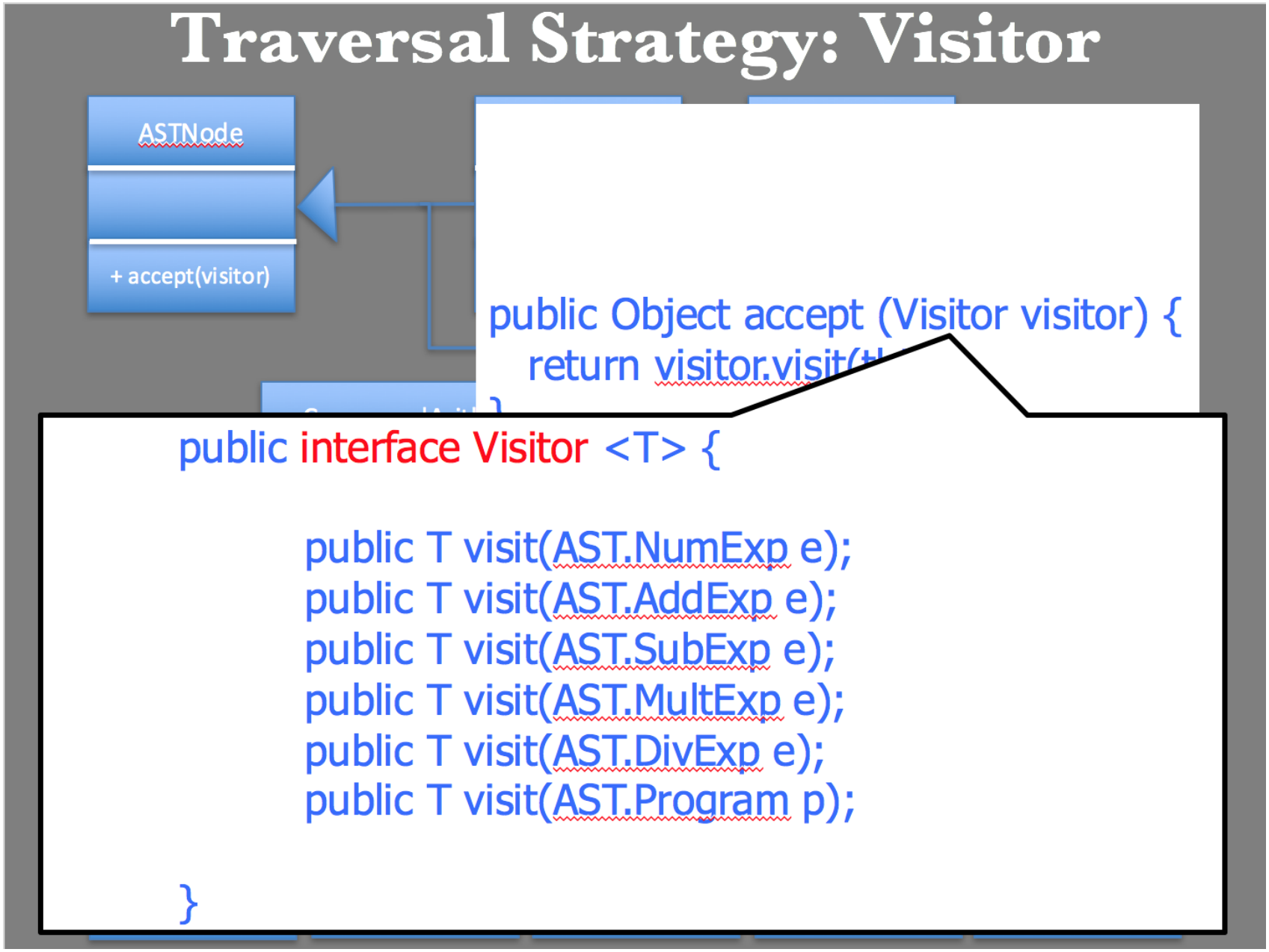
AST implementation

```
1   class ASTNode {
2     public abstract Object accept( Visitor   visitor );
3   }
4   class Program extends ASTNode {
5     Exp _e;
6     public Program(Exp e) { _e = e; }
7     public Exp e() { return _e; }

9     public Object accept( Visitor   visitor ) {
10      return  visitor . visit (this);
11    }
12  }
13  abstract class Exp extends ASTNode { }
14  class NumExp extends Exp {
15    double _val;
16    public NumExp(double v) { _val = v; }
17    public double v() { return _val ; }

19    public Object accept( Visitor   visitor ) {
20      return  visitor . visit (this);
21    }
22  }
```

Arithmetic Expression implementation

```
23  abstract class CompoundArithExp extends Exp {
24     List<Exp> _rep;
25     public CompoundArithExp(List<Exp> args) {
26        _rep = new ArrayList<Exp>();
27        _rep.addAll(args);
28     }
29     public List<Exp> all() {
30        return _rep;
31     }
32  }
33  class AddExp extends CompoundArithExp {
34     public AddExp(List<Exp> args) {
35        super(args);
36     }
37     public Object accept( Visitor visitor ) {
38        return visitor . visit (this);
39     }
40  }
```

Visitor Pattern



Visitor Pattern

```
1   class Formatter implements AST.Visitor<String> {
2     String visit(Program p) {
3       return (String) p.e().accept(this);
4     }
5     String visit(NumExp e) {
6       return "" + e.v();
7     }
8     String visit(AddExp e) {
9       String result = "(+";
10      for(AST.Exp exp : e.all())
11        result += (" " + exp.accept(this));
12      return result + ")";
13    }
14  ...
15  }
```

Figure 2.11: A code formatter: an example visitor

```
1   Exp exp3 = new NumExp(3);
2   Exp exp4 = new NumExp(4);
3   Exp exp2 = new NumExp(2);
4   List<Exp> expList = new ArrayList<Exp>();
5   expList.add(exp3);
```

Page 19

Rajan's Book


```
6   expList.add(exp4);
7   expList.add(exp2);
8   AddExp addExp = new AddExp(expList);
9   Program prog = new Program(addExp);
```

We can then try to find the formatted form of this program as follows.

```
10  Formatter f = new Formatter();
11  prog.accept(f);
```

(+ 3 4 2)

Value relation We will write the statement "value of a program" more precisely as the following mathematical relation.

```
value :  Program -> Value
```

Here, think of `value` as a mathematical function that takes the program AST node as an argument, e.g. `p`.

We will write the statement "value of an expression" more precisely as the following mathematical relation.

```
value :  Exp -> Value
```

Logical rules In describing the intended semantics of programming languages, we will often need to make statements like "B is true when A is true". We will state this relation as follows.

RELATION BIFA

$$\frac{A}{B}$$

Similarly, the relation "C is true when both A and B are true" is stated as follows.

RELATION CIFAANDB

$$\frac{A \quad B}{C}$$

We take the conjunction of the conditions above the line.

A relation "A is unconditionally true" is stated as follows.

RELATION AUNCONDITIONALLY

$$A$$

Semantics: Legal Value

Semantics: *style* $\rightarrow$ *informed*, *explanation*

`Value`	::=		*Values*
		`NumVal`	*Numeric Values*
`NumVal`	::=	(`NumVal n`), where `n` $\in$ the set of doubles	*NumVal*

Figure 2.12: The Set of Legal Values for the Arithlang Language

```
1  public interface Value {
2    public String toString();
3    static class NumVal implements Value {
4      private double _val;
5        public NumVal(double v) { _val = v; }
6        public double v() { return _val; }
7        public String toString() { return "" + _val; }
8    }
9  }
```

Semantics: Evaluate a Program

value of a program p is the value of its inner expression e

$$\textsc{Value of Program}\quad \frac{\texttt{value e = v}}{\texttt{value p = v}}$$

let p be a program and e be an expression

```
class Evaluator implements Visitor<Value> {
  Value valueOf(Program p) {
    // Value of a program is the value of the expression
    return (Value) p.accept(this);
  }
  ...
}

Value visit (Program p) {
  return (Value) p.e().accept(this);
}
```

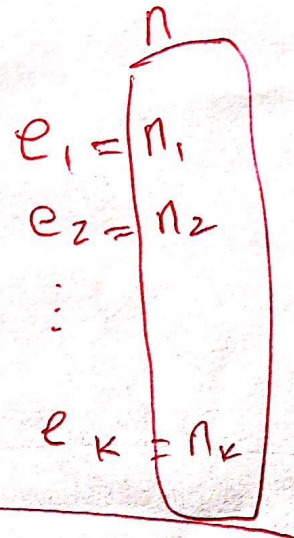
Semantics: Number

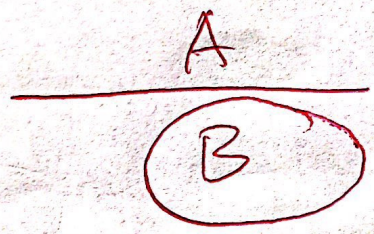
VALUE OF NUMEXP

```
value (NumExp n) = (NumVal n)
```

```
Value  visit (NumExp e) {
  return new NumVal(e.v());
}
```

Semantics: Addition





VALUE OF ADDEXP

$$\frac{\text{value } e_i = (\text{NumVal } n_i), \text{for } i = 0 \ldots k \quad n_0 + \ldots + n_k = n}{\text{value } (\text{AddExp } e_0 \ldots e_k) = (\text{NumVal } n)}$$

$e_1 + e_2 + \ldots + e_k = n$

```
Value visit (AddExp e) {
  List<Exp> operands = e.all();
  double result = 0;
  for(Exp exp: operands) {
    NumVal interim = (NumVal) exp.accept(this);
    result += interim.v();
  }
  return new NumVal(result);
}
```

Semantics: Subtraction

VALUE OF SUBEXP

$$\frac{\texttt{value } e_i = \texttt{(NumVal } n_i\texttt{)}, \text{for } i = 0...k \qquad n_0 - \ldots - n_k = n}{\texttt{value (SubExp } e_0 \ldots e_k\texttt{) = (NumVal n)}}$$

Implementation

```
public Value visit (SubExp e) {
  List<Exp> operands = e.all();
  NumVal lVal = (NumVal) operands.get(0).accept(this);
  double result = lVal.v();
  for(int i=1; i < operands.size(); i++) {
    NumVal rVal = (NumVal) operands.get(i).accept(this);
    result = result - rVal.v();
  }
  return new NumVal(result);
}
```

Semantics: Multiplication and Division

VALUE OF MULTEXP

$$\frac{\texttt{value } e_i \texttt{ = (NumVal } n_i\texttt{)}, \text{for i} = 0...\text{k} \qquad n_0 * \ldots * n_k = n}{\texttt{value (MultExp } e_0 \ldots e_k\texttt{) = (NumVal n)}}$$

VALUE OF DIVEXP

$$\frac{\texttt{value } e_i \texttt{ = (NumVal } n_i\texttt{)}, \text{for i} = 0...\text{k} \qquad n_0 / \ldots / n_k = n}{\texttt{value (DivExp } e_0 \ldots e_k\texttt{) = (NumVal n)}}$$