Lecture 3. ArithLang – Arithmetics

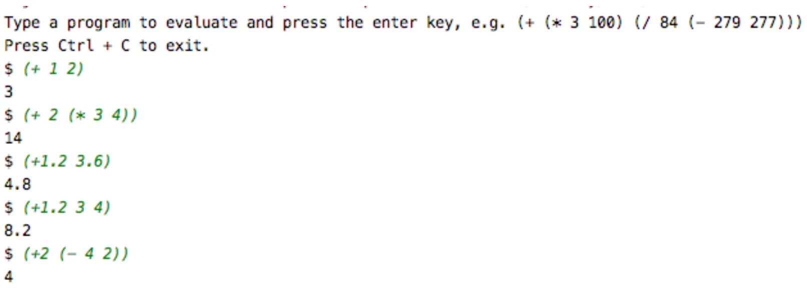
January 27, 2021

Overview

- A language that implements float/integer arithmetic
- Design decision: prefix, infix and postfix
- Interpreter: Java
- Arithlang syntax and semantics, and their implementations

Note. Syntax, semantics, design desicisions of ArithLang are taken from the language Scheme.

Interpreter Demo



```
Type a program to evaluate and press the enter key, e.g. (+ (* 3 100) (/ 84 (- 279 277)))
Press Ctrl + C to exit.
$ (+ 1 2)
3
$ (+ 2 (* 3 4))
14
$ (+1.2 3.6)
4.8
$ (+1.2 3 4)
8.2
$ (+2 (- 4 2))
4
```

Prefix, Infix, Postfix

- **Prefix**: Operators are written before their operands. Operators act on the two nearest values on the right. (Lisp, Scheme uses prefix forms)
  $+ + A * B\ C\ D$
  $* + A\ B + C\ D$
  $+ + 1\ 2\ 3$
- **Infix**: Operators are written in-between their operands.
  $1 + 2 + 3$
- **Postfix**: Operators are written after their operands. Operators act on values immediately to the left of them.
  $A\ B\ C * + D +$
  $A\ B + C\ D + *$
  $2\ 3\ 4 + * 5 *$
  $1\ 2 + 3 +$

Prefix, Infix, Postfix

Pre- and postfix have the same advantages over infix notation:

- unambiguous: infix notation requires precedence and associativity rules to disambiguate it, or adding extra parentheses. 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, 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.

Infix, Prefix and Postfix Expresssions

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 +

ArithLang Interpreter

- input: a program, output: a value or error
- read, evaluate and print three steps (see Interpreter.java)
- implemented a visitor design pattern

Interpreter – code overview

- Interpreter.Java:
  - entry point, the beginning of the program
  - eval.valueOf(p) is the entry point for Evaluator
- Evaluator.java:
  - Takes program, p as input and generates output
  - Each type of expression has a "visit" function in Evaluator
- AST.java:
  - Stores the data structure AST, each AST is a program that can be an expression
  - In other words, ASTs are the implementation of the parse trees
  - AST has different types of nodes starting with program, expression ...
  - Implementing the Visitor design pattern: visitor interface at line 100, input is different type of AST nodes and output is either a value (Evaluator) or a string (Formatter)
- Value.java:
  - defines types of the values in the programs

ArithLang Syntax and Its Implentation

Interpreter – Read Phase

- lexical analysis: identify tokens and their types
- parsing: constructing parse trees and convert them to **(abstract syntax tree (AST)**

Arithlang Syntax

```
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

- arithlang.g (antlr syntax): check antlr cheat sheet to understand the symbols in the grammar file
- antlr: parser generator
- you can see generated ArithLangLexer.java and ArithLangParser.java under build/generated-src/...
- some other useful links: entire document, locals

see some examples of grammar rules

ArithLang Semantics and Its Implentation

Interpreter – Evaluate Phase

- Goal: convert AST to value
- Process like recursive tree traversal: to find value of program "(* 3 (+ 4 2))", we first 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.

Arithlang Semantics: Legal Value

`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

```
public interface Value {
  public String toString();
  static class NumVal implements Value {
    private double _val;
      public NumVal(double v) { _val = v; }
      public double v() { return _val; }
      public String toString() { return "" + _val; }
  }
}
```

Interpreter – Evaluate Phase (visitor patterns)

```
interface Visitor <T> {
  T visit (NumExp e);
  T visit (AddExp e);
  T visit (MultExp e);
  T visit (SubExp e);
  T visit (DivExp e);
  T visit (Program p);
}
```

```
class Formatter implements AST.Visitor<String> {
  String visit (Program p) {
    return (String) p.e().accept(this);
  }
  String visit (NumExp e) {
    return "" + e.v();
  }
  String visit (AddExp e) {
    String result = "(+";
    for(AST.Exp exp : e. all ())
      result += (" " + exp.accept(this));
    return result + ")";
  }
 ...
}
```

Interpreter – Evaluate Phase (visitor patterns)

- public class Evaluator implements Visitor⟨*Value*⟩
- Value valueOf(Program p) {
  return (Value) p.accept(this); }
- public Value visit(Program p) {
  return (Value) p.e().accept(this); }
  public Value visit(AddExp e)
  public Value visit(NumExp e)
  ...

Semantics: Number

Value of NumExp

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

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

Semantics: Addition

$$\frac{\texttt{value } e_i \texttt{ = (NumVal } n_i\texttt{)}, \text{for } i = 0...k \qquad n_0 \texttt{ + } \ldots \texttt{ + } n_k \texttt{ = n}}{\texttt{value (AddExp } e_0 \ \ldots \ e_k\texttt{) = (NumVal n)}}\ \textsc{Value of AddExp}$$

```
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

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

```
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

$$\frac{\texttt{value}\ \texttt{e}_i\ \texttt{=}\ \texttt{(NumVal}\ \texttt{n}_i\texttt{)},\ \text{for i} = 0...\text{k} \qquad \texttt{n}_0\ \texttt{*}\ \ldots\ \texttt{*}\ \texttt{n}_k\ \texttt{=}\ \texttt{n}}{\texttt{value}\ \texttt{(MultExp}\ \texttt{e}_0\ \ldots\ \texttt{e}_k\texttt{)}\ \texttt{=}\ \texttt{(NumVal n)}}$$

Value of DivExp

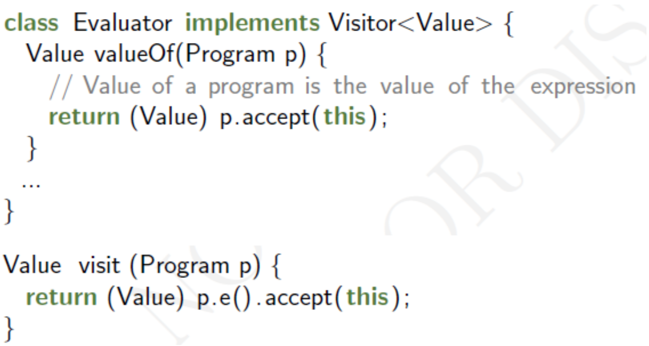
$$\frac{\texttt{value}\ \texttt{e}_i\ \texttt{=}\ \texttt{(NumVal}\ \texttt{n}_i\texttt{)},\ \text{for i} = 0...\text{k} \qquad \texttt{n}_0\ \texttt{/}\ \ldots\ \texttt{/}\ \texttt{n}_k\ \texttt{=}\ \texttt{n}}{\texttt{value}\ \texttt{(DivExp}\ \texttt{e}_0\ \ldots\ \texttt{e}_k\texttt{)}\ \texttt{=}\ \texttt{(NumVal n)}}$$

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);
}
```

Review

- ArithLang: a language only implements arithmetic
- Design decisions: infix, prefix, postfix
- Syntax: arithlang.g and its implementation in Antlr
- Semantics: formal inference rules and its implementation in Visitor patterns