

Lecture 3. ArithLang – Arithmetics

September 16, 2019

Overview

- ▶ A language that implements float/integer arithmetic
- ▶ Design decision: prefix, infix and postfix
- ▶ Interpreter: Java
- ▶ Arithlang Syntax and Semantics (how semantics are specified and implemented)

Note. Syntax, semantics, design decisions 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)

$++ A * B C D$ Note. Infix for this expression: $A + (B * C) + D$

$* + A B + C D$

$++ 1 2 3$

- ▶ **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.

$A B C * + D +$ Note. Infix for this expression: $A + (B * C) + D$

$A B + C D + *$

$2 3 4 + * 5 *$

$1 2 + 3 +$

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 Expressions

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$++A*BCD$	$ABC*+D+$
$(A + B) * (C + D)$	$*+AB+CD$	$AB+CD+*$
$A * B + C * D$	$+*AB*CD$	$AB*CD*+$
$A + B + C + D$	$+++ABCD$	$AB+C+D+$

ArithLang Interpreter

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

Interpreter – Read Phase

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

Code and Examples

Interpreter – Some Explanations

- ▶ Interpreter.Java: Entry point, eval.valueOf(p) is the entry point for Evaluator.
- ▶ Evaluator.java:
 - ▶ Takes program, p as input and generates output
 - ▶ Each type of expression will have a function to visit in Evaluator
 - ▶ During evaluation, depending on the current expression the dedicated accept function calls the visit of that expression
- ▶ 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, so on and so forth.
 - ▶ For 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:
 - ▶ Contains domain of the values
 - ▶ To define the semantics, you need to define the classes of legal values

Arithlang Syntax

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

Interpreter – Read Phase

arithlang.g (**Antlr Syntax**)

You can see generated ArithLangLexer.java and ArithLangParser.java
under build/generated-src/...

ANTLR Symbols

See also [Grammars](#) and [Special symbols in actions](#).

Symbol	Description
\$	Attribute
@	Action
==	action or dynamically-scoped attribute scope specifier
:	rule definition
;	end rule
	alternative
's'	char or string literal
*	wildcard
=	label assignment
+=	list label assignment
[...]	argument or return value spec
{...}	action
{{...}}	forced action; execute even while backtracking
{...}	subrule
+	1 or more
*	0 or more
?	optional or semantic predicate
~	match not
!	don't include in AST
^	make AST root node
=>	always execute predicate
->	rewrite rule
<token options>	token option spec like ID<node=VarNode>
*[...]	tree grammar or rewrite element
// ...	single-line comment
/ * ... */	multi-line comment

Keyword	Description
scope	Dynamically-scoped attribute
fragment	lexer rule is a helper rule, not real token for parser
lexer	grammar type
tree	grammar type
parser	grammar type
grammar	grammar header
returns	rule return value(s)
throws	rule throws exception(s)
catch	catch rule exceptions
finally	do this no matter what
options	grammar or rule options
tokens	can add tokens with this; usually imaginary tokens
import	import grammar(s)

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.

Arithlang Semantics: Legal Value

Value	::=	NumVal	<i>Values</i> <i>Numeric Values</i>
NumVal	::=	(NumVal n), where $n \in$ the set of doubles	<i>NumVal</i>

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 }
```

Interpreter – Evaluate Phase (visitor patterns)

```
1 interface Visitor <T> {  
2     T visit (NumExp e);  
3     T visit (AddExp e);  
4     T visit (MultExp e);  
5     T visit (SubExp e);  
6     T visit (DivExp e);  
7     T visit (Program p);  
8 }
```

```
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 }
```


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

VALUE OF ADDEXP

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

```
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

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

```
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

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

VALUE OF DIVEXP

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

Semantics: Evaluate a Program

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

$$\frac{\text{VALUE OF PROGRAM} \quad \text{value } e = v}{\text{value } p = v}$$

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