# ICSI 311 Assignment 4 – The Parser Part 3

**This assignment is extremely important – (nearly) every assignment after this one uses this one!**

**If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.**

**You must submit .java files. Any other file type will be ignored. Especially “.class” files.**

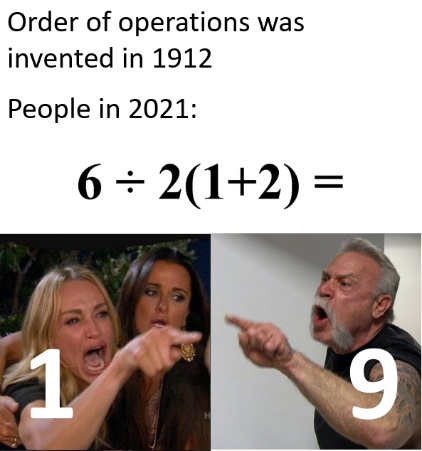
**You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.**

**You must submit every file for every assignment.**

***You must submit buildable .java files for credit.***

## Introduction

Order of operations is a critical part of building your parser. Let’s start by reviewing.

You’ve all seen these memes 🡪

These all revolve around understanding order of operations. Many of us learned in grade school:

PEMDAS (Please excuse my dear Aunt Sally):

Parenthesis

Exponents

Multiplication & Division

Addition & Subtraction

Of course, there are operations in C-like languages that this doesn’t take into account, like ++, --, etc. But let’s look at how we could deal with just this, to start.

The traditional way to deal with these in Parsers is the Expression-Term-Factor pattern:

Expression: TERM {+|- TERM}

Term: FACTOR {\*|/ FACTOR}

Factor: number | ( EXPRESSION )

A couple of notation comments:

| means “or” – either + or -, either \* or /

{} means an optional repeat. That’s what allows this to parse 1+2+3+4

Think of these as functions. Every CAPTIALIZED word is a function call.

Consider our meme expression above: 6/2\*(1+2)

We will start by looking at expression. Expression starts with a TERM. We can’t do anything until we resolve TERM, so let’s go there.

A term starts with FACTOR. Again, we can’t do anything until we deal with that.

A factor is a number or a (EXPRESSION). Now we can look at our token (hint: MatchAndRemove). We see a number. OK – our factor is a number. We “return” that. Remember that we got to factor from term. Let’s substitute our number in:  
TERM: FACTOR(6) {\*|/ FACTOR}

Now we deal with our optional pattern. Is the next character \* or /? Yes! Now is the next thing a factor?

It turns out that it is. Let’s substitute that in:  
TERM: FACTOR(6) / FACTOR(2)

But remember that our pattern is a REPEATING pattern (hint: loop):

TERM: FACTOR(6) / FACTOR(2) {\*|/ FACTOR}

We see the \* and call factor. But this time, the factor is not a number but a parenthetical expression.

Factor: number | ( EXPRESSION )

Factor calls expression.

Expression calls term, term calls factor, factor returns the number 1. Term doesn’t see a \* or / so it passes the 1 up. Expression sees the + and calls term. Term calls factor which returns the 2. Expression doesn’t see a +|- value so ends the loop and returns 1+2.

So, remember that we were here:

TERM: FACTOR(6) / FACTOR(2) \* FACTOR

Our factor is (1+2). That can’t be broken down. That math HAS to be done before can multiply or divide. That’s what enforces order of operations.

In code, this looks like something like this (pseudo-code):  
Node Factor()

num = matchAndRemove(NUMBER)

if (num.isPresent) return num

if (matchAndRemove(LPAREN).isPresent)

exp = Expression()

if (exp == null) throw new Exception()

if (matchAndRemove(RPAREN).isEmpty)

throw new Exception()

Node Term()

left = Factor()

do

op = MatchAndRemove(TIMES)

if (op.isEmpty) op=MatchAndRemove(DIVIDE)

if (op.isEmpty) return left

right = Factor()

left = MathOpNode(left, op, right)

while (true)

Node Expression()

left = Term()

do

op = MatchAndRemove(PLUS)

if (op.isEmpty) op=MatchAndRemove(MINUS)

if (op.isEmpty) return left

right = Term()

left = MathOpNode(left, op, right)

while (true)

What is this “MathOpNode”? It’s “just” a new node type that holds two other nodes (left and right) and an operation type: \*, /, +, -. Notice that the loops use the result of one operation as the left side of the next operation. This is called left associative – the left most part is done first. Right associativity is the opposite – the rightmost part is done first, then we work our way left.

Generally, notice the pattern here – we call a function that is the lowest level of precedence. That function uses results from a higher level of precedence. The lower level can’t do anything until the higher level is resolved. That "magic” is what enforces order of operation.

## Details

There are a LOT of levels of precedence in AWK:

|  |  |  |
| --- | --- | --- |
| **Expressions in Decreasing Precedence in *awk*** | | |
| **Syntax** | **Name** | **Associativity** |
| (*expr*) | Grouping (DONE!) | N/A |
| $*expr* | Field reference (DONE!) | N/A |
| lvalue ++  lvalue -- | Post-increment  Post-decrement | N/A  N/A |
| ++ lvalue  -- lvalue | Pre-increment (DONE!)  Pre-decrement | N/A  N/A |
| *expr* ^ *expr* | Exponentiation | Right |
| ! expr  + expr  - expr | Logical not (DONE!)  Unary plus (DONE!)  Unary minus (DONE!) | N/A  N/A  N/A |
| expr \* expr  expr / expr  expr % expr | Multiplication  Division  Modulus | Left  Left  Left |
| expr + expr  expr - expr | Addition  Subtraction | Left  Left |
| *expr* *expr* | String concatenation | Left |
| expr < expr  expr <= expr  expr != expr  expr == expr  expr > expr  expr >= expr | Less than  Less than or equal to  Not equal to  Equal to  Greater than  Greater than or equal to | None  None  None  None  None  None |
| expr ˜ expr  expr !˜ expr | ERE match  ERE non-match | None  None |
| expr in array  (index) in array | Array membership  Multi-dimension array membership | Left  Left |
| *expr* && *expr* | Logical AND | Left |
| *expr* || *expr* | Logical OR | Left |
| *expr1* ? *expr2* : *expr3* | Conditional expression | Right |
| lvalue ^= expr  lvalue %= expr  lvalue \*= expr  lvalue /= expr  lvalue += expr  lvalue -= expr  lvalue = expr | Exponentiation assignment  Modulus assignment  Multiplication assignment  Division assignment  Addition assignment  Subtraction assignment  Assignment | Right  Right  Right  Right  Right  Right  Right |

Source: <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/awk.html#tab41>

Last assignment, we created the two hardest methods. Now we will fill in the rest.

Start by removing the call in ParseOperation to ParseBottomLevel().

Going through the rest of the chart is easier. The next level is PostIncrement/Decrement (example: x++ or y--). The patterns are:

ParseBottomLevel() INC 🡪 Operation(result of ParseBottomLevel, POSTINC)  
ParseBottomLevel() DEC 🡪 Operation(result of ParseBottomLevel, POSTDEC)

else return ParseBottomLevel().

The right associative methods are a little more complex than the left. Consider:

2^3^4. Left associative would be: (2^3)^4. This is 4096.

2^3^4. Right associative would be 2^(3^4). This is 2^81 which is … very large.

There are two similar ways to build this – either use a Stack OR use recursion (which uses the built-in call stack).

The operations marked as “None” are terminal – there is no associativity because they can’t recur. You can’t have 3<4<5 or a ~ `hello` ~ `world`

Implement the rest of the chart by creating methods that follow the patterns above. A few hints:

1. expr is usually the next highest level of priority.
2. index and array are both the next highest level of priority (the matches).
3. Make sure to throw exceptions if the input is INVALID. That’s different from “not what I am looking for” in this method. +-\* is invalid, for example. You will see this when a Parse\_\_\_\_ returns an empty Optional when you expect a value.
4. Write good exception error messages to help you debug.
5. My solution for the previous assignment and this one together is about 400 lines of fairly repetitive code.

To simplify the assignments, I created an AssigmentNode (Node target, Node expression). But how do we handle something like: a+=5

I split this into two parts – AssignmentNode and OperationNode. I would create this as:

AssignmentNode (a, OperationNode(a + 5) )

We can now test with full expressions. Let’s leave ParseOperation public so that we can write unit tests against it.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rubric | Poor | OK | Good | Great |
| Code Style | Few comments, bad names (0) | Some good naming, some necessary comments (3) | Mostly good naming, most necessary comments (6) | Good naming, non-trivial methods well commented, static only when necessary, private members (10) |
| Unit Tests | Don’t exist (0) | At least one (3) | Missing tests (6) | All functionality tested (10) |
| Post Increment / Decrement | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Exponents | Doesn’t exist (0) | Attempted(5) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (10) |
| Factor | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Term | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Expression | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Concatenation | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Boolean Compare | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Match | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Array membership | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| AND | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Or | Doesn’t exist (0) | Attempted(3) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (5) |
| Ternary | Doesn’t exist (0) | Attempted(5) |  | Accepts tokens appropriately and generates OperationNode or returns partial result (10) |
| Assignment | Doesn’t exist (0) | Attempted(5) |  | Accepts tokens appropriately and generates AssignmentNode or returns partial result (10) |