ICSI 311 Assignment 4 – The Parser Part 2

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.

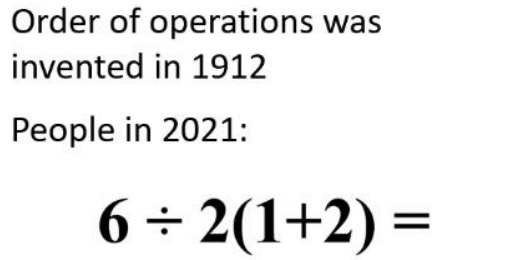
You must submit every file for every assignment.

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	N/A
$*expr*	Field reference	N/A
lvalue ++ lvalue --	Post-increment Post-decrement	N/A N/A
++ lvalue -- lvalue	Pre-increment Pre-decrement	N/A N/A
expr ^ *expr*	Exponentiation	Right

! expr + expr - expr	Logical not Unary plus Unary minus	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

These are a little tedious to write, but not hard. First, we need to create OperationNode. Normally, we call this "MathOpNode", but because there are so many operations in AWK that are not math, it seemed wise to rename it. Operation node has a Node left, an Optional<Node> right and a list of possible operations – use an enum for this. Mine are:

```
EQ, NE, LT, LE, GT, GE, AND, OR, NOT, MATCH, NOTMATCH, DOLLAR,
PREINC,POSTINC,PREDEC, POSTDEC,UNARYPOS, UNARYNEG, IN,
EXPONENT, ADD, SUBTRACT,MULTIPLY, DIVIDE,MODULO, CONCATENATION
```

Let's start with the lowest level. We notice that there are a few "key words" in these definitions. One is lvalue. This is "compiler speak" for a variable name. It has to be something that can be assigned to. That

means that it is either a variable reference(for example, x), an array reference(for example, y[3]) or a $ reference(for example, $6). Note that an array index can be an expression (like y[2+2]).

Make a VariableReferenceNode. It will have a name of the variable, and an optional Node that is the expression for the index.

Create new new methods:
Optional<Node>ParseBottomLevel()

Optional<Node> ParseLValue()

ParseLValue looks for these patterns:
DOLLAR + ParseBottomLevel() → OperationNode(value, DOLLAR)

WORD + OPENARRAY + ParseOperation() + CLOSEARRAY → VariableReferenceNode(name, index)

WORD (and no OPENARRAY) → VariableReferenceNode(name)

You might ask what ParseOperation() is – that is my name for the lowest level of the expressions. Why? Because this is valid:

a[x+=7] = “hello”

Now, we can look at our other method – ParseBottomLevel().

If you look at the first several layers, we can (except for exponentiation) combine them because we can tell by parsing what is happening. But there are a few other bottom level constants that we have to add in that aren’t EXPRESSIONS, so the expression chart leaves off. Your ParseBottomLevel should handle:
STRINGLITERAL → ConstantNode(value)

NUMBER → ConstantNode (value)

PATTERN → PatternNode(value)

LPAREN ParseOperation() RPAREN → result of ParseOperation

NOT ParseOperation() → Operation(result of ParseOperation, NOT)

MINUS ParseOperation() → Operation(result of ParseOperation, UNARYNEG)

PLUS ParseOperation() → Operation(result of ParseOperation, UNARYPOS)

INCREMENT ParseOperation() → Operation(result of ParseOperation, PREINC)

DECREMENT ParseOperation() → Operation(result of ParseOperation, PREDEC)

Else return ParseLValue()

ConstantNode and PatternNode are very simple – they are just AST nodes that hold one value.

These are the two longest and hardest methods.

For testing, ParseOperation() will call ParseBottomLevel(). Next assignment, we will be undoing that and adding all of the other levels.

For this week, make ParseOperation public so that we can test it independently from the rest of our parser. We cannot test full expressions yet, but we can test things like:

++a ++$b (++d) -5 `[abc]` e[++b] $7

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)
OperationNode	Doesn't exist (0)	Two of: Three of: Has enum, left and Optional right members, good constructors, ToString is good (3)	Three of: Has enum, left and Optional right members, good constructors, ToString is good (6)	Has enum, left and Optional right members, good constructors and ToString is good (10)
VariableReference Node	Doesn't exist (0)	One of: name and Optional index, good constructors, ToString is good (6)	Two of: name and Optional index, good constructors, ToString is good (6)	Has name and Optional index, good constructors and ToString is good (10)
Constant & Pattern Node	Doesn't exist (0)		Attempted (3)	Have name, good constructor and ToString is good (5)
ParseLValue – variables	Doesn't exist (0)	Either Accepts a variable name or creates an appropriate Variable Reference Node (3)		Accepts a variable name and creates an appropriate Variable Reference Node (5)
ParseLValue – arrays	Doesn't exist (0)	One of: Accepts a name, appropriately gets an index and creates an appropriate Variable Reference Node (3)	Two of: Accepts a name, appropriately gets an index and creates an appropriate Variable Reference Node (6)	Accepts a name, appropriately gets an index and creates an appropriate Variable Reference Node (10)
ParseLValue – dollar	Doesn't exist (0)	Attempted (5)		Creates an operation node, gets the value of the $ operator appropriately (10)

ParseBottomLevel – constants & patterns	Doesn't exist (0)			Detects strings, numbers and patterns and creates appropriate nodes (5)
ParseBottomLevel – parenthesis	Doesn't exist (0)		Creates an operation node OR gets the contents of the parenthesis appropriately (5)	Creates an operation node AND gets the contents of the parenthesis appropriately (10)
ParseBottomLevel – unary operators	Doesn't exist (0)	Two are correct (7)		All four are correct (15)