# Project 4. Operator precedence parsing

### [Learn You a H](http://learnyouahaskell.com/chapters)askell

### Videos: ([link to all Drake videos](https://www.youtube.com/playlist?list=PLAYqRAte9oRIChcPR_DD4uc8mCR6d3RiJ))

The following table shows the precedence of Haskell operators.

|  |  |  |  |
| --- | --- | --- | --- |
| Prec. | Left associative | Non-associative | Right associative |
|  | operators | operators | operators |
| 9 | !! |  | . |
| 8 |  |  | ^, ^^, \*\* |
| 7 | \*, /, `div`, |  |  |
|  | `mod`, `rem`, `quot` |  |  |
| 6 | +, - |  |  |
| 5 |  |  | :, ++ |
| 4 |  | ==, /=, <, <=, >, >=, |  |
|  |  | `elem`, `notElem` |  |
| 3 |  |  | && |
| 2 |  |  | || |
| 1 | >>, >>= |  |  |
| 0 |  |  | $, $!, `seq` |
|  |  |  |  |

Notice that exponentiation (^), multiplication (\*) and division (/), and addition (+) and subtraction (-) have precedence 8, 7, and 6 respectively. That’s why 3\*4+5^2 is evaluated as (3\*4)+(5^2) rather than (3\*(4+5))^2.

The higher the precedence, the “tighter” the binding: e.g., (\*) binds more tightly than (+).

The program in [this folder](https://drive.google.com/drive/folders/0B-I58s-_d3o5ekpnZE5TMzNuRUk?usp=sharing) uses operator precedence to parse an arithmetic expression expressed as string into a tree. The expression is limited to integers and the operators, +, -, \*, /, and ^. It then evaluates the tree to get a number. The program is organized as 6 Haskell modules. To run it, download all 6 files, load **ArithExprTests.hs** into GHCi and then run  
> testArithExprs "V1"

This runs a number of test cases, which are shown at the bottom of **ArithExprTests.hs**. The penultimate test case cannot be parsed. (It was deliberately written to be an error.)

The parsing is greatly simplified. The precedence values are not the same as in the table above, but the precedence order is the same. Parentheses are assumed to have lowest precedence.

After the tree is built, the program evaluates it and produces a numerical result.

You can also parse and evaluate a single expression, for example, as follows.

> V1.evalArith "3+4\*5"  
("3+4\*5",(3+(4\*5)),23)

The output is a triple:  
(<input string>, <fully parenthesized expression tree>, value)

Here’s a high level explanation of how the parser works.

1. Convert an initial input String into a list of elements and add extra parentheses around the entire thing. E.g., “3+4\*5” → [‘(‘, 3, ‘+’, 4, ‘\*’, 5, ‘)’]
2. Scan the list left-to-right looking for high-precedence operators embedded between lower-precedence or equal-precedence operators and reduce the high-precedence operator and its two arguments to a subexpression. (See example below.)

The parse breaks the input into three segments: left, window, and right. The window contains the portion of the input that is currently under examination.

1. Following are the steps to parse [‘(‘, 3, ‘+’, 4, ‘\*’, 5, ‘)’]. The final parse result will be an expression: (3 ‘+’ (4 ‘\*’ 5)). As a tree it appears as follows.

+

/ \

3 \*

/ \

4 5

[] [‘(‘, 3, ‘+’, 4, ‘\*’] [5, ‘)’] -- The starting configuration. Since ‘+’ has lower precedence than ‘\*’ we can’t do anything. Shift left. (The left-hand-list is initially [].)

[‘(‘] [3, ‘+’, 4, ‘\*’, 5] [‘)’] -- Still can’t do anything. Shift left again.

[‘(‘, 3] [‘+’, 4, ‘\*’, 5, ‘)’] [] -- Notice that 4, ‘\*’, 5 is between ‘+’ and ‘)’. Since ‘+’ and ‘)’ have lower precedence than ‘\*’, we can reduce 4, ‘\*’, 5 to a subtree with ‘\*’ at the root.

[‘(‘, 3] [‘+’, (4 ‘\*’ 5), ‘)’] [] -- Since we reduced three elements to one, shift two elements back to the right. (The expression (4 ‘\*’ 5) is one element of the list.)

([], [‘(‘, 3, ‘+’, (4 ‘\*’ 5), ‘)’]) -- Notice that 3, ‘+’, (4 ‘\*’ 5) now has parentheses around it. (The expression we just constructed is the right-hand argument to ‘+’.) Parentheses are considered to have essentially the lowest precedence. So construct a tree with ‘+’ at the root.

[] [‘(‘, (3 ‘+’ (4 ‘\*’ 5)), ‘)’] [] -- Discard the outer parentheses since they surround a single element.

[][(3 ‘+’ (4 ‘\*’ 5))] [] -- We’re done. All that’s left is a single expression. The parsed result is: (3 ‘+’ (4 ‘\*’ 5)).

1. To trace the parsing process, set shouldTrace to True at the last line of ParseStateClass.hs, and run evalArith on some input String. Instead of showing the three segments as lists, they are shown without the square brackets, and the window is shown surrounded by << and >>.

> V1.evalArith "2 + 3 \* 5 ^ 2"

<< (, 2, +, 3, \* >> 5,^,2,) -- Initial list of tokens

==> ( << 2, +, 3, \*, 5 >> ^,2,) -- Shift ( left.

==> (,2 << +, 3, \*, 5, ^ >> 2,) -- Shift 2 left.

==> (,2,+ << 3, \*, 5, ^, 2 >> ) -- Shift + left.

==> (,2,+,3 << \*, 5, ^, 2, ) >> -- Shift 3 left.

==> ( << 2, +, 3, \*, (5^2) >> ) -- Reduce 5 ^ 2 to (5^2)

-- and shift 2, +, 3 right.

==> (,2 << +, 3, \*, (5^2), ) >> -- Shift 2 left.

==> << (, 2, +, (3\*(5^2)), ) >> -- Reduce 3 \* (5^2) to (3\*(5^2)

-- and shift (, 2 right.

==> << (, (2+(3\*(5^2))), ) >> -- Reduce 2 + <expr> to (2+<expr>)

==> << (2+(3\*(5^2))) >> -- Drop extraneous parentheses

("2 + 3 \* 5 ^ 2", (2+(3\*(5^2))), 77) -- The result.

1. The first element of the tuple is the input string.
2. The second element is the fully parsed expression, i.e., the parse tree.
3. The third element is the value of that expression.

*Your job is to understand the program add your comments, and explain how it works.*

Note that ParseState\_v1 and ParseState\_v2 are two implementations of the ParseState Class. You may ignore ParseState\_v2 for the main part of Project 4.

**Extra credit**

1. Explain how ParseState\_v2 provides a second implementation of the ParseState Class.
2. Associativity is declared for each operator. But as written, the program ignores the declared associativity and assumes left associativity for all operators—even (^), which is normally right associative (see table above). Modify the program so that it takes associativity into account. As a test, evaluate an expression that uses ^ both before and after your modification.
   * 1. (^) treated as left associative: 4^3^2 = (4^3)^2 = 64^2 = 4096
     2. (^) treated as right associative: 4^3^2 = 4^(3^2) = 4^9 = 262144
3. Currently the system expects all operators to be binary. Add and explain a rule for unary plus or minus. (Treat them as having highest precedence.)  
   **Hint:** The easiest approach is to transform - <expr> to 0 – <expr> and + <expr> to 0 + <expr>.

> V1.evalArith "3+(-(5+6))"

("3+(-(5+6))",(3+(0-(5+6))),-8)

> V1.evalArith "3+(+(5+6))"

("3+(+(5+6))",(3+(0+(5+6))),14)

> V1.evalArith "3+(-5)"

("3+(-5)",(3+(0-5)),-2)

> V1.evalArith "3+(+5)"

("3+(+5)",(3+(0+5)),8)

> V1.evalArith "3 + - 5"

("3 + - 5",(3+(0-5)),-2)

> V1.evalArith "3 + + 5"

("3 + + 5",(3+(0+5)),8)

> V1.evalArith "3 - - 5"

("3 - - 5",(3-(0-5)),8)

> V1.evalArith "3 - + 5"

("3 - + 5",(3-(0+5)),-2)

> V1.evalArith "-3 \* -5"

("-3 \* -5",((0-3)\*(0-5)),15)

> V1.evalArith "+3 \* -5"

("+3 \* -5",((0+3)\*(0-5)),-15)

> V1.evalArith "-3 \* - - 5"

("-3 \* - - 5",((0-3)\*(0-(0-5))),-15)

> V1.evalArith "3 + -5\*6"

("3 + -5\*6",(3+((0-5)\*6)),-27)

> V1.evalArith "3 + +5\*6"

("3 + +5\*6",(3+((0+5)\*6)),33)

> V1.evalArith "3--5^2"

("3--5^2",(3-((0-5)^2)),-22)

> V1.evalArith "3-+5^2"

("3-+5^2",(3-((0+5)^2)),-22)

> V1.evalArith "5^--2"

("5^--2",(5^(0-(0-2))),25)

> V1.evalArith "5^++2"

("5^++2",(5^(0+(0+2))),25)

> V1.evalArith "16^+2"

("16^+2",(16^(0+2)),256)

Since we are assuming integer operations, negative exponents are not allowed—even if the result would be an integer.

> V1.evalArith "16^-2"

\*\*\* Exception: Negative exponent