

## 0.0.1 Parsing Algorithm

### 0.0.1.1 Least Significant Operator

An important part of the parsing stage, is the ability to find the least significant operator within an expression.

**Algorithm 1:** Least Significant Operator Position

```
1 function leastSigOperatorPos(String input):
2   int parenthesis = 0
3   int leastSigOperatorPos = -1 /* stores the position of the least significant operator
   so far */
4   int leastSigOpcode = 1000
5   char[] operators = ["+", "-", "/", "*", "^"] /* stores each operator in order of
   increasing significance */
6   int currentOpcode /* the current index of the operator in the array operators */
7   char currentChar
8   for i=0 to (input.size - 1) by 1 do
9     currentChar = input[i]
10    if currentChar in operators then
11      currentOpcode = operators.find(currentChar)
12      if (currentOpcode ≤ leastSigOpcode) and (parenthesis == 0) then
13        leastSigOperatorPos = i
14        leastSigOpcode = currentOpcode /* Update the least significant operator so
        far, as it is now the current character */
15    else if currentChar == "(" then
16      parenthesis++
17    else if currentChar == ")" then
18      parenthesis--
19    end
20  end
21  return leastSigOperatorPos
22 end
```

### 0.0.1.2 Removing Brackets

The function above assumes that there is no whitespace and that there are no brackets enclosing the entire expression (e.g.  $(x - 4)$ ). We can deal with our whitespace issue in our constructor<sup>1</sup> however we need to make another function check for and remove any brackets surrounding an expression.

However algorithm 2 has some issues. For example if we have the expression  $\frac{x+1}{x+2}$ , this would be input as,  $(x + 1)/(x + 2)$ . Now if we apply algorithm 2 which removes enclosing brackets we get  $x + 1)/(x + 2$ . This is completely wrong, as in this case we do not want to any remove brackets at all. The significant issue here is that we only want to remove the enclosing brackets, if they are *matching*.

To do this the algorithm 3 is more suited. This algorithm is based on 2 but checks for matching brackets. This algorithm will also throw an exception if there are unequal number of opening and closing brackets. This is so that we can inform the user later of the error that they have made and so that we can kill the process instantly rather than letting this error have consequences later on (probably during the evaluation of a value).

---

<sup>1</sup>In our function class we will store the original input so we can show the user, therefore we do not need to remove whitespace here

**Algorithm 2:** Check for and remove any Brackets surrounding an input

```

1 function checkBracket(String input):
2   Boolean done = False
3   while !done do
4     done = True
5     if (input[0] == '(' and (input[input.size - 1] == ')') then
6       done = False
7       input = input.subString(1, input.size - 2)
8     end
9   end
10  return input
11 end

```

**Algorithm 3:** Check for and remove any Matching Brackets surrounding an input

```

1 function checkBracket(String input):
2   Boolean done = False
3   while !done do
4     done = True
5     if input[0] == '(' and input[input.size - 1] == ')' then
6       int countMatching = 1
7       for i=1 to (input.size - 2) by 1 do
8         if countMatching == 0 then /* if countMatching is 0, then the matching
          closing bracket has been found before the end therefore we return the
          input without modification */
9           return input
10        else if input[i] == '(' then
11          countMatching++ /* If there is an opening bracket then we increment
          */
12        else if input[i] == ')' then
13          countMatching--
14          /* If there is a closing bracket then we decrement
15        end
16      end
17      if countMatching == 1 then /* we haven't looped til the last character which
          is ')' , and therefore at this point countMatching would be 1 for a
          standard expression */
18        done = False
19        input = input.subString(1, input.size - 2)
20      else
21        /* if not it is not an accepted expression, unequal brackets */
22        throw "There is an unequal number of opening and closing brackets"
23      end
24    end
25  return input
26 end

```

### 0.0.1.3 RegEx

Another important part of the parsing stage is to standardize the input. This is where we convert any inconsistencies discussed in section ??, page ?. The easiest way to do this is to use RegEx. RegEx stands for regular expression and is a standardized form of pattern recognition in strings, usually used during syntax analysis during compilation of software. Many languages support regex in some form or another and Java is no exception. These were our 5 inconsistencies that we needed to fix:

1. Any instance of  $ax$  where  $a \in \mathbb{R} : a \neq 0$  is to be converted to  $a * x$ .
2. Any instance of  $a($  and  $)a$  where  $a$  is not an operator, is to be converted to  $a * ($  and  $) * a$  respectively.
3. Any instance of  $(f(x))(g(x))$  is to be converted to  $(f(x)) * (g(x))$ .
4. Any instance of  $! - f(x)$  where  $!$  is to be any operator (e.g.  $*$  or  $/$ ) is to be converted to  $!(-f(x))$ .
5. Any instance of  $-f(x)$  at the start or next to an opening bracket is to be converted to  $0 - f(x)$ .

For the first and second examples, we look around every instance of  $x$ ,  $($  or  $)$  and if the adjacent characters are not operators or brackets then we replace with  $*x$  or  $x*$ . Therefore we can combine the first and second examples to use two separate RegEx expressions to deal with the case where we have  $x$  after and where we have  $x$  before.

The RegEx expression for the first case is `"([\+\-\*\\/\(\)\[\]])([x])"` with the replacement expression being `"$1*$2"`. If we take the RegEx expression, it creates two capture groups, `"$1"` and `"$2"`, which are `"([\+\-\*\\/\(\)\[\])"` and `"([x])"` respectively. A capture group stores a set of characters for each match that is made, so that we can perform actions on it later. The first capture group checks if the first character, in the substring that is currently being checked, is not any of the operators or brackets<sup>2</sup>. The not is signified by the first `\`. The second capture group checks if the second character, in the substring that is currently being checked, is an  $x$  or a  $($ . If both capture groups return true then a match is found and the match is replaced with `"$1*$2"` where `"$1"` is the first capture group, `"$2"` is the second capture group and the `*` asterisk between them signifying the multiply.

The RegEx expression for the second case is `"([x])([\+\-\*\\/\(\)\[\]])"` with the replacement expression being `"$1*$2"` again. This expression does the same as the first but checks for the reverse order i.e.  $xa$  and instead checks for a  $)$  instead of  $($  as we are checking the back of a substring instead of the start.

For the third inconsistency, the RegEx expression is `"\)\(""` with the replacement expression being `"*)*("`. This expression returns a match if it finds a  $)$  followed by a  $($ . If it finds a match it then replaces the entire match with `"*)*("`.

For the fourth inconsistency, the RegEx expression is `"([\+\-\*\\/\(\)\[\]])([\+\-\*\\/\(\)\[\]]*)"` with the replacement expression being `"$1(-$2)"`. The RegEx expression returns a match when there is an operator followed by a minus sign followed by any number of characters that are not operators or brackets. There are two capture groups. The first is `"([\+\-\*\\/\(\)\[\])"` and this captures the operator. The second is `"([\+\-\*\\/\(\)\[\]]*)"` and this captures the expression after the minus sign. The match is then replaced by the first capture group, followed by an opening bracket, a minus sign, the second capture group, then a closing bracket.

For the fifth inconsistency, the RegEx expression is `"(\[\()"` with the replacement expression being `"$10-"`. The RegEx expression returns a match when it is either the start of a line, signified by the `\`, or<sup>3</sup> a  $($  followed by a minus sign. The start of the line of bracket is captured and is used in the replacement expression, when the match is replaced with a  $0-$  preceded by either a bracket or nothing depending on if the start of a line or an opening bracket was captured.

<sup>2</sup>the reason there are so many backslashes is because a lot of the operators are actually key characters in RegEx and a backslash is an escape character which means that it signifies to treat the next character as a pure character

<sup>3</sup>the or keyword in RegEx is signified by `|`