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. If there are no operators -1 will be returned.

Algorithm 1: Least Significant Operator Position

```
function leastSigOperatorPos(String input):
    int parenthesis = 0
    int leastSigOperatorPos = -1 /* stores the position of the least significant operator
        so far                                                                          */
    int leastSigOpcode = 1000
    char[] operators = [“+”,’ “-”, “/”, “ * ”, “ ˆ”] /* stores each operator in order of
        incresing significance                                                          */
    int currentOpcode /* the current index of the operator in the array operators      */
    char currentChar
    for i=0 to (input.size - 1) by 1 do
        currentChar = input[i]
        if currentChar in operators then
            currentOpcode = operators.find(currentChar)
            if (currentOpcode ≤ leastSigOpcode) and (parenthesis == 0) then
                leastSigOperatorPos = i
                leastSigOpcode = currentOpcode /* Update the least significant operator so
                    far, as it is now the current character                             */
        else if currentChar == “(” then
            parenthesis++
        else if currentChar == “)” then
            parenthesis−−
        end
    end
    return leastSigOperatorPos
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[1] 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. It does this by using a variable that increments everytime there is a opening bracket and decrements every time there is a closing bracket. When the variable becomes 0 the matching bracket has been found. If this happens at the end the opening and closing brackets are removed, else nothing happens. This algorithm will also throw an exception if there are unequal number of opening and closing brackets. This is so that we can

[1]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

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).

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

```
function checkBracket(String input):
    Boolean done = False
    while !done do
        done = True
        if (input[0] == '(') and (input[input.size - 1] == ')') then
            done = False
            input = input.subString(1, input.size - 2)
        end
    end
    return input
end
```

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

```
function checkBracket(String input):
    Boolean done = False
    while !done do
        done = True
        if input[0] == '(' and input[input.size - 1] == ')' then
            int countMatching = 1
            for i=1 to (input.size - 2) by 1 do
                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                                        */
                    return input
                else if input[i] == '(' then
                    countMatching++/* If there is an opening bracket then we increment
                        */
                else if input[i] == ')' then
                    countMatching–
                    If there is an closing bracket then we decrement
                end
            end
            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                                                   */
                done = False
                input = input.subString(1, input.size - 2)
            else
                /* if not it is not an accepted expression, unequal brackets     */
                throw "There is an unequal number of opening and closing brackets"
            end
        end
    end
    return input
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[2]. 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 The expression returns a match if it finds a) folllowed 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.

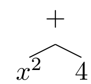
For the fifth inconsistency, the RegEx expression is **“(∧|\()-”** with the replacement expression being **“$10-”**. The RegEx expression returns a match when it is either the start of a line, signified by the ∧, or[3] a (followed by a minus sign. The start of the line of bracket is captured and is used in the replacement expression, when the match is replaced with a 0- preceeded by either a bracket or nothing depending on if the start of a line or an opening bracket was captured.

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

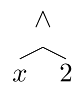
[3] the or keyword in RegEx is signified by |

0.0.1.4 Creating the Tree

Using the functions above we can create a syntax tree from our expression. We will do this using the Divide and Conquer methodology by recursively splitting the original expression until it becomes a single constant or variable. We can know if we should split the expression and if so where we should split the function by using our Least Significant Operator function. Our recursive case will return a tree where the left and right nodes are made up of the trees of the two sub-expressions and the root node will be the operator that we split our original expression with. Our base case will return a tree containing the constant or variable that is remaining. For example if we have the expression "$x^2 + 4$", we will first split this expression by the least significant operator which is the "+". We then make this our root node and our left and right trees will be what is left and right of that operator.



Now the left side has the operator $\wedge$ so we can create another tree for this sub-expression.



Now the left and right nodes of this tree have no operators so we create a tree of consisting of these variables and constants as the root node with no child nodes. So we have: