Algorithm to conver infix to postfix expression *

Alexis I. Aspauza Lescano[1][0000-0001-8685-010X] and Robson L. F. Cordeiro[1][0000-0002-6795-3004]

University of São Paulo – USP, Av. Trab. São Carlense 400, São Carlos, São Paulo, Brazil

Algorithm 1 Convert infix to postfix expression.

Input: c: Predicate in infix notation
Output: $postfixExp$: Expression in postfix notation

```
function ToPostfix(c)
    create stack pendingOps
    create vector postfixExp
    push "(" onto pendingOps
    add ")" to the end of c
    for each token extracted from c do
        if token is an operand then
            add token to postfixExp
        else if token is "(" then
            add token to pendingOps
        else if token is ")" then
            lastToken = pop from pendingOps
            while lastToken != "(" do
                add lastToken to postfixExp
                lastToken = pop from pendingOps
            end while
        else if token is an operator then
            while top of pendingOps is an operator and it has equal or higher precedence than token do
                lastToken = pop from pendingOps
                add lastToken to postfixExp
            end while
            add token to pendingOps
        end if
    end for
    return postfixExp
end function
```

Algorithm 1 defines a function to convert a predicate expressed in infix notation to another in postfix notation. This is useful in order to process then

* Supported by CAPES, FAPESP, USP.

the operations in the correct order. As an example of this, let's take the predicate from our case study: "DP.Category = SP.Category AND DP.Product = SP.Product AND (DP.Units ≤ SP.Units AND NOT (DP.Price < SP.Price) OR DP.Price ≥ SP.Price * 2)", which after executing the algorithm on it will produce the postfix expression as a vector of tokens: {DP.Category, SP.Category, =, DP.Product, SP.Product, =, AND, DP.Units, SP.Units, ≤, DP.Price, SP.Price, <, NOT, AND, DP.Price, SP.Price, 0.5, *, ≥, OR, AND}.

Lines 2 and 3 are part of the initialization, where we create a stack of pending operations and a vector to contain the postfix expression to be returned. Also, in the initialization, line 4 pushes an open bracket "(" onto the stack of pending operations, as well as line 5, adds the closing bracket ")" to the end of our input expression. Lines 6 to 24 are the main loop, which will get every single token extracted from the predicate (c). The current token being read will be processed according to its type. If the token is an operand (any constant or reference to a column table), it will be added directly to the postfix expression vector. In case the token is an opening bracket "(", it will be added to the stack of pending operations. But if the token is a closing bracket ")", we will pop tokens from the pending operations and add them to the postfix expression until an opening bracket is found, and the bracket will just be discarded. As the last option, if the token is an operator of any type (logical negation, arithmetic operator, logical operator, or logical connector), we will add to the postfix expression all top operators from the pending operations while the top is an operator with higher or equal precedence than the current token; and then, add the token to the stack of pending operations. The next operators are ordered by higher to lower precedence:

1. $\neg$
2. *, / both with same precedence
3. +, - both with same precedence
4. $<, \leq, >, \geq, =, \neq$ both with same precedence
5. $\wedge$
6. $\vee$

Finally, we will return our postfix expression vector.