**HOCHIMINH CITY UNIVERSITY OF TECHNOLOGY**
**FACULTY OF COMPUTER SCIENCE AND ENGINEERING**

**Course title : Discrete Structures for Computing**
**Course ID: CO1007**
**Academic Year: 2021-2022. Semester: 2**

**Lecturers:**
Trần Tuấn Anh, Nguyễn Tiến Thịnh, Nguyễn Văn Minh Mẫn

**Assignment**

# EVALUATING EXPRESSIONS

*(Deadline for submission: 23:59, 15/05/2022)*

## I. Introduction:

People usually evaluate a (arithmetic) expression by using its infix notation, where every operator lies in between two operands in the expression. For example, the expression "24/(6+2)-3" is evaluated to 0 because the sum of 6 and 2 in the parentheses equals to 8, the division "24/8" evenly equals to 3, and 3 is then subtracted from it, which results in 0. However, it's not an effective way for a computer to evaluate such an expression. The computers use the prefix or postfix notations instead.

### 1) Prefix, infix, and postfix notations:

Let $E$ be the expression "24/(6+2)-3". We study its prefix and postfix notations. By means of graph theory, $E$ can be represented by the following graph.
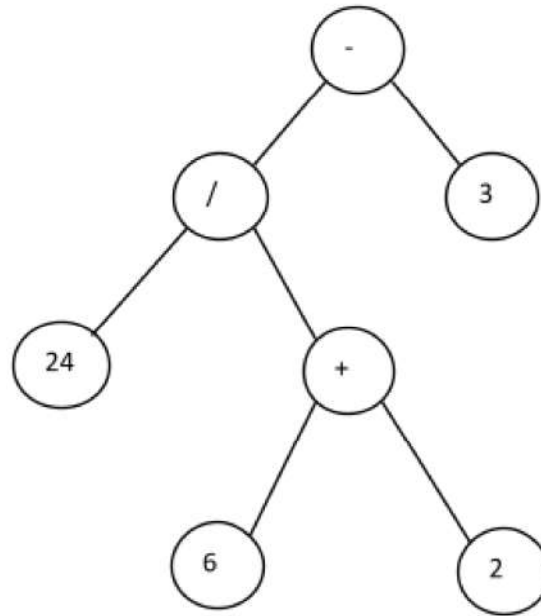
Fig. 1: Graph representation of the expression "24/(6+2)-3

Indeed, we can make *E* clearer by adding some extra parentheses to it. The expression *E* then becomes "((24/(6+2))-3)". Now we can represent the innermost sum as a binary tree with 3 nodes only, where the root of the tree is the addition operator and the two leaves are the numbers 6 and 2. This tree doesn't have internal nodes unless we consider the root as an internal node. By thinking that way, we can imagine that the division "24/(6+2)" can be also represented by a binary tree, where the root is the division operator, the left node of the root is the number 24 and the right-hand-side of the root is, however, no longer a single node with no child but a subtree representing the sum "6+2". We proceed and end up with a complete tree as in Fig. 1. With this representation, we can define the prefix and postfix notations for the expression *E* by performing the Pre-order and Post-order Tree Traversal algorithms respectively on the tree starting from the root. Here we consider non-recursive algorithms.

---

**Algorithm** PreoderTreeTraversal

---

**Input:** a binary tree

**Output:** an ordered list of the nodes of the tree

**Step 1:** Create an empty list and an empty stack.

Discrete Structures for Computing Assignment 212

**Step 2:** Push the root of the tree into the stack.

**Step 3:** Repeat the following steps until the stack is empty.

    **Step 3a:** Pop one node off the stack and add it to the list.

    **Step 3b:** Push the right-hand-side child of the node into the stack and then the left-hand-side one if they exist.

Tab. 1: Pre-order Tree Traversal Algorithm

| | | 24 | 6 | | |
|---|---|---|---|---|---|
| | / | + | 2 | 2 | |
| - | 3 | 3 | 3 | 3 | 3 |

Tab. 2: Stack behavior when we apply the algorithm PreorderTreeTraversal to the graph in Fig. 1

**Algorithm** PostoderTreeTraversal

**Input:** a binary tree

**Output:** an ordered list of the nodes of the tree

**Step 1:** Create an empty list and an empty stack.

**Step 2:** Push the root of the tree into the stack.

**Step 3:** Repeat the following steps until the stack is empty.

    **Step 3a:** Pop one node off the stack and add it to the list if it doesn't have children. Otherwise, go to Step 3b.

    **Step 3b:** Push the right-hand-side child of the node into the stack and then the left-hand-side one. Go to Step 3.

Tab. 3: Post-order Tree Traversal Algorithm

| | | | | 6 | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 24 | | 2 | 2 | | | | |
| | | + | + | + | + | + | | | |
| | / | / | / | / | / | / | / | | |
| | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | |
| - | - | - | - | - | - | - | - | - | - |

Tab. 4: Stack behavior when we apply the algorithm PostorderTreeTraversal to the graph in Fig. 1

For example, the behavior of the stack when we apply the algorithm PreorderTreeTraversal and PostorderTreeTraversal to the graph in Fig. 1 are given in Tab. 1, Tab. 3, and the resulting lists are "- / 24 + 6 2 3" and "24 6 2 + / 3 -" respectively.

**2) Evaluating arithmetic expressions:**

We discuss how a computer evaluates prefix/postfix-notation arithmetic expressions. Let's consider the expression "24 6 2 + / 3 -". The idea is to scan through the expression from left to right (from right to left if it's a prefix-notation expression). On the other hand, we use a stack to store only the operands.

---

**Algorithm** PostfixEvaluation

---

**Input:** a postfix-notation arithmetic expression

**Output:** the value represented by the expression

**Step 1:** Create an empty stack.

**Step 2:** Scan through the expression from left to right and repeat the following steps until the whole expression is scanned through completely.

    **Step 2a:** If it is an operand, push it into the stack. Otherwise, go to Step 2b.

    **Step 2b:** Pop 2 items off the stack, perform the operation, and add the resulting value to the stack.

**Step 3:** Return the last item added to the stack.

---

Tab. 5: Postfix Evaluation Algorithm

|     |     | 2   |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|     | 6   | 6   | 6   |     | 8   |     |     |     | 3   |     |
| 24  | 24  | 24  | 24  | 24  | 24  | 24  | 3   | 3   | 3   | 0   |

Tab. 6: Stack behavior when we apply the algorithm PostfixEvaluation to evaluate the expression "24 6 2 + / 3 -".

Firstly, we push the 3 consecutive numbers 24, 6, 2 into the stack. The plus sign is then encountered, we pop 2 and 6 off the stack, perform the addition "6+2" to obtain 8, and add 8 to the stack. After those steps, the division operator is encountered. Thus, we pop 8 and 24 off the stack and calculate the division "24/8", which equals to 3. The number 3 is then pushed into the stack and is taken off the stack together with the other number 3 because the minus sign is encountered. Finally, we calculate "3-3" and push 0 into the stack. This is also the last operation that we do. Therefore, the value represented by the expression is 0.

For expressions that haven't transformed into prefix or postfix-notation yet. We need to build their representing trees and perform the pre-order or post-order tree traversal algorithms before we apply the algorithm to evaluate the expression. If it is the case, the algorithm building the trees must take care of the precedence order. For example, the tree representing the expression "24/(6+2)-3", which is evaluated to 0, is different from the tree representing the expression "24/6+(2-3)", which is evaluated to 3. Moreover, we also need to be very careful about the negative numbers in the expressions. For example, the expression "6/(-3)+2" has the postfix-notation "6 3 - / 2 +", which may be considered as a critical case to the algorithm PostfixEvaluation to evaluate the expression.

## 3) Evaluating logical expressions:

In this assignment, we will modify the above algorithms to adapt to logical expressions. For example, the parse tree representing the infix-notation expression "p∧¬q→r∨s" is given in Fig. 2. By means of the algorithm PostorderTreeTraversal, the postfix-notation of the expression is "p q ¬ ∧ r s ∨ →". With this postfix notation and for specific values of propositions p, q, r, and s, for example, $p = 1$, $q = 0$, $r = 1$, and $s = 0$, we aim that the modified algorithm returns 1.

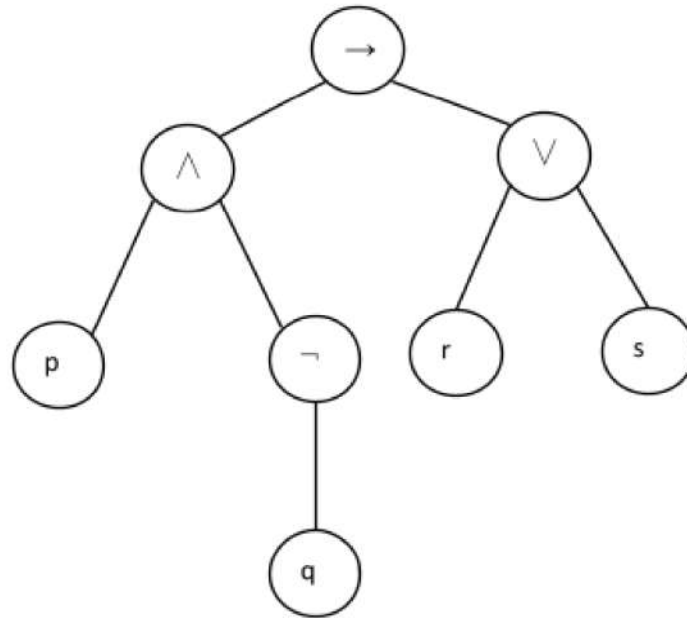Fig. 2: The parse tree representing the infix-notation expression "p∧¬q→r∨s"

## II. Exercises:

*Group by 4 students (go to https://tinyurl.com/2fpphu8x to list all of your team members) and do the following programming exercises. Note that the total score of the exercises is worth only 70% of the assignment. The remaining 30% is for the report. Students must write and explain everything that they will have done for the assignment carefully and coherently in the reports. Only team leaders submit the reports.*

Exercise 1 (4 pts):

a) Write a function that receives a constant string, which is an infix-notation arithmetic expression, and returns another string, which is the prefix-notation representation of the expression (1.25 pts).

b) Write a function that receives a constant string, which is an infix-notation arithmetic expression, and returns another string, which is the postfix-notation representation of the expression (1.25 pts).

c) Write a function that receives a constant string, which is either a prefix or a postfix notation arithmetic expression, and returns the value represented by the expression (1.5 pts).

Exercise 2 (6 pts):

a) Write a function that receives a constant string, which is an infix-notation logical expression, and returns another string, which is the prefix-notation representation of the expression (1.5 pts).

b) Write a function that receives a constant string, which is an infix-notation logical expression, and returns another string, which is the postfix-notation representation of the expression (1.5 pts).

c) Write a function that receives a constant string, which is either a prefix or a postfix notation logical expression, and a dictionary of variable names and their values. Returns the value represented by the expression corresponding to the given values of the variables (3 pts).

*The programs must be written using C++ and there are 6 .cpp files in total, respectively, the exercises 1a, 1b, 1c, 2a, 2b, and 2c.*

## III. Requirements for the input strings:

| Arithmetic operator | Meaning | Examples | Operand and output data type |
|---|---|---|---|
| * | Multiplication | 2 * 3 returns 6 | (Double, Double, Double) |
| / | Division | 1 / 2 returns 0.5, 2/0 raises divided-by-0 error | (Double, Double, Double) |
| + | Addition | 3 + 5 returns 8, 3 + -5 returns -2 | (Double, Double, Double) |
| - | Subtraction | 3 - 5 returns -2, 5-3 returns 2 | (Double, Double, Double) |
| ^ | Power | 4 ^ 2 returns 16, 0^0 raises undefined error | (Double, Double, Double) |

Tab. 7: Operators and data types of operands used in inputs of arithmetic expressions

| Type | Examples | Exceptions |
|---|---|---|
| Consecutive operators | 2//3, 4**5, 2^^4, 1^*3, 2*^8 raise undefined error | 1++1, 2+-5, 2-+3,--5, 3--4 |
| Precedence order | 2/3*2, 1-2+3 raise multiple-output error | 2+3*4, 4-2/3 |
| Parenthesis | (4(+3)-2, 4+)(3*2 raise syntax error | (4()+3)-2, (((2+3))-4) |

| Floating point | 3..4+2*2, 5+2.+2 raise syntax error | None |
|---|---|---|
| Blank | 2 4+3, 2    3 raise syntax error | 3   +   5   ^ 8 |

Tab. 8: Ambiguous and illegal input cases of arithmetic expressions

| Logical operator | Meaning | Examples (1 = true, 0 = false) | Operand and output data type |
|---|---|---|---|
| ∧ (use &) | Conjunction | 1∧1=1, 1∧0=0, 0∧0=0 | (Boolean, Boolean, Boolean) |
| ∨ (use \|) | Disjunction | 1∨1=1, 1∨0=1,0∨0=0 | (Boolean, Boolean, Boolean) |
| → (use >) | Implication | 1→1=1,1→0=0,0→0=1 | (Boolean, Boolean, Boolean) |
| ¬ (use !) | Negation | ¬1=0, ¬0=1, ¬¬1=1 | (Boolean, Boolean, Boolean) |
| ⊕ (use +) | Exclusive Or | 1⊕1=0,1⊕0=1,0⊕0=0 | (Boolean, Boolean, Boolean) |

Tab. 9: Operators and data types of operands used in inputs of logical expressions

| Type | Examples | Exceptions |
|---|---|---|
| Consecutive operators | p∧∧q, p∨∨q, p∨→q, p¬∧q raise undefined error | ¬¬p, p∧¬q, p→q→r |
| Precedence order | p∧q∨r,       p∨q∧r       raise multi-output error | p∧¬q, p∧q→r, p→q→r |
| Parenthesis | (p∧(q)∨r, (p∨q))(∧r raise syntax error | (p∧()q)∨r, ((p∧(q∨r)) |
| Blank | p    q, p∨→q       r raise syntax error | p    ∨ q→    r, pq (name of a proposition) |

Tab. 10: Ambiguous and illegal input cases of logical expressions