1085130

Part 1: Pseudocode

```
function ArithmeticExpressions(string)
    NUMBERS <- {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
    OPERATIONS <- {(, ), +, –, *, /}
    opsStack <- init(stack)
    numStack <- init(stack)
    for each character c in string do
        if c is in NUMBERS then
            numStack.push(c)
        else if c is in OPERATIONS then
            opsStack.push(c)
        if prev = ‘(‘ and c != ‘(‘ and c is not in NUMBERS then
            return “NotWellFormed”
        if c = ‘)’ and prev != ‘)’ and prev is not in NUMBERS then
            return “NotWellFormed”
        if c = ‘)’ then
            while opsStack.peek() = ‘)’ do
                if opsStack.size() < 3 or numsStack.size() < 2 then
                    return “NotWellFormed”
                b1 <- opsStack.pop()
                operator <- opsStack.pop()
                b2 <- opsStack.pop()
                if b2 = ‘(‘ and operator is in OPERATIONS\{(, )} then
                    num2 <- numStack.pop()
                    num1 <- numStack.pop()
                    if operator is ‘+’ then
                        numStack.push(num1 + num2)
                    else if operator is ‘– ‘ then
                        numStack.push(num1 – num2)
                    else if operator is ‘*’ then
                        numStack.push(num1 * num2)
                    else if operator is ‘/’ then
                        numStack.push(num1 / num2)
                else then
                    opsStack.push(b2, operator, b1)
                    numStack.push(num1, num2)
        prev <- c
    if opsStack.size() = 0 and numStack.size() = 1 then
        return numStack.pop()
    return “NotWellFormed”
```

Part 3(b):

Explain your algorithm, and explain why your algorithm runs in linear time. Clear pseudocode may help you convey your algorithm to the reader, but is not required for full marks for this part. You should not include very detailed pseudocode if you do decide to present this, just an overview of your algorithm.

The algorithm consists of these parts:

1. Construct an adjacency list for the graph.
2. Sort the graph topologically using Depth-First-Search by noting the order at which nodes are popped off the traversal stack.
3. Initialize an array the same size as the number of nodes to zero, representing the maximum number of nodes that can be reached before arriving at that node.
4. Go through the nodes in topological order and check the current maximum number of nodes that can be reached before arriving at each of its adjacent nodes. If more nodes can be reached by going from the current node to the adjacent node, update the array accordingly.
5. A single run is possible if there is a node such that the maximum number of nodes that can be reached before arriving at that node is equal to the number of nodes. Otherwise, it is impossible.

At any given point, each node has a maximum number of nodes before it, called its ‘distance’, stored in an array. When visiting a node, the algorithm checks each adjacent node to compare its current maximum distance with the distance if visiting the adjacent node via the current node, and updates the distances accordingly. This greedy algorithm can only work if the nodes are visited in topological order to ensure each node does not point to nodes that come before it, otherwise the greedy algorithm has to be updated retrospectively. If there is a node that can be reached by traversing through a maximum of n nodes, then there is a way to traverse the graph in a single pass.

Topological sorting works by traversing the graph using Depth-First-Search and noting the order at which nodes are popped off a traversal stack. This works because nodes are only popped off a traversal stack when there are no more unvisited nodes adjacent to it, meaning the reverse order ensures nodes are ordered such that each node only points to nodes that come after it.

Given n is the number of nodes and m is the number of edges, constructing the adjacency list is O(n+m) to initialize deques for each node and insert each edge. In DFS, each node is visited exactly once, and we check each of its edges once using an adjacency list, which means topological sorting is O(n+m). Given a fixed starting node, updating the maximum number of nodes that can be reached before each node requires visiting every node and each edge in its adjacency list, which takes O(n+m). Hence, the overall time complexity is O(n+m).