1106. Parsing A Boolean Expression

Hard Stack Recursion String Leetcode Link

Problem Description

The problem at hand entails evaluating a boolean expression that can be either `true` (`'t'`) or `false` (`'f'`). The expression can take one of several forms: a direct `true` or `false` value, a negation of a subexpression, a logical AND of one or more subexpressions, or a logical OR of one or more subexpressions. The different expressions are represented as:

- `'t'`: evaluates to `true`
- `'f'`: evaluates to `false`
- `'!(subExpr)'`: evaluates to the logical NOT of the inner expression `subExpr`
- `'&(subExpr1, subExpr2, ..., subExprn)'`: evaluates to the logical AND of all the inner expressions `subExpr1, subExpr2, ..., subExprn` (with `n` >= 1)
- `'|(subExpr1, subExpr2, ..., subExprn)'`: evaluates to the logical OR of all the inner expressions `subExpr1, subExpr2, ..., subExprn` (with `n` >= 1)

The goal is to compute the result of the given boolean expression, assured by the problem's constraints to be valid and conform to the rules above.

Intuition

The intuition behind the solution for evaluating the boolean expression is to process the expression in a structured way, making use of a stack to keep track of the characters that form a subexpression until it's complete and can be evaluated. This resembles the classic stack-based evaluation of expressions which is commonly used in parsing algorithms.

Here's how the approach breaks down:

1. **Initialize a Stack**: Since the expression is nested with parentheses, we can use a stack to keep track of and evaluate the inner expressions in the correct order.

2. **Iterate through Each Character**:

   - If we encounter a `t`, `f`, `!`, `&`, or `|`, we push it onto the stack as these characters indicate the beginning of an expression or a value.
   - If a character is a `)`, it means we have reached the end of an inner expression, so we begin to evaluate it.

3. **Evaluate the Expression**:

   - We keep popping from the stack until we get `t`, `f`, or an operator. Here, we count how many `t` and `f` we have encountered which corresponds to the truth values.
   - Based on the operator before the sequence of `t` and `f` values (`!`, `&`, `|`), we apply the NOT, AND, or OR operation accordingly. For AND, the result is `false` if any `f` is found; for OR, the result is `true` if any `t` is found; for NOT, the result is `true` if `f` is found and vice versa.
   - The result of the inner expression (resulting `t` or `f`) is then pushed back onto the stack as the new 'value' for that subexpression.

4. **Result**:

   - After evaluating the entire expression, the last element on the stack is the value of the whole expression. We check if it's equal to `'t'` to return `True` else it will be `False`.

This approach carefully constructs truth values by evaluating the smallest subexpressions first and combining them according to logical operations as specified by the nested structure of the parentheses and operators.

Solution Approach

To decipher and evaluate the boolean expression provided in the problem statement, the solution leverages the use of a stack (`stk`) data structure to keep track of the components of the expression.

Here is how the implementation works:

- Iterate through every character (`c`) in the string `expression`.
- When encountering characters that are `'t'`, `'f'`, `'!'`, `'&'`, or `'|'`, they are pushed onto the stack (`stk`), as these signify values or operators.
- If the character is a closing parenthesis `')'`, it indicates the end of a subexpression and triggers the evaluation sequence:
  - Initialize counters for true (`t`) and false (`f`) occurrences within the subexpression.
  - Pop characters from the stack until an operator is found, counting the `t` and `f` values found in the way.
  - Based on the popped operator, apply the logical NOT (`!`), AND (`&`), or OR (`|`) operation:
    - For `!`: Invert the boolean value, push `'t'` if there was an `f` in the subexpression, and `'f'` if there was a `t`.
    - For `&`: If any `f` was found, the result of the AND operation is `'f'`, otherwise, it's `'t'`.
    - For `|`: If any `t` was found, the result of the OR operation is `'t'`, otherwise, it's `'f'`.
  - The result of this operation replaces the subexpression by being pushed onto the stack.
- After the iteration is complete, the last character left in the stack is the result of the entire expression. Return `True` if this character is `'t'`, otherwise return `False`.

A key algorithmic pattern used here is stack-based parsing, which is a common approach for handling nested expressions. Using a stack allows us to respect the precedence of the logical operations, dealing with the innermost expressions first before combining the results into higher-level expressions—essentially a form of bottom-up evaluation.

This pattern is effective because we process the expression linearly, maintaining only the information necessary for the current subexpression's evaluation. The stack implicitly handles the "waiting" operations, resuming them when their operands become available after inner expressions are evaluated.

In Python, the solution makes use of the `match` statement (available from Python 3.10), which provides a clean and readable way to perform actions based on the value of the operator. This increases the clarity of the code, especially when dealing with multiple conditions.

Overall, the combination of a simple stack along with a case-based action sequence allows the implementation to concisely and effectively evaluate the boolean expressions as required by the problem's specifications.

Example Walkthrough

Let's consider a small example that illustrates the solution approach:

Given the expression: `&(t,|(f,t),!(f))`

Now, let's step through the process of evaluating it:

Step 1: Initialize Stack

Initially, our stack (`stk`) is empty.

Step 2: Start Iterating

- We encounter `'&'` and push it onto `stk`: `stack = ['&']`
- We encounter `'('` and ignore it as it's not part of evaluation.
- We encounter `'t'` and push it onto `stk`: `stack = ['&', 't']`
- We encounter `','` (a delimiter between expression values) and ignore it.
- We encounter `'('` and ignore it.
- We encounter `'|'` and push it onto `stk`: `stack = ['&', 't', '|']`
- We encounter `'f'` and push it onto `stk`: `stack = ['&', 't', '|', 'f']`
- We encounter `','` and ignore it.
- We encounter `'t'` and push it onto `stk`: `stack = ['&', 't', '|', 'f', 't']`
- We encounter `')'` and now we have to evaluate the subexpression with OR.

Step 3: Evaluate OR Subexpression

- We start popping from `stk` until we hit the `|` operator: we get `'t'` and `'f'` (we count `1` true and `1` false). Once we hit `'|'`, we check for the `|` operation rules.
- We found a `'t'` in our count, so the result of the OR subexpression is `'t'`. We push this result on `stk`: `stack = ['&', 't', 't']`
- We encounter `','` and ignore it.
- We encounter `'!'` and push it onto `stk`: `stack = ['&', 't', 't', '!']`
- We encounter `'('` and ignore it.
- We encounter `'f'` and push it onto `stk`: `stack = ['&', 't', 't', '!', 'f']`
- We encounter `')'` and now evaluate the NOT subexpression.

Step 4: Evaluate NOT Subexpression

- We start by popping from `stk` and we find `'f'`. Once we hit `'!'`, we apply NOT on our `f`, which makes it a `'t'`. We push this `'t'` on `stk`: `stack = ['&', 't', 't', 't']`

At this point, we've evaluated all inner expressions and are left with the AND operation to perform on the remaining stack elements.

Step 5: Final Evaluation

- We keep popping and find all `t`s, meaning that our AND operation will result in `true`. The top-level operator `'&'` ensures we check all values.
- Once we've evaluated this final expression, our stack is now: `stack = ['t']`.

Step 6: Result

- Our stack is left with a single element `'t'`, which means our entire expression evaluates to `True`.

This step-by-step process demonstrates how the stack-based evaluation method effectively breaks down and solves the expression by evaluating inner subexpressions and progressively building up to the entire expression's value.

Python Solution

```python
class Solution:
    def parseBoolExpr(self, expression: str) -> bool:
        stack = []

        # Iterate through each character in the expression
        for char in expression:
            # If it's an operator or a boolean value, push it onto the stack
            if char in 'tf!&|':
                stack.append(char)
            # When we encounter a closing parenthesis, we should evaluate the expression
            elif char == ')':
                true_count = false_count = 0

                # Count the number of true ('t') and false ('f') until we hit an operator
                while stack[-1] in 'tf':
                    true_count += stack[-1] == 't'
                    false_count += stack[-1] == 'f'
                    stack.pop() # Remove the boolean value as it's counted

                # Pop the operator from the stack and apply the logic for each operator
                operator = stack.pop()
                if operator == '!':
                    # For not, result is true if we have a false, otherwise it's false
                    result_char = 't' if false_count else 'f'
                elif operator == '&':
                    # For and, result is false if we have any false, otherwise it's true
                    result_char = 'f' if false_count else 't'
                elif operator == '|':
                    # For or, result is true if we have any true, otherwise it's false
                    result_char = 't' if true_count else 'f'

                # Push the result of the evaluation back onto the stack
                stack.append(result_char)

        # At the end, the stack should only have one element which is the result
        # Return True if the result is 't', otherwise False
        return stack[0] == 't'

```

Java Solution

```java
class Solution {
    public boolean parseBoolExpr(String expression) {
        // Stack to hold characters representing boolean values and operators
        Deque<Character> stack = new ArrayDeque<>();

        // Iterate through each character in the expression
        for (char c : expression.toCharArray()) {
            if (c != '(' && c != ')' && c != ',') {
                // If the character is not a parenthesis or comma, push it onto the stack
                stack.push(c);
            } else if (c == ')') {
                // When a closing parenthesis is encountered, evaluate the expression inside
                int countTrue = 0; // Count of 't' (true) characters
                int countFalse = 0; // Count of 'f' (false) characters

                // Pop characters from the stack until the corresponding operator is found
                while (stack.peek() == 't' || stack.peek() == 'f') {
                    if (stack.peek() == 't') {
                        countTrue++;
                    } else {
                        countFalse++;
                    }
                    stack.pop();
                }

                // Pop the operator ('!', '&', or '|')
                char operator = stack.pop();

                // Evaluate the expression based on the operator
                if (operator == '!' && countFalse > 0) {
                    // For '!', the expression is true if it has any 'f' (since '!' is a negation)
                    c = 't';
                } else if (operator == '&' && countFalse == 0) {
                    // For '&', the expression is true if all are 't'
                    c = 't';
                } else if (operator == '|' && countTrue > 0) {
                    // For '|', the expression is true if at least one is 't'
                    c = 't';
                } else {
                    // In all other cases, the expression is false
                    c = 'f';
                }

                // Push the result of the evaluated expression back onto the stack
                stack.push(c);
            }
        }
        // The result of the entire expression is at the top of the stack
        // Return true if the top of the stack is 't', otherwise false
        return stack.peek() == 't';
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to evaluate the boolean expression.
    bool parseBoolExpr(string expression) {
        stack<char> charStack; // Create a stack to manage the characters.

        // Iterating over each character in the input expression.
        for (char c : expression) {
            // If the character is not an operator or a comma, push it onto the stack.
            if (c != '(' && c != ')' && c != ',') {
                charStack.push(c);
            } else if (c == ')') { // When a closing bracket is encountered, evaluate the expression inside it.
                int trueCount = 0, falseCount = 0; // Variables to count the number of true and false values.

                // Process all operands up to the previous operator.
                while (charStack.top() == 't' || charStack.top() == 'f') {
                    // Increment trueCount or falseCount based on the operand.
                    trueCount += charStack.top() == 't';
                    falseCount += charStack.top() == 'f';
                    charStack.pop(); // Remove the processed operand from the stack.
                }

                // Now, pop the operator from the stack.
                char operatorChar = charStack.top();
                charStack.pop();

                // Evaluate the expression based on the operator.
                if (operatorChar == '!') {
                    c = falseCount > 0 ? 't' : 'f'; // Logical NOT: 'true' if any is 'false', else 'false'.
                }
                if (operatorChar == '&') {
                    c = falseCount > 0 ? 'f' : 't'; // Logical AND: 'false' if any is 'false', else 'true'.
                }
                if (operatorChar == '|') {
                    c = trueCount > 0 ? 't' : 'f'; // Logical OR: 'true' if any is 'true', else 'false'.
                }

                // Push the result onto the stack.
                charStack.push(c);
            }
        }

        // The top of the stack now contains the result, return 'true' if 't', 'false' otherwise.
        return charStack.top() == 't';
    }
};

```

Typescript Solution

```typescript
// Parses the given boolean expression and returns the result.
// Supported operators: '!', '&', and '|'
// Operands can be 't' (true) or 'f' (false)
// Expression examples: "|(f,t)", "&(t,f)", "!(t)", "&(!(&(t,f,t)),|(f,t))"
// Each expression is guaranteed to be valid and only consist of these characters.
function parseBoolExpr(expression: string): boolean {
    // The current index pointer used while scanning through the expression
    let currentIndex = 0;
    // The length of the expression
    const expressionLength = expression.length;

    // Recursive function to evaluate the expression
    const evaluate = (): boolean[] => {
        let resultStack: boolean[] = [];
        while (currentIndex < expressionLength) {
            const currentChar = expression[currentIndex++];

            if (currentChar === ')') {
                // End of the current expression
                break;
            }

            if (currentChar === '!') {
                // Not operation: Flip the boolean value of the next expression
                resultStack.push(!evaluate()[0]);
            } else if (currentChar === '|') {
                // Or operation: Return true if any of the values in the next expression is true
                resultStack.push(evaluate().some(value => value));
            } else if (currentChar === '&') {
                // And operation: Return true only if all values in the next expression are true
                resultStack.push(evaluate().every(value => value));
            } else if (currentChar === 't') {
                // Literal 'true' value
                resultStack.push(true);
            } else if (currentChar === 'f') {
                // Literal 'false' value
                resultStack.push(false);
            }
        }
        return resultStack;
    };

    // Start evaluating the expression and return the result of the top-level expression
    return evaluate()[0];
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `expression` string. This is because the code iterates over each character in the input string exactly once. During the iteration, each character is processed in a way that takes constant time. Operations like pushing to and popping from the stack, comparing characters, and assigning values all occur in `O(1)` time.

The space complexity of the code is also `O(n)`. This is due to the use of a stack that, in the worst case, might need to store an element for every character in the `expression` string before any reduction happens. The case that would result in this space usage is one where the string contains a long series of open parentheses before reaching any operator or closed parenthesis.