736. Parse Lisp Expression

Seen this question in a real interview before?

- Difficulty:Hard
- Total Accepted:532
- Total Submissions:1.4K
- Contributor: 1337c0d3r
-
- Subscribe to see which companies asked this question.

You are given a string `expression` representing a Lisp-like expression to return the integer value of.

The syntax for these expressions is given as follows.

- An expression is either an integer, a let-expression, an add-expression, a mult-expression, or an assigned variable. Expressions always evaluate to a single integer.
- (An integer could be positive or negative.)
- A let-expression takes the form `(let v1 e1 v2 e2 ... vn en expr)`, where `let` is always the string `"let"`, then there are 1 or more pairs of alternating variables and expressions, meaning that the first variable `v1` is assigned the value of the expression `e1`, the second variable `v2` is assigned the value of the expression `e2`, and so on **sequentially**; and then the value of this let-expression is the value of the expression `expr`.
- An add-expression takes the form `(add e1 e2)` where `add` is always the string `"add"`, there are always two expressions `e1, e2`, and this expression evaluates to the addition of the evaluation of `e1` and the evaluation of `e2`.
- A mult-expression takes the form `(mult e1 e2)` where `mult` is always the string `"mult"`, there are always two expressions `e1, e2`, and this expression evaluates to the multiplication of the evaluation of `e1` and the evaluation of `e2`.
- For the purposes of this question, we will use a smaller subset of variable names. A variable starts with a lowercase letter, then zero or more lowercase letters or digits. Additionally for your convenience, the names "add", "let", or "mult" are protected and will never be used as variable names.
- Finally, there is the concept of scope. When an expression of a variable name is evaluated, **within the context of that evaluation**, the innermost scope (in terms of parentheses) is checked first for the value of that variable, and then outer scopes are checked sequentially. It is guaranteed that every expression is legal. Please see the examples for more details on scope.

Evaluation Examples:

```
Input: (add 1 2)
Output: 3

Input: (mult 3 (add 2 3))
Output: 15

Input: (let x 2 (mult x 5))
Output: 10

Input: (let x 2 (mult x (let x 3 y 4 (add x y))))
Output: 14
Explanation: In the expression (add x y), when checking for the value of the
variable x,
we check from the innermost scope to the outermost in the context of the
variable we are trying to evaluate.
Since x = 3 is found first, the value of x is 3.

Input: (let x 3 x 2 x)
Output: 2
Explanation: Assignment in let statements is processed sequentially.

Input: (let x 1 y 2 x (add x y) (add x y))
Output: 5
Explanation: The first (add x y) evaluates as 3, and is assigned to x.
The second (add x y) evaluates as 3+2 = 5.

Input: (let x 2 (add (let x 3 (let x 4 x)) x))
Output: 6
Explanation: Even though (let x 4 x) has a deeper scope, it is outside the
context
of the final x in the add-expression.  That final x will equal 2.

Input: (let a1 3 b2 (add a1 1) b2)
Output 4
Explanation: Variable names can contain digits after the first character.
```

Note:

- The given string `expression` is well formatted: There are no leading or trailing spaces, there is only a single space separating different components of the string, and no space between adjacent parentheses. The expression is guaranteed to be legal and evaluate to an integer.
- The length of `expression` is at most 2000. (It is also non-empty, as that would not be a legal expression.)
- The answer and all intermediate calculations of that answer are guaranteed to fit in a 32-bit integer.

Solutions

If the expression is a variable, we look up in the map and return the variable value. If the expression is a value, we simply return its value.

For the "let" case, we first get the variable name, and the following expression. Then we evaluate the expression, and use a map to assign the expression value to the variable. For example, consider "(let x (add 2 3) x)", the variable is "x", and we evaluate the expression "(add 2 3)", and assign x = 5. For the last "x", we recursively call the help function, and get its value 5.

For the "add" case, we evaluate the value of the first expression, and the second expression, and add them together. For example, consider "(add (add 2 3) (add 3 4))", the first expression is "(add 2 3)", and the second expression is "(add 3 4)". We get 5 after evaluating "(add 2 3)", and get 7 after evaluating "(add 3 4)", and we will return 12.

The "mult" case is similar to the "add" case.

```
class Solution {
public:
    int evaluate(string expression) {
        unordered_map<string,int> myMap;
        return help(expression,myMap);
    }

    int help(string expression,unordered_map<string,int> myMap) {
        if ((expression[0] == '-') || (expression[0] >= '0' && expression[0] <=
'9'))
            return stoi(expression);
        else if (expression[0] != '(')
            return myMap[expression];
        //to get rid of the first '(' and the last ')'
        string s = expression.substr(1,expression.size()-2);
        int start = 0;
        string word = parse(s,start);
        if (word == "let") {
            while (true) {
                string variable = parse(s,start);
                //if there is no more expression, simply evaluate the variable
                if (start > s.size())
                    return help(variable,myMap);
                string temp = parse(s,start);
                myMap[variable] = help(temp,myMap);
            }
        }
        else if (word == "add")
            return help(parse(s,start),myMap) + help(parse(s,start),myMap);
        else if (word == "mult")
            return help(parse(s,start),myMap) * help(parse(s,start),myMap);
    }

    //function to seperate each expression
    string parse(string &s,int &start) {
        int end = start+1, temp = start, count = 1;
        if (s[start] == '(') {
            while (count != 0) {
                if (s[end] == '(')
                    count++;
                else if (s[end] == ')')
                    count--;
```

```
                end++;
            }
        }
        else {
            while (end < s.size() && s[end] != ' ')
                end++;
        }
        start = end+1;
        return s.substr(temp,end-temp);
    }
};
```