Parse Lisp Expression

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.

Solution 1

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);
    }
};
```

written by qhhuyanzhuo original link here

Solution 2

Using a stack, when encountering '(', save the current tokens and variable states in the stack.

```
def evaluate(self, expression):
    st, d, tokens = [], {}, ['']

    def getval(x):
        return d.get(x, x)

    def evaluate(tokens):
        if tokens[0] in ('add', 'mult'):
            tmp = map(int, map(getval, tokens[1:]))
            return str(tmp[0] + tmp[1] if tokens[0] == 'add' else tmp[0] * tmp[1])
        else: #let
            for i in xrange(1, len(tokens)-1, 2):
                if tokens[i+1]:
                    d[tokens[i]] = getval(tokens[i+1])
            return getval(tokens[-1])

    for c in expression:
        if c == '(':
            if tokens[0] == 'let':
                evaluate(tokens)
            st.append((tokens, dict(d)))
            tokens =  ['']
        elif c == ' ':
            tokens.append('')
        elif c == ')':
            val = evaluate(tokens)
            tokens, d = st.pop()
            tokens[-1] += val
        else:
            tokens[-1] += c
    return int(tokens[0])
```

written by totolipton original link here

Solution 3

The code I wrote is so ugly, yet the idea besides it is very simple:

In a whole view, the code implements a regression idea: From an outer expression to an inner expression. We call eval(String exp, Map<Integer, Integer> upper) function regressively every time when we face a Lisp expression inside an outer Lisp function so that we could ignore the detail to deal with it.

Let's explain the most two ugly implementations I took, though I have no idea to beautify it.

Explanation 1: The function readNext(String exp, int low){}

This function is used to do a simple work: to get the next expression name/Variable/Value/expresion. I implemented it so long since the expression name ends with a space, yet the expression ends when the '(' and ')' match. I want to return the current position as well as the nextStr, however only one variable is allowed to return, So I have to make nextStr global.

Explantion 2: The use of upperMap in eval(String exp, Map<String, Integer> upperMap){
...
Map<String, Integer> map = new HashMap<>();
map.putAll(upperMap);
...
}

Let's look at the testing case:

```
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.
```

People tend to return the output 8 since the inner expression "(let x 4 x))" has changed variable x to 4. The right logic is that the HashMap outside the inner expression will have effects in the inner one, yet the inner expression cannot affect the HashMap of outside expression

Here is the accepted code:

```
class Solution {
    String nextStr = "";
    public int evaluate(String exp){
        return eval(exp.substring(1, exp.length() - 1), new HashMap<>());
    }

    public int eval(String exp, Map<String, Integer> upperMap) {
        int pos = 0;
        int len = exp.length() - 1;
        Map<String, Integer> map = new HashMap<>();
        map.putAll(upperMap);

        pos = readNext(exp, pos);
```

```
        pos = readNext(exp, pos);

        if(nextStr.equals("add")){

            pos = readNext(exp, pos);
            // we have to consider if nextStr a number, a variable or a expression
            int left = nextStr.indexOf("(") == -1 ? map.containsKey(nextStr) ? map.
get(nextStr) : Integer.parseInt(nextStr) : eval(nextStr.substring(1, nextStr.length(
) - 1), map);

            pos = readNext(exp, pos);
            int right = nextStr.indexOf("(") == -1 ? map.containsKey(nextStr) ? map
.get(nextStr) : Integer.parseInt(nextStr) : eval(nextStr.substring(1, nextStr.length
() - 1), map);

            return left + right;
        }

        else if(nextStr.equals("mult")){

            pos = readNext(exp, pos);
            int left = nextStr.indexOf("(") == -1 ? map.containsKey(nextStr) ? map.
get(nextStr) : Integer.parseInt(nextStr) : eval(nextStr.substring(1, nextStr.length(
) - 1), map);

            pos = readNext(exp, pos);
            int right = nextStr.indexOf("(") == -1 ? map.containsKey(nextStr) ? map
.get(nextStr) : Integer.parseInt(nextStr) : eval(nextStr.substring(1, nextStr.length
() - 1), map);

            return left * right;
        }
        else{

            pos = readNext(exp, pos);
            // read the next Str one time, if there pos doesn't reach len, it means
we are still in the process of assignment. Since if pos < len, we still have the nex
t str (of nextStr) to match with nextStr
            while(pos < len){
                String var = nextStr;

                pos = readNext(exp, pos);
                int val = nextStr.indexOf("(") == -1 ? map.containsKey(nextStr) ? m
ap.get(nextStr) : Integer.parseInt(nextStr) : eval(nextStr.substring(1, nextStr.len
gth() - 1), map);

                map.put(var, val);
                pos = readNext(exp, pos);
            }

            return nextStr.indexOf("(") == -1 ? map.containsKey(nextStr) ? map.get(
nextStr) : Integer.parseInt(nextStr) : eval(nextStr.substring(1, nextStr.length() -
1), map);
        }
    }

    // read the next expression/variable/value
    public int readNext(String exp, int pos){
```

```
    public int readNext(String exp, int pos){
        int len = exp.length();
        int tmp = pos;
        int cnt = 0;
        // deal with expression
        if(exp.charAt(pos) == '('){
            pos++; cnt++;
            for(; pos < len; pos++){
                char ch = exp.charAt(pos);
                if(cnt == 0){
                    break;
                }
                if(ch == '('){
                    cnt++;
                }
                else if(ch == ')'){
                    cnt--;
                }
            }
        }
        // deal with variable or value
        else{
            for(; pos < len; pos++){
                char ch = exp.charAt(pos);
                if(ch == ' ' || ch == ')'){
                    break;
                }
            }
        }

        nextStr = exp.substring(tmp, pos);
        return pos + 1;
    }
}
```

written by victorzhang21503 original link here