Expression parsing

Table of Contents

A string containing a mathematical expression containing numbers and various operators is given. We have to compute the value of it in $O(n)$, where n is the length of the string.

The algorithm discussed here translates an expression into the so-called **reverse Polish notation** (explicitly or implicitly), and evaluates this expression.

Reverse Polish notation

The reverse Polish notation is a form of writing mathematical expressions, in which the operators are located after their operands. For example the following expression

$$a + b * c * d + (e - f) * (g * h + i)$$

can be written in reverse Polish notation in the following way:

$$abc*d*+ef-gh*i+*+$$

The reverse Polish notation was developed by the Australian philosopher and computer science specialist Charles Hamblin in the mid 1950s on the basis of the Polish notation, which was proposed in 1920 by the Polish mathematician Jan Łukasiewicz.

The convenience of the reverse Polish notation is, that expressions in this form are very **easy to evaluate** in linear time. We use a stack, which is initially empty. We will iterate over the operands and operators of the expression in reverse Polish notation. If the current element is a number, then we put the value on top of the stack, if the current element is an operator, then we get the top two elements from the stack, perform the operation, and put the result back on top of the stack. In the end there will be exactly one element left in the stack, which will be the value of the expression.

Obviously this simple evaluation runs in $O(n)$ time.

Parsing of simple expressions

For the time being we only consider a simplified problem: we assume that all operators are **binary** (i.e. they take two arguments), and all are **left-associative** (if the priorities are equal, they get executed from left to right). Parentheses are allowed.

We will set up two stacks: one for numbers, and one for operators and parentheses. Initially both stacks are empty. For the second stack we will maintain the condition that all operations are ordered by strict descending priority. If there are parenthesis on the stack, than each block of operators (corresponding to one pair of parenthesis) is ordered, and the entire stack is not necessarily ordered.

We will iterate over the characters of the expression from left to right. If the current character is a digit, then we put the value of this number on the stack. If the current character is an opening parenthesis, then we put it on the stack. If the current character is a closing parenthesis, the we execute all operators on the stack until we reach the opening bracket (in other words we perform all operations inside the parenthesis). Finally if the current character is an operator, then while the top of the stack has an operator with the same or higher priority, we will execute this operation, and put the new operation on the stack.

After we processed the entire string, some operators might still be in the stack, so we execute them.

Here is the implementation of this method for the four operators $+ - * /$:

```
bool delim(char c) {
    return c == ' ';
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' ||
```

```cpp
}

int priority (char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    int r = st.top(); st.pop();
    int l = st.top(); st.pop();
    switch (op) {
        case '+': st.push(l + r); break;
        case '-': st.push(l - r); break;
        case '*': st.push(l * r); break;
        case '/': st.push(l / r); break;
    }
}

int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            while (!op.empty() && priority(op.
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnu
                number = number * 10 + s[i++]
            --i;
            st.push(number);
        }
    }

    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}
```

Thus we learned how to calculate the value of an expression in $O(n)$, at the same time we implicitly used the reverse Polish notation. By slightly modifying the above implementation it is also possible to obtain the expression in reverse Polish notation in an explicit form.

Unary operators

Now suppose that the expression also contains **unary** operators (operators that take one argument). The unary plus and unary minus are common examples of such operators.

One of the differences in this case, is that we need to determine whether the current operator is a unary or a binary one.

You can notice, that before an unary operator, there always is another operator or an opening parenthesis, or nothing at all (if it is at the very beginning of the expression). On the contrary before a binary operator there will always be an operand (number) or a closing parenthesis. Thus it is easy to flag whether the next operator can be unary or not.

Additionally we need to execute a unary and a binary operator differently. And we need to chose the priority of a binary operator higher than all of the binary operations.

In addition it should be noted, that some unary operators (e.g. unary plus and unary minus) are actually **right-associative**.

Right-associativity

Right-associative means, that whenever the priorities are equal, the operators must be evaluated from right to left.

As noted above, unary operators are usually right-associative. Another example for an right-associative operator is the exponentiation operator ($a \wedge b \wedge c$ is usually perceived as a^{b^c} and not as $(a^b)^c$).

What difference do we need to make in order to correctly handle right-associative operators? It turns out that the changes are very minimal. The only difference will be, if the priorities are equal we will postpone the execution of the right-associative operation.

The only line that needs to be replaced is

```
while (!op.empty() && priority(op.top()) >= pr
```

with

```
while (!op.empty() && (
        (left_assoc(cur_op) && priority(op.top
        (!left_assoc(cur_op) && priority(op.to
    ))
```

where `left_assoc` is a function that decides if an operator is left_associative or not.

Here is an implementation for the binary operators $+ - * /$ and the unary operators $+$ and $-$.

```
bool delim(char c) {
    return c == ' ';
```

```
}

bool is_op(char c) {
    return c == '+' || c == '-' || c == '*' ||
}

bool is_unary(char c) {
    return c == '+' || c=='-';
}

int priority (char op) {
    if (op < 0) // unary operator
        return 3;
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return -1;
}

void process_op(stack<int>& st, char op) {
    if (op < 0) {
        int l = st.top(); st.pop();
        switch (-op) {
            case '+': st.push(l); break;
            case '-': st.push(-l); break;
        }
    } else {
        int r = st.top(); st.pop();
        int l = st.top(); st.pop();
        switch (op) {
            case '+': st.push(l + r); break;
            case '-': st.push(l - r); break;
            case '*': st.push(l * r); break;
            case '/': st.push(l / r); break;
        }
    }
}

int evaluate(string& s) {
    stack<int> st;
    stack<char> op;
    bool may_be_unary = true;
    for (int i = 0; i < (int)s.size(); i++) {
        if (delim(s[i]))
            continue;

        if (s[i] == '(') {
            op.push('(');
            may_be_unary = true;
        } else if (s[i] == ')') {
            while (op.top() != '(') {
                process_op(st, op.top());
                op.pop();
            }
            op.pop();
            may_be_unary = false;
        } else if (is_op(s[i])) {
            char cur_op = s[i];
            if (may_be_unary && is_unary(cur_o
                cur_op = -cur_op;
            while (!op.empty() && (
                    (cur_op >= 0 && priority(o
                    (cur_op < 0 && priority(op
                )) {
                process_op(st, op.top());
                op.pop();
            }
            op.push(cur_op);
            may_be_unary = true;
        } else {
            int number = 0;
            while (i < (int)s.size() && isalnu
                number = number * 10 + s[i++]
            --i;
            st.push(number);
            may_be_unary = false;
        }
    }

    while (!op.empty()) {
        process_op(st, op.top());
        op.pop();
    }
    return st.top();
}
```