Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Discrete Structures for Computing CO1007

Assignment Report

Academic year: 2021 - 2022. Semester 212

2/5/2022

Assignment: Evaluating expression.

Discrete Structures for Computing (CO1007)

Semester: 212 - *Class:* CC02 - *Group:* 7

Group members:
1. Nguyen Huynh An - 2154018
2. Luong Le Long Vu - 2153980
3. Tran Gia Huy - 2152600
4. Le Tran Nguyen Khoa - 2152674

Contents

1 Introduction

Mathematical expressions can be calculated effortlessly by human, but not for computers. In the beginning times of computing, not many efficient algorithms for *infix* expressions (expressions that we usually use) were developed, thus, limiting the electrical computer's capability.

In 1954, Arthur Burks, Don Warren, and Jesse Wright proposed a new way to present an expression using postfix notation (reverse Polish notation), which is then reinvented Friedrich L. Bauer and Edsger W. Dijkstra in 1960s with an aim to enhance performance in evaluating expressions.[1] Up till now, this is still an efficient way to present an expression, which is still used in some modern devices.

In this assignment, we are going to use postfix and prefix notations to compute an expression using stack abstract data type, as well as algorithms to convert infix expressions into postfix and prefix accordingly.

2 Theory

2.1 Infix, prefix and postfix

2.1.1 Infix notation

Infix notation is the notation commonly used in arithmetical and logical formulas and statements. It is presented in the form:

```
<operand> <operator> <operand>
```

Below are the examples of infix notation:

```
3 + 2
26 / 7 + 9
A * B + C / D
```

2.1.2 Prefix notation

Prefix notation (also known as Polish notation) is the notation in which, the operands follow the operators.

```
<operator> <operand> <operand>
```

Below are the examples of prefix notation:

```
+ 3 2
+ / 26 7 9
+ * A B / C D
```

2.1.3 Postfix notation

Postfix notation (also known as reverse Polish notation) is in contrast with prefix notation: the operator follows the operands.

```
<operand> <operand> <operator>
```

Below are the examples of postfix notation:

```
3 2 +
26 7 / 9 +
A B * C D / +
```

2.2 Stack

Stack is an abstract data type that serves as a collection of elements. The stream of elements in and out a stack must follow: Last In - First Out (LIFO).

Stack is very useful in computer science, it is used everywhere from organizing memory in computer's hardware, to algorithms such as backtracking and most importantly, evaluating prefix and postfix expressions.

A simple stack should consist of 3 basic operations:

1. **push** : Placing an element on the top of the stack.
2. **pop** : Taking out an element from the top of the stack.
3. **top** : Getting an element currently on the top of the stack.

The below figure visualizes the process of these operations:

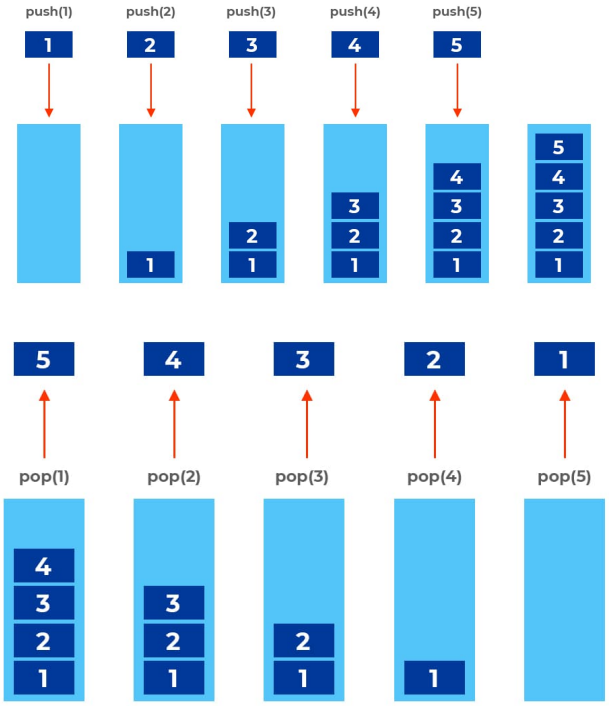

Figure: Visualization of **push** and **pop** operations

2.3 Binary expression tree

A binary expression tree is a specific kind of a binary tree used to represent expressions[3].

Given any expression, it could be presented as a binary tree. Depending on how we traverse it, the expression would be postfix (postorder traversal), prefix (preorder traversal) and infix (inorder traversal).

In programming, we usually implement a binary tree as a doubly linked list and use recursion to add new nodes to the tree according to some rules relating to the structure of the tree.

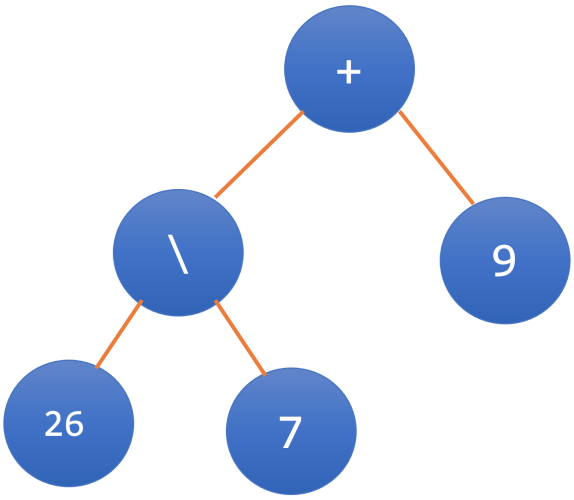

Figure: An example of an expression tree.

As we can see, the of *leafs* of the tree are always operands, while the *internal nodes* of the tree are operators. In postorder and preorder traversal, the pair *leafs* would always appear to the left or right of their according *internal node*.

```
Pre-order: <internal node> <leaf> <leaf>

Post-order: <leaf> <leaf> <internal node>
```

Because of that, when converting the expression (originally infix), instead of transform it to a binary tree, and traverse it again, we will make use of **stack** to convert the expression directly from infix to postfix or prefix, without going through the step of constructing the tree. This would save computing time, and we do not have to parse the input expression multiple times to decide which is the less prior operator to be the root of the tree.

2.4 Infix to postfix conversion algorithm

Comparing to prefix, postfix is relatively more convenient to be converted from infix, so we would to this first. Generally, the algorithm[1] can be described as follows:

[1] The algorithm assumes that the input string is already in the correct format.

0. Prepare two data structures: **string** to store the result and **stack** to store the operators.

1. Read the input string from left to right, character by character.

   (a) If it is an operand[2], append it to **string**.

   (b) If it is an operator, check if the top element in **stack** *has more priority* than the current operator. While yes, pop it and append to **string**. If no, push the current operator to **stack**.

   (c) If it is a closing bracket **')'**, pop and append to **string** all the elements in **stack** until its top is an opening bracket. Pop the opening bracket (not append it to **string**) and continue the loop with the next character.

2. When the loop finished, pop and append all the elements remaining in **stack**.

3. Return **string** as a result.

Explantion:

Operator prirority. As we could see, the biggest problem we face during the conversion is operators' priority, the expressions inside brackets must be computed first, then ' ^ '[3] , then '*' and '/', then '+' and '-'. This problem, however, can be solved conveniently using **stack**.

Since we have popped out any prior operators before pushing new ones, the operators lie exactly as we intended : higher priority on top, lower at the bottom.

When we take out the operators, due to LIFO property, the operators would be organized in correct order.

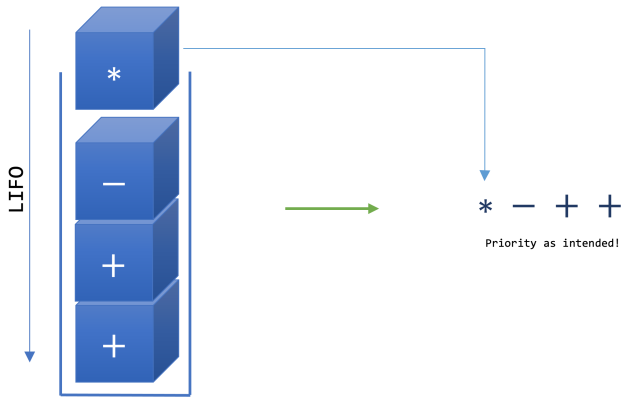

Figure: LIFO helps to organize the order of operators

Long operand. There is another problem arose when parsing the input string. Within a string, our computer can not distinguish distinct operands. Because of that, we have to tell the computer where is the end of an operand, and when to start the next.

```
1       <- 1 character
123     <- 3 characters
64723   <- 5 characters
```

[2] The opening operator '(' serves as an operator, with no priority in this case.

[3] In the case of nested power, there is still no universal standard regarding to its priority.

In this case, what we will do is to accept all the characters parsed **before** an operator to be **an operand**. This mechanism would be helpful when dealing with operands larger than one character. This would be shown clearer in our actual code.

Brackets of infix notation. To deal with brackets, we just need to consider the opening and closing brackets as **marks** for a **'sub-stack'** within a stack. We could consider that **sub-stack** to be a special operator which has the **highest** priority, lying in the stack.

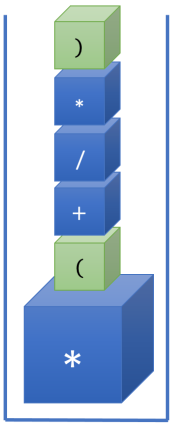

Figure: A sub-stack within a stack, marked by '(' and ')'

With this algorithm, we only need to parse the input string once.

2.5 Infix to prefix conversion algorithm

Once we have achieved the infix to postfix, converting to prefix is more straightforward. Theoretically, prefix is an inverse traversal comparing to postfix. Thus, to convert infix to prefix, we can follow the below algorithm:

1. Read the input string from *right* to left, character by character.
2. Use the same procedure as in postfix conversion. However, the brackets must also be inverse. In more detail, if the loop reach *an opening bracket* **'('**, pop and append to **string** all the elements in **stack** until its top is *a closing bracket*. Pop *the closing bracket* (not append it to **string**) and continue the loop with the next character.
3. *Reverse* the **string**.
4. Return the **string**.

2.6 Evaluating prefix and postfix algorithm

The expression, once converted into prefix or postfix form, can be conveniently computed by computers using **stack**.

As we could see, all the operators we concerned are *binary*. Thus, it needs exactly two operands to perform an evaluation. The order of operands, thanks to the property of **stack**, are organized orderly. And postfix and prefix traversal assures that there are always two operands when the loop reach an operator.

Our algorithm will take a postfix (or prefix) as an input string, and yield a numerical (or logical) result.

2.6.1 Postfix

0. Prepare **stack** to store the operands.
1. Read the input string from left to right.
   (a) If it is an operand, push it into **stack**.
   (b) If it is an operator, we take out two top elements in **stack**, perform the operation (logic or arithmetic), and push the result back into **stack**.
2. Continues until there is no operator left, and **stack** only contains one element.
3. Return the top element of **stack**.

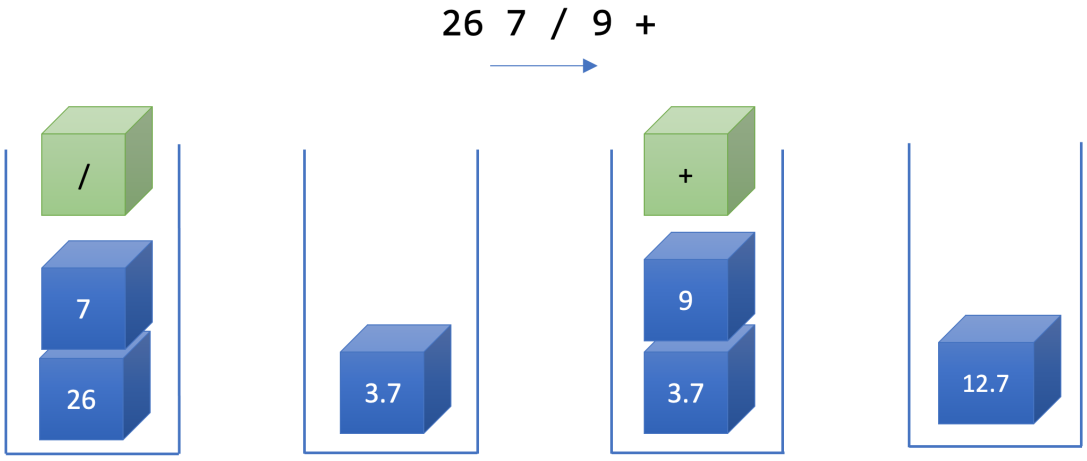

Figure: Visualization of postfix evaluation

2.6.2 Prefix

1. Reverse the string.
2. Perform the same procedure as prefix, with the input is the reversed string.
3. Return the result.

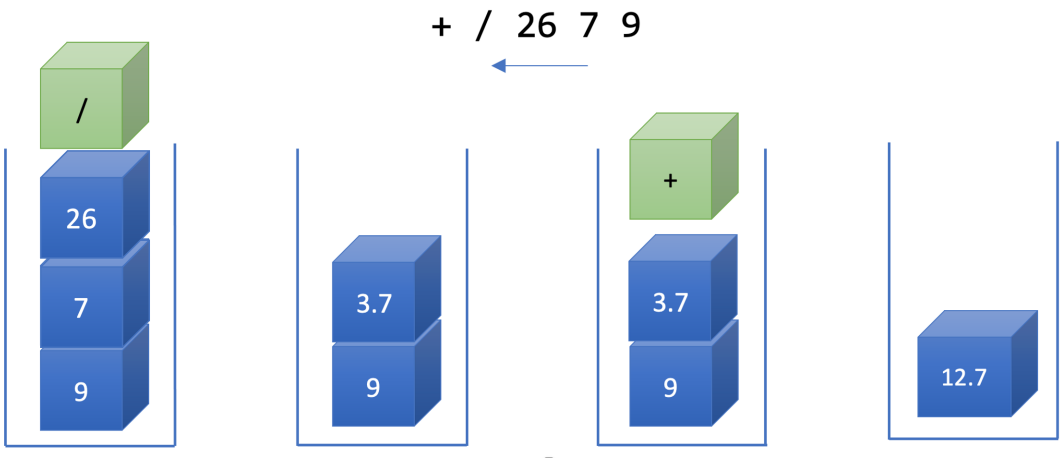

Figure: Visualization of prefix evaluation

2.7 Input string's validity check algorithm and Code

2.7.1 Exercise 1

There are 5 types of illegal input cases of arithmetic expression:

1. Consecutive operators

   In this case, if input string has from 2 or more consecutive operators, it can be seen as illegal case because it can raise undefined error.
   For example: $2//3, 4**5, 1+*3, ...$
   However, there are some legal cases for operators + and -. If the consecutive operators only have + or - or both of them, the input is still legal

   - For the consecutive operators play a role as the sign of operand, if the operators have the positive value, we can eliminate them, else we only put one operator - before the operand.
     For example $++2 \rightarrow 2, +-+5 \rightarrow -5, ...$
   - For the consecutive operators play a role as the operator between two operands, if the operators have the positive value, we put only one operators + to replace the consecutive operators, else we replace by one operator -.
     For example $1++1 \rightarrow 1+1, 2+-5 \rightarrow 2-5, ...$

This is our code to check string input in this case. If it is legal, the function will return the simplest form. Otherwise, it will return string "Error".

```
string checkConsecutiveOperators(const string& s) {
    string res = "";
    if (s.compare("Error") == 0)  return "Error";
    for (unsigned int i = 0; i < s.size(); i++) {
        if (isOperator(s[i])) {
            int num = 0, sign = 0, numH = 0, pos;
            int index = i;
            while (isOperator(s[index])) {
                index++;
                num++;
            }
            for (int j = i; j < i + num; j++) {
                if (s[j] == '+')  sign += 2;
                if (s[j] == '-')  sign += 1;
                if (s[j] == '*' || s[j] == '^' || s[j] == '/') {
                    numH++;
                    pos = j;
                }
            }
            if (i == 0) {
                if (numH > 0) return "Error";
                else {
                    if (sign % 2 == 0) res += "";
                    else res += '-';
                }
            }
            else {
                if (numH >= 1 && num>=2) return "Error";  //
                else if (numH == 1 && num == 1) {
                    res += s.at(i);
                }
                else  {
                    if (sign % 2 == 0) res += '+';
                    else res += '-';
                }
            }
            i += num - 1;
        }
        else res += s.at(i);
    }
    if (isOperator(res[res.size() - 1])) return "Error";
    return res;
}
```

2. Precedence order

In this case, the input will be illegal if there are operators which have the same order of precedence between the operands without any parenthesis to give the priority between them, causing the difference between the value of the prefix and postfix notation which are converted from the same infix-notation.

For example: Suppose that the following inputs are valid, we will use prefix and postfix algorithm to calculate the value of each:

- 1-2+3
  Postfix value: 2
  Prefix value: -4
  $\Longrightarrow$ Error
- 2/3*2
  Postfix value: $\frac{4}{3}$
  Prefix value: $\frac{1}{3}$
  $\Longrightarrow$ Error
- 1-2+1
  Postfix value: 0
  Prefix value: 0
  $\Longrightarrow$ Valid

So to check to check precedence order, we suppose the input string is valid and use the code to evaluate prefix and postfix value.
Firstly, we decalare the function prototypes as follows:

```
    string to_post_fix(const string& s);
    string to_pre_fix(const string& s);
    double arithmetic_value(const string& expression, unsigned
   int _mode);
```

Then, we caompare 2 results, if they are not equal, return string "Error". Otherwise, the function will return the initial string.

```
    string checkOrder(const string& s) {
        if (s.compare("Error") == 0)  return "Error";
        double u, v;
        u = arithmetic_value(to_post_fix(s), 1);
        v = arithmetic_value(to_pre_fix(s), 0);
        if (u == v) return s;
        else return "Error";
    }
```

3. Parenthesis

To check the parenthesis, we use stack to check whether the open brackets match with closed bracket and also store the position of open bracket in the initial string.

In this function, we use some functions related to brackets which are initialized before like isOpenBracket, isClosedBracket, isEnclosed.

For example: $4+)(3*2, (4(+3)-2$ is invalid.

After checking the brackets are matched, we will eliminate the brackets that do not have any operand inside. We check the breackets from the left to right by using the stack that stores the position of open brackets.

For example: $(4()+3)-2 \rightarrow (4+3)-2$

```
string checkParenthesis(const string& s) {
    // Check whether open brackets match with closed brackets or
    not
    if (s.compare("Error") == 0)  return "Error";
    string res;
    stack<char> bracket;
    stack<int> pos; //position of open bracket
    bool check = true;
    for (unsigned int i = 0; i < s.size(); i++) {
        if (isOpenBracket(s[i])) {
            bracket.push(s.at(i));
            pos.push(i);
        }
        else if (isClosedBracket(s[i])) {
            if (bracket.empty()) {
                check = 0;
                break;
            }
            else if (bracket.size() > 0) {
                if (!isEnclosed(s.at(i), bracket.top())) {
                    check = 0;
                    break;
                }
                else {
                    bracket.pop();
                }
            }
        }
        else continue;
    }
    if (check == 0) return "Error";
    else {   //eliminate brackets that do not have operand
   inside
        for (int i = 0; i < s.size(); i++) {
            res += s.at(i);
        }
        while (!pos.empty()) {
            if (isClosedBracket(res[pos.top() + 1])) {
                res[pos.top() + 1] = 0;
                res[pos.top()] = 0;
            }
            pos.pop();
        }
        return res;
    }
```

```
}
```

4. Floating Point

The decimal points are valid only when they are placed between the whole number part and fractional part.

For example: $3..4+2*2,5+2.+2$ are invalid.

About the algorithm of this function, it will check the first and last position of the string, if at least one of them is decimal point, the function will return string "Error". Then, with the rest positions, if one position is decimal point, we will check whether 2 positions beside it are numbers or not. After checking, if there are not any invalid decimal points, function will return the initial string.

```
string checkFloatingPoint(const string& s) {
    string res;
    if (s.compare("Error") == 0)  return "Error";
    int check = 1;
    int l = s.size();
    if (s[0] == '.' || s[l - 1] == '.') return "Error";
    for (int i = 1; i < l - 1; i++) {
        if (s[i] == '.') {
            if (isOperand(s[i - 1]) == 0 || isOperand(s[i + 1])
  == 0) {
                return "Error";
            }
            else res += s.at(i);
        }
        else res += s.at(i);
    }
    return s;
}
```

5. Blank

Input string will be invalid if there is any blank that is between 2 operands.

For example: 2 $4+3,2$ 3,...

So to check a string, the function will check whether the blank is between 2 operands or not. If there is no error, we will eliminate all the blank to make the string simpler.

```
string checkBlank(const string& s) {
    string res;
    if (s.compare("Error") == 0)  return "Error";
```

```
    for (int i = 0; i < s.size()-1; i++) {
        if (isOperand(s.at(i)) && s.at(i + 1) == ' ') {
            int j = 1;
            while (s.at(i + j) == ' ') {
                j++;
            }
            if (isOperand(s.at(i + j))) {
                return "Error";
            }
        }
    }
    for (int i = 0; i < s.size(); i++) {
        if (s.at(i) == ' ') continue;
        else res += s.at(i);
    }
    return res;
}
```

6. Check validity

   In this function, we will input string to check s and the string after check checkedS. We will clone checkedS as a copy of s, and then we will check checkedS by let is pass the following functions: checkBlank, checkConsecutiveOperators, checkFloating-Point, checkParenthesis, checkOrder. In each functions, I have written some lines that let they return string "Error" when the input string is "Error". The order of functions that checkedS passing is quite important, and that order can minimize the error when checking.

```
void checkValidity(const string& s, string& checkedS) {
    for (int i = 0; i < s.size(); i++) {
        checkedS += s.at(i);
    }
    checkedS = checkBlank(checkedS);
    checkedS = checkConsecutiveOperators(checkedS);
    checkedS = checkFloatingPoint(checkedS);
    checkedS = checkParenthesis(checkedS);
    checkedS = checkOrder(checkedS);
}
```

2.7.2 Exercise 2

There are 4 types of illegal input cases of logical expressions:

1. Consecutive operators

   In this case, if input string has at least 2 consecutive operators, it can be seen as illegal case because it can raise undefined error.

For example: p ∨∨ q; p ∧∧ q; p ∧ → q;...

However, there are some legal cases for operator ! (or ¬):

- 2 ! (or ¬) can be consecutive
- if there are 2 consecutive operators, the second operators can be ! (or ¬)

For example: p ∨¬ q; ¬¬ p;...

This is our code to check if input string is in this case. If it is illegal, the function will return 0.Otherwise, if the string is legal, the result will be 1:

```
    bool checkOperators_2(const string& s) {
    bool check = 1;
    for (unsigned int i = 0; i < s.size() - 1; i++) {
        if (isOperator(s.at(i)) && isOperator(s.at(i + 1))) {
            if (s.at(i + 1) != '!') { check = 0; break; }
        }
    }
    return check;
}
```

2. Precedence order

In this case, the input will be illegal if there are operators, which have the same order of precedence, between the operands without any parenthesis to check priority between them.

For example: p ∨ q ∧ r; p ∧ q ∨ r;...

We can see that ∨ and ∧ have the same precedence, so it can cause multi-output error.

This is our code to check if input string have the correct order. If it is illegal, the function will return 0.Otherwise, if the string is legal, the result will be 1:

```
    bool checkOrder_2(const string& s) {
    bool check = 1;
    for (unsigned int i = 0; i < s.size() - 2; i++) {
        if (s.at(i) == '&' || s.at(i) == '|' || s.at(i) == '+')
   {
            if ((s.at(i + 2) == '&' || s.at(i + 2) == '|' || s.
   at(i + 2) == '+') && s.at(i + 2) != s.at(i)) {
                check = 0;
                break;
            }
        }
    }
    return check;
}
```

3. Parenthesis

There are 2 possible parenthesis errors when we check an input string:

- There is an operators after an open bracket (except ¬) or before a closed bracket. **For example:** (p ∨ q)(
- The number of open brackets and closed brackets are not equal.

First, we have to check the first case. If the input is illegal, the function will return 0.

```
        bool check = 1;
    int open = 0, close = 0;
    for (unsigned int i = 0; i < s.size() - 2; i++) {
        if (isOpenBracket(s.at(i))) {
            if (isClosedBracket(s.at(i + 2)) && isOperator(s.at(
   i + 1))) {
                check = 0;
                break;
            }
            else if (isOperator(s.at(i + 1)) && s.at(i + 1) != '
   !') {
                check = 0;
                break;
            }
        }
    }
    for (unsigned int i = 1; i < s.size(); i++) {
        if (isClosedBracket(s.at(i))) {
            if (isOperator(s.at(i - 1))) { check = 0; break; }
        }
    }
```

If the input is legal in the first case, we will check the second one.

```
        for (unsigned int i = 0; i < s.size(); i++) {
            if (isOpenBracket(s.at(i))) {
                open++;
            }
            if (isClosedBracket(s.at(i))) {
                close++;
            }
            if (close > open) { check = 0; break; }
        }
```

```
        if (close != open) { check = 0; }
        return check;
```

4. Blank

   Input string will get blank error if there is only blank between 2 operands.
   For example: p q; p ∨ q r; ...

```
        bool check = 1;
        for (unsigned int i = 0; i < s.size() - 1; i++) {
            if (isOperand(s.at(i)) && s.at(i + 1) == ' ') {
                int j = 1;
                while (s.at(i + j) == ' ') {
                    j++;
                }
                if (isOperand(s.at(i + j))) { check = 0; break;
}
            }
        }
```

   Moreover, we also have to check 3 other errors after deleting all blanks of input string.

```
        if (check == 1) {
        string s1;
        for (unsigned int i = 0; i < s.size(); i++) {
            if (s.at(i) != ' ') {
                s1 += s.at(i);
            }
        }
        if (!checkOperators_2(s1) || !checkOrder_2(s1) || !
checkParenthesis_2(s1)) { check = 0; }
 }
 return check;
```

5. Check validity

   In this function, we will input string to check expression and the string after check s. We will check validity of expression. If expression is legal, we will clone s as a copy of expression. If it is illegal, s will be "Error

```
void checkValidity_2(const string &expression, string &s) {
    if (checkOperators_2(expression) && checkOrder_2(expression)
    && checkParenthesis_2(expression) && checkBlank_2(expression
   )) { s = expression; }
    else { s = "Error"; }
}
```

3 Code

3.1 Exercise 1. Arithmetic expression

3.1.1 Part (a)

The code is retrieved from `1a.cpp`

First, we declare a function prototype as follows:

```
string to_pre_fix(const string & s);
```

Then, we define the function `to_pre_fix()`, we will start by preparing the variables and data structures needed:

```
string res = "";
stack<char> operators;

bool start_operand = 0;
```

Notice that, we declare `bool start_operand = 0;` as a flag to read *long operands* (as explained in Section 2.3). The mechanism of this flag is simple:

1. If the loop's current character is an operand:
   (a) If `start_operand == 0`, append `';'` (to separate the last element from this element), and set `start_operand = 1` (turn it **on**).
   (b) Once the flag in turned on, the next characters would not be separated by `';'` (accept it as a part of ONE operand).
2. However, if the loop reaches an operator,
   the flag will be turned **off**-set `start_operand=0`. The next character will no longer be a part of the last operand.

Next, we do a backward `for` loop, start at position `size - 1` (as the last non-null character in a string is at position `size -1`), till the beginning of the string at position `0`. However, we let the loop ends at position `-1`.

```
for (int i = s.size() - 1 ; i >= -1 ; i--)
{
    ...
}
```

As described in Section 2.3 (Step 2.), even when the parsing process finishes after it read the string, there is still (possibly) some operators left in **stack** which have not been appended to the result **string**. Therefore, we let the final iteration at `-1` to pop and append all the remaining operators in **stack**. Here is how we handle it inside the `for` loop:

```
    // Put this at the beginning of the for loop
    if (i == -1)
    {
        while (!operators.empty())
        {
            res += ';';
            res += operators.top();
            operators.pop();
        }
        break;
        // break; : exit the loop to forbid the loop reaching other
   parts, preventing Segmentation fault.
    }
```

There are also other `if - else` conditions to handle if the current character is a bracket, an operand, or an operator.

First, to handle the bracket, we will have to check whether it is a closing bracket or *an opening bracket* using `bool isOpenBracket()`.

1. If the current character is a closing bracket, just push it into the **stack** as usual (Section 2.3, Step 1.).
2. If the current character is an opening bracket, we need to take out all the elements from **sub-stack** inside `operators`. To do that, we will use `bool isOpened()`[4] to check **if the current character is a closing bracket ')' or not**. (Section 2.4, Step 2.)

Here is how we implement it:

```
    if ( isBracket(s.at(i)) )
        {
            start_operand = 0;
            if ( isOpenBracket(s.at(i)) )
            {
                while ( !operators.empty() && !isOpened(s.at(i),
   operators.top()) )
                {
                    res += ';';
                    res += operators.top();
                    operators.pop();
                }

                operators.pop();
            } else
            {
                operators.push(s.at(i));
```

[4]The name 'isOpened' means that from the perspective of the reversed string, the closing bracket is 'opening'. Note that, we read the string from **right to left.**

```
            }
        }
```

Second, the operands. Notice that, the operands are only separated when `start_operand == 0`.

```
    if ( isOperand(s.at(i)) )
        {
        // The operands are only separated when start_operand == 0
            if ( !start_operand )
            {
                start_operand = 1;
                if (i != s.size() - 1)
                    res += ';';
            }

            res += s.at(i);
        }
```

Third, the operators. The `while` loop is to continuously pop and append all the less prior operators on top of **stack**, until it reaches higher one.

```
    if ( isOperator(s.at(i)) )
        {
            start_operand = 0;
            if ( !operators.empty() && morePrior(operators.top(), s.
at(i)) )
            {
                while (!operators.empty() && morePrior(operators.top
(), s.at(i)))
                {
                    res += ';';
                    res += operators.top();
                    operators.pop();
                }

                operators.push(s.at(i));
            } else
            {
                operators.push(s.at(i));
                // If no less prior operators, just push it into
stack as normal
            }
        }
```

We also need to *reverse* the result string before returning it (using `reverse_string()` function).

```
    reverse_string(res);
```

Throughout our implementation, we use character `';'` to separate out the elements. This would make it easier for our computer to identify during the *evaluation* step.

3.1.2 Part (b)

The code is retrieved from `1b.cpp`

Converting to postfix is similar to converting to prefix in certain steps. The most notable difference is that, in postfix-conversion, there is *no reversing before and after the parsing* and also, the *brackets are not inversed*.

In fact, it is more straightforward for us to understand infix to postfix algorithm, because there is no reversed things, and the input string is parsed from left to right as normal.

In the below code, instead of `isOpened()` in infix to prefix conversion, we use `isEnclosed()` to detect the opening bracket **'('**.

```
if ( isBracket(s.at(i)) )
    {
        start_operand = 0;
        if ( isClosedBracket(s.at(i)) )
        {
            while ( !operators.empty() && !isEnclosed(s.at(i),
operators.top()) )
            {
                res += ';';
                res += operators.top();
                operators.pop();
            }

            operators.pop();
        } else
        {
            operators.push(s.at(i));
        }
    }
```

3.1.3 Part (c)

The code is retrieved from `1c.cpp`

In this exercise, we would define a function that could compute an expression presented in either prefix or postfix. Notice that, the second parameter `_mode` acts as a switch between prefix and postfix. As a convention, `_mode == 0`: evaluate prefix string and `_mode == 1`: evaluate postfix string.

```
double arithmetic_value(const string & expression, unsigned int
    _mode);
```

First, we prepare necessary variables for our procedure.

1. `double res;` to store the result
2. `string temp_operand = "";` to store temporarily digits in an operand before being pushed into **stack**. This is how we handle a *long operand*.
3. `stack<string> _stack;` to store values (still in the form of string), waiting to be used for calculation.

```
    double res = 0;
    string temp_operand = "";
    stack<string> _stack;
```

Then, we start to calculate the expression:

1. Prefix evaluation. We parse the string from *right to left*, whenever it is an operator, we perform a calculation accordingly.

Notice that, we take out the first top of **stack** as the *first operand*, and the next to be the *second operand* because, the first operator always goes in stack after the second operator since we parse the input string from right to left.(see Section 2.5.2's figure)

Here is how we implement this:

```
    for (int i = expression.size() - 1 ; i >= 0 ; --i)
        {
            if ( _stack.size() >= 2 && isOperator(expression.at(i))
    )
            {
                    // Getting two top operands
                    // a - first operand
                    // b - second operand
                double a = stod(_stack.top());
                _stack.pop();
                double b = stod(_stack.top());
                _stack.pop();

                // Evaluate & push it back to _stack
                // Contains if - else conditions
                [ ... ]

            } else

            if (expression.at(i) == ';')
            {
                if (temp_operand.size() > 0)
                    _stack.push(temp_operand);
                temp_operand = "";
                // Clear temp_operand string, reset it to "".
            } else

            {
                // NOTICE that because we are now reading
                // the input string REVERSELY we must insert
                // the current character to the beginning of
                // the operand, instead of append it to the end
                // of the operand.
                temp_operand.insert(0, 1, expression.at(i));
            }
        }
```

.

2. Postfix evaluation. We parse the string from *left to right*, whenever it is an operator, we perform a calculation accordingly.

Notice that, we take out the first top of **stack** as the *second operand*, and the next to be the *first operand* since this time, we parse from left to right. (see section 2.5.1's figure)

```
    for (unsigned int i = 0 ; i < expression.size() ; ++i)
        {
            if ( _stack.size() >= 2 && isOperator(expression.at(i))
    )
            {
                    // Getting two top operands
                    // a - first operator
                    // b - first operator
                double b = stod(_stack.top());
                _stack.pop();
                double a = stod(_stack.top());
                _stack.pop();

                // Evaluate & push it back to _stack
                // Contains if - else conditions
                [ ... ]

            } else

            // NOTICE: append the current character
            // to the end of the operand.
            temp_operand += expression.at(i);
        }
```

After the loop finished, we just need to take out the top (the only) element in **stack**, convert it to `double` and return it as a result.

```
    res = stod(_stack.top());

    return res;
```

3.2 Exercise 2. Logic expression

3.2.1 Part (a) & (b)

The conversion from infix to prefix and postfix of logical expression is similar to one of arithmetic expression. However, it is necessary to know that precedence of each operator in a logical expression.

Precedence	Operator	Alternative character (Use in code)
1	$\neg$	!
2	$\wedge$	&
3	$\vee$	|
4	$\rightarrow$	>
5	$\oplus$	+

To handle the priority of logical operators, we define the function `bool morePrior_2()`:

```
bool morePrior_2(char _stack_top, char c)
{
    if (_stack_top == '!')
    {
        if (c == '+' || c == '>' || c == '&' || c == '|')
            return true;
    }
    else
        if (_stack_top == '+' || _stack_top == '&' || _stack_top
== '|')
        {
            if (c == '+' || c == '>' || c == '&' || c == '|')
                return true;
        }
        else
            if (_stack_top == '>')
            {
                if (c == '>')
                    return true;
            }

    return false;
}
```

3.2.2 Part (c)

The code is retrieved from `2c.cpp`

The procedure for computing logical value is the same with arithmetic value. However, for symbolic variables such as p, q, a, b, c we need to give them a boolean value so that the evaluation could work.

Before the evaluation procedure, we let the user type in their boolean values, here is how we implement it:

```
// Getting value of variables
    string new_expression = "";
    int size = 0;
    char* variable = new char[size];
    int* value = new int[size];
    for (unsigned int i = 0; i < expression.size(); i++) {
        if (isVariable(expression.at(i))) {
            bool check = 1;
            for (int j = 0; j < size; j++) {
                if (expression.at(i) == variable[j]) { check =
0; break; }
            }
            if (check == 1) {
                variable[size] = expression.at(i);
                size++;
            }
        }
    }
```

```
        for (int i = 0; i < size; i++) {
            cout << variable[i] << ": ";
            cin >> value[i];
        }
        for (unsigned int i = 0; i < expression.size(); i++) {
            bool check = 0;
            for (int j = 0; j < size; j++) {
                if (expression.at(i) == variable[j]) {
                    new_expression += to_string(value[j]);
                    check = 1;
                    break;
                }
            }
            if (check == 0) { new_expression += expression.at(i); }
        }
```

Notice in the above code, during the input procedure, we *replace* the symbolic variable with its boolean value accordingly as user type his value in. This results in a new expression string, which only contains 0 and 1 and logical operators.

This process would allow the computation algorithm to perform without any hindrances.

4 Conclusion

After finishing this assignment, we feel that we have learnt a lot about stack data structure, basic string handling and how computer processes an expression to yield a desirable answer, thus, appreciate the works and the computational ability we have today.

From this project, we can create a working calculator, know how to implement a binary tree for a specific purpose and have an intuition of how a computer reads the programming language's syntax.

After all, we are thankful of this wonderful assignment and hope that we could develop this project further to make interesting applications.

Source code:

This GitHub repository contains our source code.
`https://github.com/SteveKhoa/discrete_assignment`

References:

[1] https://en.wikipedia.org/wiki/Reverse_Polish_notation
[2] https://raj457036.github.io/Simple-Tools/prefixAndPostfixConvertor.html
[3] https://en.wikipedia.org/wiki/Binary_expression_tree

Contacts

For any questions, please send an email to:

huy.trandev@hcmut.edu.vn
khoa.lesteve@hcmut.edu.vn