# Parsing Expression

There is a group of string problems where the goal is to parse the expression. To resolve such problems, you can think the concepts what you learned in the compile class. The solution is normally to build a state machine.

However, in some case the process order can be different than the input order, so you may need a stack to help you to parse the expression.

## 224. Basic Calculator

Hard

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, **non-negative** integers and empty spaces .

**Example 1:**

**Input:** "1 + 1"

**Output:** 2

**Example 2:**

**Input:** " 2-1 + 2 "

**Output:** 3

**Example 3:**

**Input:** "(1+(4+5+2)-3)+(6+8)"

**Output:** 23

**Note:**

* You may assume that the given expression is always valid.
* **Do not** use the eval built-in library function.

### **Analysis**

1. **When you build an integer, you can simply use num = num \* 10 + (ch – ‘0’)**
2. **You calculate the accumulated result and store it in stack on ‘(‘, pop it on ‘)’.**
3. **When we start ‘(‘ the new accumulated result is set 0, and previous one is in stack.**
4. **For any + or - in this problem, we can remember by a sign which is + -. Otherwise we have to store the operator**

/// <summary>

/// Leet code #224. Basic Calculator

/// Implement a basic calculator to evaluate a simple expression string.

/// The expression string may contain open ( and closing parentheses ),

/// the plus + or minus sign -, non-negative integers and empty spaces .

/// You may assume that the given expression is always valid.

/// Some examples:

/// "1 + 1" = 2

/// " 2-1 + 2 " = 3

/// "(1+(4+5+2)-3)+(6+8)" = 23

/// </summary>

int LeetCode::calculate(string s)

{

stack <int> nums, signs;

int result = 0;

int num = 0;

int sign = 1;

for (size\_t i = 0; i < s.size(); i++)

{

char c = s[i];

if (isdigit(c))

{

num = num \* 10 + (s[i] - '0');

}

else

{

result += sign \* num;

num = 0;

if (c == '+') sign = 1;

else if (c == '-') sign = -1;

else if (c == '(')

{

nums.push(result);

signs.push(sign);

result = 0;

sign = 1;

}

else if (c == ')')

{

num = result;

result = nums.top();

nums.pop();

sign = signs.top();

signs.pop();

}

}

}

result += sign \* num;

return result;

}

## 227. Basic Calculator II

Medium

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only **non-negative** integers, +, -, \*, /operators and empty spaces . The integer division should truncate toward zero.

**Example 1:**

**Input:** "3+2\*2"

**Output:** 7

**Example 2:**

**Input:** " 3/2 "

**Output:** 1

**Example 3:**

**Input:** " 3+5 / 2 "

**Output:** 5

**Note:**

* You may assume that the given expression is always valid.
* **Do not** use the eval built-in library function.

### **Analysis**

We also use stack to parse here.

/// <summary>

/// Leet code #227. Basic Calculator II

/// </summary>

inline int LeetCode::calculateII(int a, int b, char op)

{

if (op == '+')

{

return a + b;

}

else if (op == '-')

{

return a - b;

}

else if (op == '\*')

{

return a \* b;

}

else // (op == '/')

{

return a / b;

}

}

/// <summary>

/// Leet code #227. Basic Calculator II

/// Implement a basic calculator to evaluate a simple expression string.

/// The expression string contains only non-negative integers, +, -, \*, /

/// operators and empty spaces . The integer division should truncate toward

/// zero.

/// You may assume that the given expression is always valid.

/// Some examples:

/// "3+2\*2" = 7

/// " 3/2 " = 1

/// " 3+5 / 2 " = 5

/// Note: Do not use the eval built-in library function.

/// </summary>

int LeetCode::calculateII(string s)

{

int num = 0;

int product = 1;

stack<int> nums;

stack<char> ops;

for (size\_t i = 0; i < s.size(); i++)

{

char c = s[i];

if (isspace(c)) continue;

if (isdigit(s[i]))

{

num = num \* 10 + (c - '0');

}

else

{

while (!ops.empty())

{

if ((ops.top() == '\*') || (ops.top() == '/') ||

(c == '+') || (c == '-'))

{

num = calculateII(nums.top(), num, ops.top());

nums.pop();

ops.pop();

}

else

{

break;

}

}

nums.push(num);

num = 0;

ops.push(c);

}

}

while (!ops.empty())

{

num = calculateII(nums.top(), num, ops.top());

nums.pop();

ops.pop();

}

return num;

}

## 772. Basic Calculator III

Hard

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open ( and closing parentheses ), the plus + or minus sign -, **non-negative** integers and empty spaces .

The expression string contains only non-negative integers, +, -, \*, /operators , open ( and closing parentheses ) and empty spaces . The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [-2147483648, 2147483647].

Some examples:

"1 + 1" = 2

" 6-4 / 2 " = 4

"2\*(5+5\*2)/3+(6/2+8)" = 21

"(2+6\* 3+5- (3\*14/7+2)\*5)+3"=-12

### Analysis

We use a different method here which is called interpreter pattern which is described here.

<https://sourcemaking.com/design_patterns/interpreter>

The key idea of interpreter is that we given the current state and next token we see, we will expect certain command, we parse and execute that command and accumulate the result.

For example here, when we see ‘+’, ‘-’ we get next factor and accumulate result in expression, when we see ‘\*’ or ‘/’ we get next term and convert result to factor. If we see ‘(‘ we start another inner expression which ending by ‘)’.

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

string LeetCode::parseExpressionToken(string s, int& index)

{

string result;

while (index < (int)s.size())

{

if (isalnum(s[index]))

{

result.push\_back(s[index]);

index++;

}

else if (isspace(s[index]))

{

index++;

}

else

{

if (!result.empty()) break;

else

{

result = s[index];

index++;

break;

}

}

}

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

int LeetCode::parseTerm(vector<string>& tokens, int& index)

{

int result = 0;

if (isdigit(tokens[index][0]))

{

result = atoi(tokens[index].c\_str());

}

else if (tokens[index] == "(")

{

index++;

result = parseExpression(tokens, index);

// when return with ")" left there

}

index++;

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

int LeetCode::parseFactor(vector<string>& tokens, int& index)

{

int result = parseTerm(tokens, index);

while (index < (int)tokens.size())

{

if (tokens[index] == "\*")

{

index++;

result \*= parseTerm(tokens, index);

}

else if (tokens[index] == "/")

{

index++;

result /= parseTerm(tokens, index);

}

else

{

break;

}

}

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

/// </summary>

int LeetCode::parseExpression(vector<string>& tokens, int& index)

{

int result = parseFactor(tokens, index);

while (index < (int)tokens.size())

{

if (tokens[index] == "+")

{

index++;

result += parseFactor(tokens, index);

}

else if (tokens[index] == "-")

{

index++;

result -= parseFactor(tokens, index);

}

else

{

break;

}

}

return result;

}

/// <summary>

/// Leet code #772. Basic Calculator III

///

/// Implement a basic calculator to evaluate a simple expression string.

/// The expression string may contain open ( and closing parentheses ),

/// the plus + or minus sign -, non-negative integers and empty spaces.

///

/// The expression string contains only non-negative integers, +, -, \*, /

/// operators , open ( and closing parentheses ) and empty spaces .

/// The integer division should truncate toward zero.

///

/// You may assume that the given expression is always valid. All

/// intermediate results will be in the range of [-2147483648, 2147483647].

///

/// Some examples:

///

/// "1 + 1" = 2

/// " 6-4 / 2 " = 4

/// "2\*(5+5\*2)/3+(6/2+8)" = 21

/// "(2+6\* 3+5- (3\*14/7+2)\*5)+3"=-12

/// </summary>

int LeetCode::calculateIII(string s)

{

vector<string> tokens;

int index = 0;

while (index < (int)s.size())

{

tokens.push\_back(parseExpressionToken(s, index));

}

index = 0;

int result = parseExpression(tokens, index);

return result;

}

## 1096. Brace Expansion II

### Analysis

We also use interpreter pattern here. if we see the comma (,), we do union, if we see {}, we do cartesian product to expand.

Hard

Under a grammar given below, strings can represent a set of lowercase words.  Let's use R(expr) to denote the **set** of words the expression represents.

Grammar can best be understood through simple examples:

* Single letters represent a singleton set containing that word.
  + R("a") = {"a"}
  + R("w") = {"w"}
* When we take a comma delimited list of 2 or more expressions, we take the union of possibilities.
  + R("{a,b,c}") = {"a","b","c"}
  + R("{{a,b},{b,c}}") = {"a","b","c"} (notice the final set only contains each word at most once)
* When we concatenate two expressions, we take the set of possible concatenations between two words where the first word comes from the first expression and the second word comes from the second expression.
  + R("{a,b}{c,d}") = {"ac","ad","bc","bd"}
  + R("a{b,c}{d,e}f{g,h}") = {"abdfg", "abdfh", "abefg", "abefh", "acdfg", "acdfh", "acefg", "acefh"}

Formally, the 3 rules for our grammar:

* For every lowercase letter x, we have R(x) = {x}
* For expressions e\_1, e\_2, ... , e\_k with k >= 2, we have R({e\_1,e\_2,...}) = R(e\_1) ∪ R(e\_2) ∪ ...
* For expressions e\_1 and e\_2, we have R(e\_1 + e\_2) = {a + b for (a, b) in R(e\_1) × R(e\_2)}, where + denotes concatenation, and × denotes the cartesian product.

Given an expression representing a set of words under the given grammar, return the sorted list of words that the expression represents.

**Example 1:**

**Input:** "{a,b}{c,{d,e}}"

**Output:** ["ac","ad","ae","bc","bd","be"]

**Example 2:**

**Input:** "{{a,z},a{b,c},{ab,z}}"

**Output:** ["a","ab","ac","z"]

**Explanation:** Each distinct word is written only once in the final answer.

**Constraints:**

1. 1 <= expression.length <= 50
2. expression[i] consists of '{', '}', ','or lowercase English letters.
3. The given expression represents a set of words based on the grammar given in the description.

/// <summary>

/// Leet code #1096. Brace Expansion II

/// </summary>

void LeetCode::braceExpansionIIProduct(string &expression, size\_t& pos, unordered\_set<string> &result)

{

pos++;

braceExpansionIIUnion(expression, pos, result);

pos++;

}

/// <summary>

/// Leet code #1096. Brace Expansion II

/// </summary>

void LeetCode::braceExpansionIIUnion(string &expression, size\_t& pos, unordered\_set<string> &result)

{

while (pos <= expression.size())

{

unordered\_set<string> left;

if (pos == expression.size())

{

break;

}

else if (expression[pos] == '}')

{

break;

}

else if (expression[pos] == '{')

{

unordered\_set<string> right;

braceExpansionIIProduct(expression, pos, right);

if (result.empty()) result.insert("");

for (auto left\_str : result)

{

for (auto right\_str : right)

{

left.insert(left\_str + right\_str);

}

}

result.clear();

result.insert(left.begin(), left.end());

}

else if (isalpha(expression[pos]))

{

if (result.empty()) result.insert("");

for (auto left\_str : result)

{

left\_str.push\_back(expression[pos]);

left.insert(left\_str);

}

pos++;

result.clear();

result.insert(left.begin(), left.end());

}

else if (expression[pos] == ',')

{

unordered\_set<string> right;

pos++;

braceExpansionIIUnion(expression, pos, right);

for (auto right\_str : right)

{

result.insert(right\_str);

}

}

}

}

/// <summary>

/// Leet code #1096. Brace Expansion II

///

/// Under a grammar given below, strings can represent a set of lowercase

/// words. Let's use R(expr) to denote the set of words the expression

/// represents.

///

/// Grammar can best be understood through simple examples:

///

/// Single letters represent a singleton set containing that word.

/// R("a") = {"a"}

/// R("w") = {"w"}

/// When we take a comma delimited list of 2 or more expressions, we take

/// the union of possibilities.

/// R("{a,b,c}") = {"a","b","c"}

/// R("{{a,b},{b,c}}") = {"a","b","c"} (notice the final set only contains

/// each word at most once)

/// When we concatenate two expressions, we take the set of possible

/// concatenations between two words where the first word comes from the

/// first expression and the second word comes from the second expression.

/// R("{a,b}{c,d}") = {"ac","ad","bc","bd"}

/// R("{a{b,c}}{{d,e}f{g,h}}") = R("{ab,ac}{dfg,dfh,efg,efh}") =

/// {"abdfg", "abdfh", "abefg", "abefh", "acdfg", "acdfh", "acefg",

/// "acefh"}

/// Formally, the 3 rules for our grammar:

///

/// For every lowercase letter x, we have R(x) = {x}

/// For expressions e\_1, e\_2, ... , e\_k with k >= 2, we have

/// R({e\_1,e\_2,...}) = R(e\_1) ∪ R(e\_2) ∪ ...

/// For expressions e\_1 and e\_2, we have R(e\_1 + e\_2) = {a + b for (a, b)

/// in R(e\_1) × R(e\_2)}, where + denotes concatenation, and × denotes the

/// cartesian product.

/// Given an expression representing a set of words under the given

/// grammar, return the sorted list of words that the expression

/// represents.

///

/// Example 1:

///

/// Input: "{a,b}{c{d,e}}"

/// Output: ["acd","ace","bcd","bce"]

///

/// Example 2:

///

/// Input: "{{a,z},a{b,c},{ab,z}}"

/// Output: ["a","ab","ac","z"]

/// Explanation: Each distinct word is written only once in the final

/// answer.

///

/// Constraints:

/// 1. 1 <= expression.length <= 50

/// 2. expression[i] consists of '{', '}', ','or lowercase English letters.

/// 3. The given expression represents a set of words based on the grammar

/// given in the description.

/// </summary>

vector<string> LeetCode::braceExpansionII(string expression)

{

vector<string> result;

unordered\_set<string> left;

size\_t pos = 0;

braceExpansionIIUnion(expression, pos, left);

for (auto str : left) result.push\_back(str);

sort(result.begin(), result.end());

return result;

}

## 1106. Parsing A Boolean Expression

Hard

Return the result of evaluating a given boolean expression, represented as a string.

An expression can either be:

* "t", evaluating to True;
* "f", evaluating to False;
* "!(expr)", evaluating to the logical NOT of the inner expression expr;
* "&(expr1,expr2,...)", evaluating to the logical AND of 2 or more inner expressions expr1, expr2, ...;
* "|(expr1,expr2,...)", evaluating to the logical OR of 2 or more inner expressions expr1, expr2, ...

**Example 1:**

**Input:** expression = "!(f)"

**Output:** true

**Example 2:**

**Input:** expression = "|(f,t)"

**Output:** true

**Example 3:**

**Input:** expression = "&(t,f)"

**Output:** false

**Example 4:**

**Input:** expression = "|(&(t,f,t),!(t))"

**Output:** false

**Constraints:**

* 1 <= expression.length <= 20000
* expression[i] consists of characters in {'(', ')', '&', '|', '!', 't', 'f', ','}.
* expression is a valid expression representing a boolean, as given in the description.

### Analysis

Here is another interpreter pattern. Based on the operator you do AND, OR or NOT.

/// <summary>

/// Leet code #1106. Parsing A Boolean Expression

/// </summary>

bool LeetCode::parseBoolExpr(string expression, int &pos)

{

if (expression[pos] == 't')

{

pos++;

return true;

}

else if (expression[pos] == 'f')

{

pos++;

return false;

}

else if (expression[pos] == '&')

{

return parseBoolExprAnd(expression, pos);

}

else if (expression[pos] == '|')

{

return parseBoolExprOr(expression, pos);

}

else // (expression[pos] == '!')

{

return parseBoolExprNot(expression, pos);

}

}

/// <summary>

/// Leet code #1106. Parsing A Boolean Expression

/// </summary>

bool LeetCode::parseBoolExprNot(string expression, int &pos)

{

pos++; // skip !

pos++; // skip (

bool result = !parseBoolExpr(expression, pos);

pos++; // skip )

return result;

}

/// <summary>

/// Leet code #1106. Parsing A Boolean Expression

/// </summary>

bool LeetCode::parseBoolExprOr(string expression, int &pos)

{

pos++; // skip |

pos++; // skip (

bool result = parseBoolExpr(expression, pos);

while (expression[pos] != ')')

{

if (expression[pos] == ',')

{

pos++;

bool next = parseBoolExpr(expression, pos);

result = result || next;

}

}

pos++; // skip )

return result;

}

/// <summary>

/// Leet code #1106. Parsing A Boolean Expression

/// </summary>

bool LeetCode::parseBoolExprAnd(string expression, int &pos)

{

pos++; // skip |

pos++; // skip (

bool result = parseBoolExpr(expression, pos);

while (expression[pos] != ')')

{

if (expression[pos] == ',')

{

pos++;

bool next = parseBoolExpr(expression, pos);

result = result && next;

}

}

pos++; // skip )

return result;

}

/// <summary>

/// Leet code #1106. Parsing A Boolean Expression

///

/// Return the result of evaluating a given boolean expression, represented

/// as a string.

///

/// An expression can either be:

///

/// "t", evaluating to True;

/// "f", evaluating to False;

/// "!(expr)", evaluating to the logical NOT of the inner expression expr;

/// "&(expr1,expr2,...)", evaluating to the logical AND of 2 or more inner

/// expressions expr1, expr2, ...;

/// "|(expr1,expr2,...)", evaluating to the logical OR of 2 or more inner

/// expressions expr1, expr2, ...

///

/// Example 1:

/// Input: expression = "!(f)"

/// Output: true

///

/// Example 2:

/// Input: expression = "|(f,t)"

/// Output: true

///

/// Example 3:

/// Input: expression = "&(t,f)"

/// Output: false

///

/// Example 4:

/// Input: expression = "|(&(t,f,t),!(t))"

/// Output: false

///

/// Constraints:

///

/// 1. 1 <= expression.length <= 20000

/// 2. expression[i] consists of characters in {'(', ')', '&', '|', '!',

/// 't', 'f', ','}.

/// 3. expression is a valid expression representing a boolean, as given in

/// the description.

/// </summary>

bool LeetCode::parseBoolExpr(string expression)

{

int pos = 0;

return parseBoolExpr(expression, pos);

}

**65. Valid Number**

Hard

Validate if a given string can be interpreted as a decimal number.

Some examples:  
"0" => true  
" 0.1 " => true  
"abc" => false  
"1 a" => false  
"2e10" => true  
" -90e3   " => true  
" 1e" => false  
"e3" => false  
" 6e-1" => true  
" 99e2.5 " => false  
"53.5e93" => true  
" --6 " => false  
"-+3" => false  
"95a54e53" => false

**Note:** It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one. However, here is a list of characters that can be in a valid decimal number:

* Numbers 0-9
* Exponent - "e"
* Positive/negative sign - "+"/"-"
* Decimal point - "."

Of course, the context of these characters also matters in the input.

**Update (2015-02-10):**  
The signature of the C++ function had been updated. If you still see your function signature accepts a const char \* argument, please click the reload button to reset your code definition.

/// <summary>

/// Leet code #65. Valid Number

/// Validate if a given string is numeric.

/// Some examples:

/// "0" => true

/// " 0.1 " => true

/// "abc" => false

/// "1 a" => false

/// "2e10" => true

/// Note: It is intended for the problem statement to be ambiguous.

/// You should gather all requirements up front before implementing one.

/// </summary>

bool LeetCode::isValidNumber(string s)

{

bool digit = false;

bool end = false;

bool exp = false;

bool point = false;

bool sign = false;

for (size\_t i = 0; i < s.size(); i++)

{

if (isdigit(s[i]))

{

if (end) return false;

digit = true;

}

else if (s[i] == 'e')

{

if ((!digit) || end || exp) return false;

digit = false;

point = false;

sign = false;

exp = true;

}

else if (isspace(s[i]))

{

if (point || digit || exp || sign) end = true;

}

else if ((s[i] == '+') || (s[i] == '-'))

{

if (digit || point || sign) return false;

sign = true;

}

else if (s[i] == '.')

{

if (exp || point || end) return false;

point = true;

}

else

{

return false;

}

}

return digit;

}

/// <summary>

/// Leet code #65. Valid Number

/// Validate if a given string is numeric.

/// Some examples:

/// "0" => true

/// " 0.1 " => true

/// "abc" => false

/// "1 a" => false

/// "2e10" => true

/// Note: It is intended for the problem statement to be ambiguous.

/// You should gather all requirements up front before implementing one.

/// </summary>

bool LeetCode::isValidNumberII(string s)

{

typedef enum { start, sign, integer, decimal\_start, decimal, exp\_start, exp\_sign, exp\_int, end } number\_state;

number\_state state = start;

for (size\_t i = 0; i < s.size(); i++)

{

if (state == start)

{

if (isspace(s[i])) state = start;

else if ((s[i] == '+') || (s[i] == '-')) state = sign;

else if (isdigit(s[i])) state = integer;

else if (s[i] == '.') state = decimal\_start;

else return false;

}

else if (state == sign)

{

if (isdigit(s[i])) state = integer;

else if (s[i] == '.') state = decimal\_start;

else return false;

}

else if (state == integer)

{

if (isdigit(s[i])) state = integer;

else if (s[i] == '.') state = decimal;

else if (s[i] == 'e') state = exp\_start;

else if (isspace(s[i])) state = end;

else return false;

}

else if (state == decimal\_start)

{

if (isdigit(s[i])) state = decimal;

else return false;

}

else if (state == decimal)

{

if (isdigit(s[i])) state = decimal;

else if (s[i] == 'e') state = exp\_start;

else if (isspace(s[i])) state = end;

else return false;

}

else if (state == exp\_start)

{

if ((s[i] == '+') || (s[i] == '-')) state = exp\_sign;

else if (isdigit(s[i])) state = exp\_int;

else return false;

}

else if (state == exp\_sign)

{

if (isdigit(s[i])) state = exp\_int;

else return false;

}

else if (state == exp\_int)

{

if (isdigit(s[i])) state = exp\_int;

else if (isspace(s[i])) state = end;

else return false;

}

else if (state == end)

{

if (isspace(s[i])) state = end;

else return false;

}

}

if ((state == start) || (state == decimal\_start) || (state == sign) ||

(state == exp\_start) || (state == exp\_sign))

{

return false;

}

else

{

return true;

}

}

## State Machine

The above expression parsing use the concept of state machine, it is implemented by the interpreter pattern, which is for every state, we use a function to process it.

In other cases, we can simply assume the state machines are list of enumeration, which move from one state to another, please notice that the state transition are limited, so we can identify the invalid state which we should never reach by such a path.

Please look at the following problem.

## 65. Valid Number

Hard

Validate if a given string can be interpreted as a decimal number.

Some examples:  
"0" => true  
" 0.1 " => true  
"abc" => false  
"1 a" => false  
"2e10" => true  
" -90e3   " => true  
" 1e" => false  
"e3" => false  
" 6e-1" => true  
" 99e2.5 " => false  
"53.5e93" => true  
" --6 " => false  
"-+3" => false  
"95a54e53" => false

**Note:** It is intended for the problem statement to be ambiguous. You should gather all requirements up front before implementing one. However, here is a list of characters that can be in a valid decimal number:

* Numbers 0-9
* Exponent - "e"
* Positive/negative sign - "+"/"-"
* Decimal point - "."

Of course, the context of these characters also matters in the input.

**Update (2015-02-10):**  
The signature of the C++ function had been updated. If you still see your function signature accepts a const char \* argument, please click the reload button to reset your code definition.

### Analysis

The best approach is to use state machine to parse the string, and when we see invalid state, we return false. If the whole string is successfully parsed. We return true.