Convert Prefix Expressions. Recursively !

Preliminary

First, we must understand what are prefix expressions.

Discovered in 1924 by a Polish logician (and thus also called Polish Notation), prefix expressions are a form of notation for mathematical expressions, in which a binary operator precedes both operands. This differs from the familiar infix expressions we are used to, where both operands surround a binary operator. Here's an example:

Infix Expression: **A+B** Prefix Expression: **+AB**

Before we begin, it would be useful to look at the formal grammar notation of prefix expressions.

$$Operand \quad \Rightarrow \quad Operand \quad | \quad Operator \quad Operand \quad Operand \quad | \quad [A-Z] \tag{1}$$

$$Operator \quad \Rightarrow \quad + \quad | \quad - \quad | \quad \times \quad | \quad \div \tag{2}$$

Clearly this is recursive. It helps to think of prefix strings in terms of prefix strings themselves. To wit, a prefix string might be one of two simple forms:

- An operator followed by two operands who might themselves be prefix strings.
- A single alphabet

Clearly, the general structure of prefix strings (and indeed of postfix and even infix strings) is recursive in nature. Our approach to solving this shall also be recursive, with 2 main functions:

1. **convert()**
2. **find()**

find()

The **find()** function must take as input a string, **S**. The assumption is that there is a prefix sub expression at the start of the string[2], **S**. **find()** must return the length of this sub expression. If no prefix sub expression is found, return 0. For example, suppose our string is **+AB+CD**, then **find()** must return the value 3.

How would **find()** do this ? One method could be to use a stack. However, as previously noted, prefix expressions have a recursive structure, so, we shall write a recursive version of **find()**. To make our job easier, we define the operation **len()**, which takes as input, a string and returns it's length.

- If the length of **S** is 0, return 0
- If the first character is an alphabet, return 1
- If the first character is an operator, then for a prefix expression to exist, there must be two operands following it. However, each of the operands may itself be a (nontrivial) prefix expression. Thus the length to be returned is $\textbf{len}(operand1) + \textbf{len}(operand2) + \textbf{len}(operator) \Rightarrow \textbf{len}(operand1) + \textbf{len}(operand2) + 1$. So, we perform the following actions:
  1. Copy **S** into a new string, $\mathbf{S}_0$., leaving out the operator (essentially *slicing* the string, i.e. $\mathbf{S}_0 = \mathbf{S}[1:-1]$)
  2. Pass $\mathbf{S}_0$ to the **find()** function and store the return value in variable **m**
  3. If $\mathbf{m} = 0$, the length of our first operand is 0 i.e. the first operand **DOES NOT EXIST**. So we return 0.
  4. Copy **S** into a new string $\mathbf{S}_1$, leaving out the first $m + 1$ characters (which is the same as $\mathbf{S}_1 = \mathbf{S}[m + 1 : -1]$
  5. Pass $\mathbf{S}_1$ to **find()** and store the returned value in variable **n**
  6. If $\mathbf{n} = 0$, return 0.
  7. Return **m+n+1**

```
1  int find(char str[])
2  {
3                  char temp[MAXLEN];
4                  int length;
5
6                  if ((length = strlen(str)) == 0)
7                                  return 0;
8
9                  if (isalpha(str[0]))
10                                                 return 1;
11
12                 substr(str, temp, 1, length - 1);
13                 int m = find(temp);
14                 if (m == 0)
15                                 return 0;
16
17                 substr(str, temp, m+1, length-1);
18                 int n = find(temp);
19                 if (n == 0)
20                                 return 0;
21
22                 return m+n+1;
23 }
```

A few points are worth noting.

- If the first character of the string is an alphabet, **find()** returns 1. **This is a base case.**
- If the first character of the string is an operator, **find()** will be called twice, on smaller parts of the string, to look for the operands. **Thus, find() descends a level into the recursion every time it encounters an operator**
- Once **find()** comes across a part of the expression in the form of **operator alphabet alphabet**, it returns a numeric value (3), which propagates up the recursion, increasing at each level.

Now what happens if the prefix expression is invalid ? We could exit with an error, but since **find()** is called recursively on smaller and smaller parts of the string, it's possible that it might be passed a smaller portion $\mathbf{S}_0$ of a valid prefix string **S**. So, we leave the validation of the input string to **convert()**. All that **find()** does is check if there exists a valid subexpression at the start of the string, and return it's length. In case, no valid subexpression exists, **find()** returns 0 as a value. This is treated as a signal by **convert()** which will then exit with the INVALID STRING error. It would help to visualize this, so, lets trace the execution of **find()** for some invalid prefix strings.

- If our prefix expression has an alphabet(or a whole operand or more) less than what is needed - eg ***+AB-C**. In this case, we have an operator (*), we have the first operand (+**AB**), but our second operand is incomplete(**-C**). Let's trace the calls for **find()** and what they do.
  - **find(*+AB-C)** : The first character is an operator. So we must descend into recursion.
    - **find(+AB-C)**: The first character is an operator, so again we descend a level
      - **find(AB-C)** : The first character is an alphabet, so this call returns value 1
      - **find(B-C)** : The first character is an alphabet, so this call returns value 1

      This call returns value **(1+1+1 = 3)**. We now make another call at this level, leaving out the first 3 characters of the string at this level.
    - **find(-C)** : The first character is an operator, so we must descend a level into the recursion
      - **find(C)** : The first character is an alphabet, so the call returns value 1
      - **find()**: black) Our input is an empty string, so find returns 0

      Since **find()**returned 0, the call at the current level (**find(-C)**) will also return 0

    Since **find(-C)** returned 0, **find(*+AB-C)**will also return 0. So, the final returned value is 0.

- If the expression is of the form **ABC**, with a missing operator. In this case, the string itself is an invalid prefix expression, but it has a valid prefix subexpression at the start of the string, ie **A**. So, the first call to **find()** will return value 1.
- Our prefix expression has extra alphabets or operators, eg - **+ABC**. Here **C** is an extra alphabet. In this case, **+AB** by themselves form a valid prefix expression. Since **find()** is only concerned with a valid prefix subexpression at the start of the string, it will return the length of **+AB**, ie 3

We will also trace the execution of the **convert()** function on these inputs, to see how it handles invalid input.

convert()

The **convert()** function shall take as input, a prefix string, and return a postfix string. To accomplish this, it needs to perform several things.

- First, if the string is of length 1 and a single alphabet, just return the alphabet as it is
- Second, if the string is of length > 1, we presume that the first character is an operator (and if not, we will handle it as an error later), and store it in the **op** variable (which is a character array).
    1. If the string is a single
    2. Copy the string from index 1 to the end, and store it in a variable **temp**.
    3. Call **find()** on **temp**, to find the length of the first operand. Store the return value in variable **m**.
    4. Copy the remaining string (from index m+1 to end), and store it in the variable **temp()**
    5. Call **find()** on **temp** to find the length of the first operand. Store the return value in a variable **n**
    6. Check if the string is valid - if either of **m** or **n** is 0, if their sum is less than the length of the string, if the first character in the string is not an alphabet, then exit with the INVALID STRING eror.
    7. Arrange both of the converted operands into the postfix order and return.

```
void convert(char prefix[], char postfix[], int tnum)
{
        char opnd1[MAXLEN], opnd2[MAXLEN];
        char post1[MAXLEN], post2[MAXLEN];
        char temp[MAXLEN];
        char op[1];
        int length;
        int m, n;


        if ((length = strlen(prefix)) == 1 && isalpha(prefix[0])){
                postfix[0] = prefix[0];
                postfix[1] = '\0';
                return;
                }

        op[0] = prefix[0];
        op[1] = '\0';

        substr(prefix, temp, 1, length-1);
        m = find(temp);
        substr(prefix, temp, m+1, length-1);
        n = find(temp);


        if (m == 0 || n == 0 || m+n+1 != length || isalpha(prefix[0]) ){
                printf("Illegal String\n");
                exit(1);
        }

```

```
31          substr(prefix, opnd1, 1, m);
32          substr(prefix, opnd2, m+1, length-1);
33
34
35          convert(opnd1, post1, tnum+1);
36          convert(opnd2, post2, tnum+1);
37
38          strcat(post1, post2);
39          strcat(post1, op);
40          substr(post1, postfix, 0, length-1);
41          return ;
42  }
```

Again, let's trace the execution of **convert()** on our prior inputs.

- **convert(*+AB-C)** : The function will extract the operator (+) and call **find(+AB-C)**, with return value 3. The function will now call **find(-C** with return value 0. This signals invalid input, and the function will exit with an error.
- **convert(ABC)** : The function will extract the operator (A) and call **find(BD)** with return value 1. The function will then call **find(C)** with return value 1. But then it will check to see if the first character of the string is an alphabet, which is true. This will cause it to exit with an error.
- **convert(+ABC)** : The function will extract the operator + and then call **find(ABC)** with return value 1. Subsequently, it will call **find(BC)** with return value 1. But the sum of the returned values + 1 (for the operator) is 3, whereas the string length is 4. Thus the function will exit with an error.

TL;DR

The convert() function takes an input string, splits it into an operator, and 2 operands, with the help of the **find()** function. Then it calls itself twice, once with each operand as it's argument. Then it takes the returned values, arranges them in postfix order and then returns.

In case the input is invalid, the **find()** function will return 0, or the sum of the values returned by the **find()** function will be less than the length of the string, or the first character of the string will be an alphabet. **convert()** will exit with an error if the input is invalid.

The **find()** function will return the length of the prefix string at the start of the input string. It will return 0 if the stringlength is 0. It might also return 0, if the input is invalid.