**Parenthesis Matching**

The problem of finding matching parentheses must be solved in many computing applications. For example, consider a Java/C/C++/C# compiler.

Matching parentheses (( and )), brackets ([ and ]), and braces ({ and }) delimit various parts of the source code. In order for these parts to be interpreted correctly, the compiler must be able to determine how these different kinds of parentheses match up with each other. Another example is processing structured data stored in XML format. Different parts of such a data set are delimited by nested begin tags like <summary> and end tags like </summary> (documentation comments in C# code are in XML format). These tags are essentially different kinds of parentheses that need to be matched.

We will restrict our attention to parentheses, brackets, and braces. We will call all six of these characters “parentheses”, but will divide them into three types. Each type then has an opening parenthesis and a closing parenthesis. We will define a **string** restricted to these six characters to be *matched* (or *balanced*) if we can repeatedly remove an opening parenthesis and a closing parenthesis of the same type to its immediate right until there are no more parentheses.

For example, suppose we have the **string**, “([]{()[]})[{}]”. We can apply the matching-pair removal process described above as follows (blank space is inserted to make it easier to see which parentheses are removed):

 ([]{()[]})[{}]

 ( {()[]})[{}]

 ( { []})[{}]

 ( { })[{}]

 ( )[{}]

 [{}]

 [ ]

Hence, this **string** is matched. On the other hand, consider the **string**, “([]{()[])}[{}]”. When we apply the above process to this **string**, we obtain:

 ([]{()[])}[{}]

 ( {()[])}[{}]

 ( { [])}[{}]

 ( { )}[{}]

 ( { )}[ ]

 ( { )}

and we can go no further. Hence, this **string** is not matched.

We can extend the definition of a matched **string** to include other characters if we first remove all other characters before we begin the matching-pair removal process. In what follows, we will focus on the problem of determining whether a given **string** is matched.

The matching-pair removal process shown above gives us an algorithm for determining whether a **string** is matched. However, if implemented directly, it isn’t very efficient. Changes to a **string** are inefficient because the entire **string** must be reconstructed. We could use a **StringBuilder**, but even then, removing characters is inefficient, as all characters to the right of the removed character must be moved to take its place. Even if we simply change parentheses to blanks, as we did in the above example, searching for matching pairs is still rather expensive.

What we would like to do instead is to find a way to apply the matching-pair removal process while scanning the **string** once. As we are scanning the **string**, we don’t want to spend time searching for a matching pair. We can do this if, while scanning the **string**, we keep all unmatched opening parentheses in a stack. Then the parenthesis at the top of the stack will always be the rightmost unmatched opening parenthesis. Thus, starting with an empty stack, we do the following for each character in the **string**:

* If the character is a opening parenthesis, push it onto the stack.
* If the character is a closing parenthesis:
	+ If the stack is nonempty, and the current character matches the character on top of the stack, remove the character from the top of the stack.
	+ Otherwise, the **string** is not matched.
* Ignore all other characters.

If the stack is empty when the entire **string** has been processed, then the **string** is matched; otherwise, it is not.

For example, consider the **string**, “{a[b]([c]){de}}f[(g)]”. In what follows, we will simulate the above algorithm, showing the result of processing each character on a separate line. The portion of the line with an orange background will be the stack contents, with the top element shown at the right. We will insert blank space in the orange area for clarity, but the stack will only contain opening parentheses. The first character with a black background is the character currently being processed.

{a[b]([c]){de}}f[(g)] --- an opening parenthesis - push it onto the stack

{a[b]([c]){de}}f[(g)] --- ignore

{ [b]([c]){de}}f[(g)] --- push onto stack

{ [b]([c]){de}}f[(g)] --- ignore

{ [ ]([c]){de}}f[(g)] --- closing parenthesis that matches the top - remove top

{ ([c]){de}}f[(g)] --- push onto stack

{ ([c]){de}}f[(g)] --- push onto stack

{ ([c]){de}}f[(g)] --- ignore

{ ([ ]){de}}f[(g)] --- a match - remove top

{ ( ){de}}f[(g)] --- a match - remove top

{ {de}}f[(g)] --- push onto stack

{ {de}}f[(g)] --- ignore

{ { e}}f[(g)] --- ignore

{ { }}f[(g)] --- a match - remove top

{ }f[(g)] --- a match - remove top

 f[(g)] --- ignore

 [(g)] --- push onto stack

 [(g)] --- push onto stack

 [(g)] --- ignore

 [( )] --- a match - remove top

 [ ] --- a match - remove top

 --- end of string and stack empty - matched string

If at any time during the above process we had encountered a closing parenthesis while the stack was empty, this would have indicated that this closing parenthesis has no matching opening parenthesis. In this case, we would have stopped immediately, determining that the string is not matched. Likewise, if we had encountered a closing parenthesis that did not match the parentheis at the top of the stack, this would have indicated a mismatched pair. Again, we would have stopped immediately. Finally, if we had reached the end of the string with a nonempty stack, this would have indicated that we had at least one opening parenthesis that was never matched. We would have again determined that the string is not matched.