Problem 591: Tag Validator

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given a string representing a code snippet, implement a tag validator to parse the code and return whether it is valid.

A code snippet is valid if all the following rules hold:

The code must be wrapped in a

valid closed tag

. Otherwise, the code is invalid.

A

closed tag

(not necessarily valid) has exactly the following format :

<TAG_NAME>TAG_CONTENT</TAG_NAME>

. Among them,

<TAG_NAME>

is the start tag, and

</TAG_NAME>

is the end tag. The TAG_NAME in start and end tags should be the same. A closed tag is

valid

if and only if the TAG_NAME and TAG_CONTENT are valid.

A

valid

TAG_NAME

only contain

upper-case letters

, and has length in range [1,9]. Otherwise, the

TAG_NAME

is

invalid

.

A

valid

TAG_CONTENT

may contain other

valid closed tags

,

cdata

and any characters (see note1)

EXCEPT

unmatched

<

, unmatched start and end tag, and unmatched or closed tags with invalid TAG_NAME. Otherwise, the

TAG_CONTENT

is

invalid

.

A start tag is unmatched if no end tag exists with the same TAG_NAME, and vice versa. However, you also need to consider the issue of unbalanced when tags are nested.

A

<

is unmatched if you cannot find a subsequent

>

. And when you find a

<

or

</

, all the subsequent characters until the next

>

should be parsed as TAG_NAME (not necessarily valid).

The cdata has the following format :

<![CDATA[CDATA_CONTENT]]>

. The range of

CDATA_CONTENT

is defined as the characters between

<![CDATA[

and the

first subsequent

]]>

.

CDATA_CONTENT

may contain

any characters

. The function of cdata is to forbid the validator to parse

CDATA_CONTENT

, so even it has some characters that can be parsed as tag (no matter valid or invalid), you should treat it as

regular characters

.

Example 1:

Input:

code = "<DIV>This is the first line <![CDATA[<div>]]></DIV>"

Output:

true

Explanation:

The code is wrapped in a closed tag : <DIV> and </DIV>. The TAG_NAME is valid, the TAG_CONTENT consists of some characters and cdata. Although CDATA_CONTENT has an unmatched start tag with invalid TAG_NAME, it should be considered as plain text, not parsed as a tag. So TAG_CONTENT is valid, and then the code is valid. Thus return true.

Example 2:

Input:

code = "<DIV>>> ![cdata[]] <![CDATA[<div>]>]]>]]>>]</DIV>"

Output:

true

Explanation:

We first separate the code into : start_tag|tag_content|end_tag. start_tag ->

"<DIV>"

end_tag ->

"</DIV>"

tag_content could also be separated into : text1|cdata|text2. text1 ->

">> ![cdata[]] "

cdata ->

"<![CDATA[<div>]>]]>"

, where the CDATA_CONTENT is

"]>"

text2 ->

"]]>>]"

The reason why start_tag is NOT

"<DIV>>>"

is because of the rule 6. The reason why cdata is NOT

"<![CDATA[<div>]>]]>]]>"

is because of the rule 7.

Example 3:

Input:

code = "<A> <B> </A> </B>"

Output:

false

Explanation:

Unbalanced. If "<A>" is closed, then "<B>" must be unmatched, and vice versa.

Constraints:

1 <= code.length <= 500

code

consists of English letters, digits,

'<'

,

'>'

,

'/'

,

'!'

,

'['

,

']'

,

'.'

, and

''

.

Code Snippets

C++:

```
class Solution {
public:
bool isValid(string code) {

}
};
```

Java:

```
class Solution {
public boolean isValid(String code) {

}
}
```

Python3:

```
class Solution:
def isValid(self, code: str) -> bool:
```

Python:

```
class Solution(object):
def isValid(self, code):
"""
:type code: str
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {string} code
```

```
 * @return {boolean}
 */
var isValid = function(code) {

};
```

TypeScript:

```
function isValid(code: string): boolean {

};
```

C#:

```
public class Solution {
public bool IsValid(string code) {

}
}
```

C:

```
bool isValid(char* code) {

}
```

Go:

```
func isValid(code string) bool {

}
```

Kotlin:

```
class Solution {
fun isValid(code: String): Boolean {

}
}
```

Swift:

```
class Solution {
func isValid(_ code: String) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn is_valid(code: String) -> bool {

}
}
```

Ruby:

```
# @param {String} code
# @return {Boolean}
def is_valid(code)

end
```

PHP:

```
class Solution {

/**
* @param String $code
* @return Boolean
*/
function isValid($code) {

}
}
```

Dart:

```
class Solution {
bool isValid(String code) {

}
}
```

Scala:

```
object Solution {
def isValid(code: String): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec is_valid(code :: String.t) :: boolean
def is_valid(code) do

end
end
```

Erlang:

```
-spec is_valid(Code :: unicode:unicode_binary()) -> boolean().
is_valid(Code) ->
.
```

Racket:

```
(define/contract (is-valid code)
(-> string? boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Tag Validator
* Difficulty: Hard
* Tags: string, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class Solution {
public:
bool isValid(string code) {

}
};
```

Java Solution:

```
/**
* Problem: Tag Validator
* Difficulty: Hard
* Tags: string, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public boolean isValid(String code) {

}
}
```

Python3 Solution:

```
"""
Problem: Tag Validator
Difficulty: Hard
Tags: string, stack

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def isValid(self, code: str) -> bool:
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

```
class Solution(object):
def isValid(self, code):
"""
:type code: str
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Tag Validator
* Difficulty: Hard
* Tags: string, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {string} code
* @return {boolean}
*/
var isValid = function(code) {

};
```

TypeScript Solution:

```
/**
* Problem: Tag Validator
* Difficulty: Hard
* Tags: string, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
```

```
*/

function isValid(code: string): boolean {

};
```

C# Solution:

```
/*
* Problem: Tag Validator
* Difficulty: Hard
* Tags: string, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public bool IsValid(string code) {

}
}
```

C Solution:

```
/*
* Problem: Tag Validator
* Difficulty: Hard
* Tags: string, stack
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

bool isValid(char* code) {

}
```

Go Solution:

```
// Problem: Tag Validator
// Difficulty: Hard
// Tags: string, stack
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func isValid(code string) bool {

}
```

Kotlin Solution:

```
class Solution {
fun isValid(code: String): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func isValid(_ code: String) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Tag Validator
// Difficulty: Hard
// Tags: string, stack
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn is_valid(code: String) -> bool {

}
```

```
}
```

Ruby Solution:

```
# @param {String} code
# @return {Boolean}
def is_valid(code)

end
```

PHP Solution:

```
class Solution {

/**
* @param String $code
* @return Boolean
*/
function isValid($code) {

}
}
```

Dart Solution:

```
class Solution {
bool isValid(String code) {

}
}
```

Scala Solution:

```
object Solution {
def isValid(code: String): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec is_valid(code :: String.t) :: boolean
def is_valid(code) do

end
end
```

Erlang Solution:

```
-spec is_valid(Code :: unicode:unicode_binary()) -> boolean().
is_valid(Code) ->
.
```

Racket Solution:

```
(define/contract (is-valid code)
(-> string? boolean?)
)
```