2021 中央大學編譯器 BossAttack 1考題

注意事項

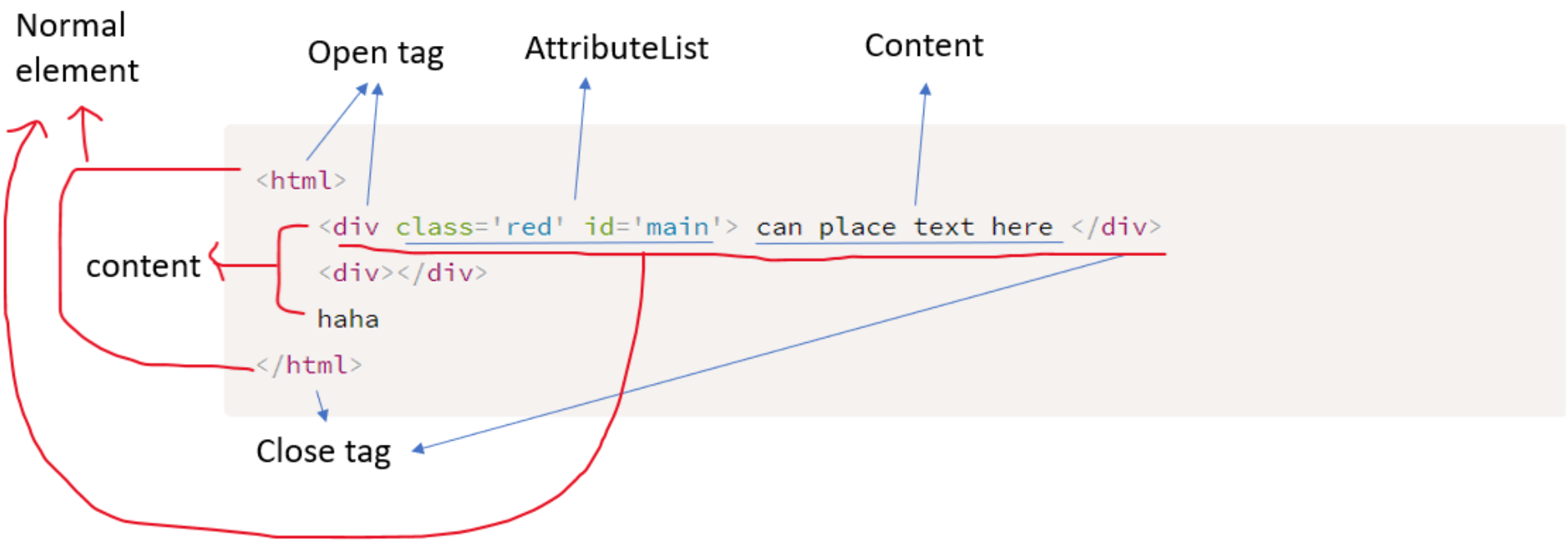
1. 此題不能使用一些lexer generator與parser generator這類的工具，如yacc、flex、bison等工具。
2. 此題的html parser需用recursive decedent parsing的方式來寫，不然不給分。

題目概述

要實作一個簡化版的html lexer與parser。html lexer負責將輸入的html切成token，再將切出來的tokens給html parser，藉由recursive decedent parsing來判斷html合法與否。
html是描述網頁的一種語言，下面是一個html的範例

```
<html>
    <div class='red' id='main'> can place text here </div>
    <div></div>
    haha
</html>
```

html主要就是由normal element組成，normal element的分解如下圖所示



組成normal element各部分的解釋如下

1. open tag與close tag
   open tag與close tag如下所示

   ```
   <div> </div>
   ```

   open tag和close tag由左右角括號(<>)組成，中間會放tag的名稱(div)，但close tag會在左角括號後多一個斜線(/)
   上面範例的open tag為`<div>`、close tag為`</div>`

   **此簡化版的html不會有「沒close tag的normal element」，ex: <input>、
**

2. attributeList
   可以有0~n個attribute(n>0)，attribute描述在open tag中，如下

   ```
   <div attribute1='test1' attribute2='test2' attribute3='test3'> </div>
   ```

   attributeList會在tag名稱的後面，上面範例有三個attribute - attribute1、 attribute2、 attribute3，其對應值分別為test1、test2、test3。

   此簡化版的html不會有attribute沒有值的情況， ex: `<div dark></div>`

3. content
   open tag和close tag中間用來放置normal element的content，可以放文字或normal element，放的數量0~n(n>0)都可。
   normal element有放三個東西的範例如下

```
<div> text<span>haha</span><div>nested div</div> </div>
```

在最頂層的div中依序放了文字、tag為span的normal element、tag為div的normal element

content沒放東西的範例如下

```
<div></div>
```

token的定義

Terminal	Regular Expression	Correct Example	Correct Example Token Value	Wrong Example
TAG_OPEN_SLASH	</	</	</	不提供
TAG_OPEN	<	<	<	不提供
TAG_CLOSE	>	>	>	不提供
TAG_EQUALS	=	=	=	不提供
SINGLE_QUOTE_STRING	'[^'<]*'	'compiler'	**compiler**	'com'piler'<
DOUBLE_QUOTE_STRING	"[^"<]*"	"compiler2"	**compiler2**	"compi"ler2"<
TAG_NAME	[a-z|A-Z|0-9]+	h1	h1	div_2
HTML_TEXT	[^<]+	text content	text content	text < content

上表的TOKEN定義，匹配優先度越上面越高，ex: `TAG_OPEN_SLASH` 的優先度高於 `TAG_OPEN`

特別注意SINGLE_QUOTE_STRING token的值不包含單引號、DOUBLE_QUOTE_STRING token的值不包含雙引號，其餘token的值都跟匹配到的一樣

grammar的定義

```
1   htmlDocument -> htmlElement htmlDocument $
2   htmlDocument -> λ

3   htmlElement -> TAG_OPEN TAG_NAME htmlAttributeList TAG_CLOSE htmlContent TAG_OPEN_SLASH TAG_NAME TAG_CLOSE

4   htmlContent -> htmlChardata htmlContent
5   htmlContent -> htmlElement htmlContent
6   htmlContent -> λ

7   htmlAttributeList -> htmlAttribute htmlAttributeList
8   htmlAttributeList -> λ

9   htmlAttribute -> TAG_NAME TAG_EQUALS attribute

10  htmlChardata -> HTML_TEXT

11  attribute -> DOUBLE_QUOTE_STRING
12  attribute -> SINGLE_QUOTE_STRING
```

lexer規則觸發的條件

lexer在切token的時候，會遇到下面的情況(*表示目前lexer讀字元的位置，剛切出TAG_OPEN token，下個要讀 `d` 字元)

```
<*div>text</div>
```

此時會有兩個TOKEN的regular expression都可以匹配 `d` 字元，那兩個TOKEN列在下面
TAG_NAME : [a-z|A-Z|0-9]+
HTML_TEXT : [^<]+
若單純使用regular expression來切token，會優先匹配結果較長的規則也就是HTML_TEXT，如下圖

```
<*div>text</div>
```

匹配TAG_NAME的結果為畫藍色底線的部分、匹配HTML_TEXT的結果為畫綠色底線的部分，由於HTML_TEXT匹配結果比較長，因此regular expression會匹配HTML_TEXT。但這樣的結果不是我們想要的。
為了解決此問題會讓lexer使用模式轉換和條件式觸發匹配規則。在此題的文法中可從第3, 7, 9條規則得知，`TAG_NAME` 一定會在 `TAG_OPEN` 與 `TAG_OPEN_SLASH` 後，以及 `TAG_CLOSE` 前。且在文法的第3, 4, 10條規則得知，`HTML_TEXT` 會在 `TAG_CLOSE` 後，以及 `TAG_OPEN` 與 `TAG_OPEN_SLASH` 前。
因此lexer可以只用一個tag模式來作為觸發token匹配的條件。
下表有完整定義lexer的模式轉換與觸發的條件(哪種token的匹配在哪種模式下才能觸發)

Terminal	Mode transition	match on which mode
TAG_OPEN_SLASH	enter tag mode	all
TAG_OPEN	enter tag mode	all
TAG_CLOSE	leave tag mode	all
TAG_EQUALS	no transition	all
SINGLE_QUOTE_STRING	no transition	tag mode
DOUBLE_QUOTE_STRING	no transition	tag mode
TAG_NAME	no transition	tag mode
HTML_TEXT	no transition	not in tag mode

當匹配TAG_OPEN_SLASH token和TAG_OPEN token時會進入tag模式，這時不能夠匹配HTML_TEXT，當匹配TAG_CLOSE token時會離開tag模式，這時不能匹配SINGLE_QUOTE_STRING、DOUBLE_QUOTE_STRING、TAG_NAME。

實際匹配的範例如下(*表示目前lexer讀字元的位置)，此範例利用名為inTag的bool變數來記錄目前是否在tag模式

Step	Current lexer position	Next Matched token	inTag
1	`*<div>text</div>`	`TAG_OPEN`	false
2	`<*div>text</div>`	TAG_NAME	true
3	`<div*>text</div>`	`TAG_CLOSE`	true
4	`<div>*text</div>`	HTML_TEXT	false
5	`<div>text*</div>`	`TAG_OPEN_SLASH`	false
6	`<div>text</*div>`	TAG_NAME	true
7	`<div>text</div*>`	`TAG_CLOSE`	true
8	`<div>text</div>*`	結束匹配	false

如果不太能理解的話有下面的cpp範例程式可以參考(有忽略部分程式)：

```cpp
// std::string input; // 存放html的字串
// ...
bool inTagDef = false;
for (int i = 0; i < input.length(); i++) {
    if (i < input.length() - 1 && input[i] == '<' && input[i + 1] == '/') {
        inTagDef = true;
        // ... generate TAG_OPEN_SLASH token
    }
    else if (input[i] == '<') {
        inTagDef = true;
        // ... generate TAG_OPEN token
    }
    else if (input[i] == '>') {
        inTagDef = false;
        // ... generate TAG_CLOSE token
    }
    else if (input[i] == '=') {
        // ... generate TAG_EQUALS token
    }
    else if (input[i] == '\'' && inTagDef == true) {
        // ... generate SINGLE_QUOTE_STRING token
    }
    else if (input[i] == '"' && inTagDef == true) {
        // ... generate DOUBLE_QUOTE_STRING token
    }
    else if (input[i] == ' ') {
        continue;
    }
    else if (inTagDef) {
```

```
        // ... generate TAG_NAME token
    }
    else {
        // ... generate HTML_TEXT token
    }
}
```

題目 - 總共兩小題(總共50分)

B. html lexer，能夠將輸入切成token。切出tokens後，依序輸出token的類型與值(20分)

輸入: 一行html文字
輸出: 一行一個token，印出token的型別後，空一格印token的值。最後會有一行空行。

不會有無法順利切出token的test case，ex:

```
<div class='>haha</div>
```

可以發現SINGLE_QUOTE_STRING只有開頭的單引號，卻沒有結束的單引號，會因為一直沒匹配到結束的單引號，而讀輸入到結束都無法匹配完SINGLE_QUOTE_STRING

example1:

input

```
<html> <div class='red'>text in div</div> </html>
```

ouput

> 注意SINGLE_QUOTE_STRING的值不包含單引號、DOUBLE_QUOTE_STRING的值不包含雙引號

```
TAG_OPEN <
TAG_NAME html
TAG_CLOSE >
TAG_OPEN <
TAG_NAME div
TAG_NAME class
TAG_EQUALS =
SINGLE_QUOTE_STRING red
TAG_CLOSE >
HTML_TEXT text in div
TAG_OPEN_SLASH </
TAG_NAME div
TAG_CLOSE >
TAG_OPEN_SLASH </
TAG_NAME html
TAG_CLOSE >
```

example2:

input

```
<div>"test"</div>
```

output

```
TAG_OPEN <
TAG_NAME div
```

```
TAG_CLOSE >
HTML_TEXT "test"
TAG_OPEN_SLASH </
TAG_NAME div
TAG_CLOSE >
```

由於匹配`"test"`的時候不在tag模式，因此是切出`HTML_TEXT`而非`DOUBLE_QUOTE_STRING`

C. 能判定給定的html文字是否符合題目中的文法定義。輸出parsing匹配完non-terminal symbol的過程，並且最後輸出給定的html是valid還是invalid(30分)

輸入: 一行html文字
輸出: 在parsing的過程中，若匹配成功non-terminal symbol就輸出其名稱並換行，最後結束時再印出valid 或 invalid，且最後會有一行空行

「在parsing的過程中，若匹配成功non-terminal symbol就輸出其名稱並換行」的做法就是在每個匹配non-terminal symbol的函數回傳前，若匹配成功就印出其non-terminal的名稱，匹配失敗就不印。ex:

```cpp
bool htmlElment(...) {
    bool match = true;
    // ...

    if (match) { std::cout << "htmlElement" << std::endl; }
    return match;
}
```

下面的eaxmples會列出所有html會valid或invalid的情境，所有測資都會按照這些情境出，不會刁難

example1:

一個基本的normal element
input

```
<div></div>
```

output

```
htmlElement
htmlDocument
valid
```

example2:

normal element的content有文字
input

```
<div>haha</div>
```

output

```
htmlCharData
htmlContent
htmlElement
htmlDocument
valid
```

example3:

normal element的content有文字和另一個tag，再來一個文字
input

```
<div>haha<span></span>haha2</div>
```

output

```
htmlCharData
htmlElement
htmlCharData
htmlContent
htmlContent
htmlContent
htmlElement
htmlDocument
valid
```

example4:

normal element有多個attribute
input

```
<div id='main' class='red'></div>
```

output

```
attribute
htmlAttribute
attribute
htmlAttribute
htmlAttributeList
htmlAttributeList
htmlElement
htmlDocument
valid
```

example5:

tag沒有tag的名稱
input

```
<div></>
```

output

```
invalid
```

example6:

normal element沒有close tag(因為是簡化版的html，normal element一定要有close tag。真正的html，存在void element，其不會有close tag)
input

```
<div>
```

output

```
invalid
```

example7:

normal element的attribute不完整
input

```
<div class=>
```

output

```
invalid
```

example8:

normal element沒有close tag(較複雜版，nested tag)
input

```
<div><span>text in span<div></span></div>
```

output

```
htmlCharData
htmlElement
htmlContent
htmlContent
htmlElement
htmlContent
invalid
```

由於parsing過程中沒有檢查open tag和close tag的tag名稱是否一致來做htmlElement的匹配，因此這個範例在匹配non-terminal symbol的過程看起來會有點詭異。下圖畫藍色底下的地方會被視為htmlElement