जटिल रेगुलर एक्सप्रेशन

हाल ही में मैं □□□□ पार्सिंग का अध्ययन कर रहा था और एक रेगुलर एक्सप्रेशन (□□□□□□□□ □□□□□□□□□□□) से मिला:

यह एक रेगुलर एक्सप्रेशन (□□□□□□) है जो □□□□ टैग्स, आईडी, क्लास, एट्रिब्यूट्स, और अन्य विशेषताओं को पार्स करने के लिए उपयोग किया जाता है। इसे हिंदी में समझाने का प्रयास करते हैं:

- `/([\w-:\*>]*)`: यह भाग टैग नाम को कैप्चर करता है। इसमें अक्षर, संख्याएं, हाइफन, कोलन, और कुछ विशेष वर्ण (जैसे `*` और `>`) शामिल हो सकते हैं।
- `(?:\#([\w-]+)|\.([\w-]+))?`: यह भाग आईडी (`#`) या क्लास (`.`) को कैप्चर करता है। यह वैकल्पिक है (इसलिए `?` का उपयोग किया गया है)।
- `(?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?`: यह भाग एट्रिब्यूट्स को कैप्चर करता है। इसमें एट्रिब्यूट नाम, ऑपरेटर (जैसे `=`, `!=`, `*=`), और मान शामिल हो सकते हैं। यह भी वैकल्पिक है।
- `([\/, ]+)`: यह भाग टैग के बाद आने वाले स्लैश (`/`) या कॉमा (`,`) और स्पेस को कैप्चर करता है।
- `/is`: यह फ्लैग्स हैं जो रेगुलर एक्सप्रेशन को केस-इनसेंसिटिव (`i`) और सिंगल लाइन मोड (`s`) में चलाने के लिए उपयोग किए जाते हैं।

यह रेगुलर एक्सप्रेशन □□□□ टैग्स और उनके एट्रिब्यूट्स को पार्स करने के लिए उपयोगी है, जैसे कि `<div id="main" class="container" data-attr="value">`।

यह □□□ चयनकर्ताओं (□□□□□□□□□□) से मेल खाने के लिए उपयोग किया जाता है, जैसे कि `div > ul`।

पहले मैंने ऐसे कई जटिल एक्सप्रेशन देखे हैं, और मैं हमेशा उनसे दूर भागता था। आज मैंने तय किया है कि इसे पूरी तरह से समझ लूंगा! आदमी को खुद पर थोड़ा सख्त होना चाहिए!

`div > ul` का मिलान करें

यह □□□ चयनकर्ता (□□□□□□□□□) उन `<ul>` तत्वों का चयन करता है जो किसी `<div>` तत्व के सीधे बच्चे (□□□□□□□ □□□□□) होते हैं। यहां एक उदाहरण दिया गया है:

```
<div>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

इस उदाहरण में, <ul> तत्व का सीधा बच्चा है, इसलिए यह div > ul चयनकर्ता से मेल खाएगा।

यदि तत्व किसी अन्य तत्व के अंदर नेस्टेड (□□□□□□□) होता है, तो यह मेल नहीं खाएगा। उदाहरण के लिए:

```
<div>
  <section>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
    </ul>
  </section>
</div>
```

इस मामले में, <ul> तत्व का सीधा बच्चा नहीं है, इसलिए यह div > ul चयनकर्ता से मेल नहीं खाएगा।

मैंने एक वेबसाइट □□□□□://□□□□□101.□□□/ ढूंढी है, जो ऑनलाइन मिलान करने और स्पष्टीकरण देने में सक्षम है।

हालांकि दाईं ओर के विवरण से कुछ स्पष्टता मिली है, लेकिन फिर भी यह स्पष्ट नहीं है कि वास्तव में मिलान कैसे किया जाता है। तो चलिए कुछ उदाहरण लेते हैं और एक-एक करके उनका विश्लेषण करते हैं।

इस रेगुलर एक्सप्रेशन का उपयोग करने वाला कोड निम्नलिखित है:

```
$matches = [];
preg_match_all($this->pattern, trim($selector).' ', $matches, PREG_SET_ORDER);
```

(यह कोड ब्लॉक □□□ में है और इसे अनुवादित नहीं किया जाना चाहिए।)

preg_match_all का मतलब है कि यह सभी स्ट्रिंग्स को प्राप्त करता है जो दिए गए पैटर्न से मेल खाते हैं। अगर हमारे पास यह है:

```
preg_match_all("abc", "abcdabc", $matches)
```

यह □□□ कोड preg_match_all फ़ंक्शन का उपयोग करता है। यह फ़ंक्शन एक स्ट्रिंग में एक पैटर्न के सभी मिलान ढूंढता है और उन्हें $matches नामक वेरिएबल में स्टोर करता है। इस उदाहरण में, पैटर्न "abc" है और स्ट्रिंग "abcdabc" है। इसलिए, यह "abc" पैटर्न को स्ट्रिंग में दो बार ढूंढेगा और $matches में स्टोर करेगा।

पहला पैरामीटर पैटर्न है, दूसरा पैरामीटर मिलान करने के लिए स्ट्रिंग है, और तीसरा पैरामीटर परिणाम संदर्भ है। चलाने के बाद, $matches सरणी में दो abc शामिल होंगे।

इस समझ के साथ, ऊपर दिए गए चित्र में div > ul केवल पहले चार वर्णों div > से मेल खाता है। क्या regex101 preg_match_all का समर्थन नहीं करता है? कोई बात नहीं, बस g नामक एक संशोधक जोड़ें और यह काम कर जाएगा:

g जोड़ने के बाद, यह सभी से मेल खाएगा, न कि केवल पहले से मिलने पर वापस आएगा।

जोड़ने के बाद, हमने `div > ul` से मिलान किया:

दाईं ओर दिखाया गया है, पहले मिलान में, यानी `div`, हमने पहले समूह के नियमों का उपयोग करके `div` से मिलान किया, और फिर सातवें समूह के नियमों का उपयोग करके स्पेस से मिलान किया।

आइए पहले नियम समूह की व्याख्या पर नजर डालते हैं:

इस लंबी अभिव्यक्ति में, पहले कोष्ठक में बंद हिस्से को पहला समूह नियम कहा जाता है। यह एक कैप्चर समूह है। कोष्ठक स्वयं मेल नहीं खाते, बल्कि समूह बनाने के लिए उपयोग किए जाते हैं। `[]` एक वर्ण समूह को दर्शाता है, और इसके अंदर के नियम बताते हैं कि यह किस प्रकार का वर्ण समूह है। इस वर्ण समूह में शामिल हैं:

- `\w` बड़े और छोटे अक्षरों, 0 से 9 तक की संख्याओं और अंडरस्कोर (_) को दर्शाता है।
- `-:` सीधे इन दोनों वर्णों को समूह में दर्शाता है।
- `\*` क्योंकि * रेगुलर एक्सप्रेशन में एक आरक्षित वर्ण है और इसका विशेष अर्थ होता है, इसलिए इसे एक साधारण * वर्ण के रूप में दर्शाने के लिए \ का उपयोग करके एस्केप किया जाता है।
- > सीधे > वर्ण को दर्शाता है।

`[\w-:\*>]*` में अंतिम * यह दर्शाता है कि पिछले वर्ण 0 या अनेक बार आ सकते हैं, लेकिन जितना संभव हो उतनी बार मेल खाने का प्रयास किया जाएगा। यह `div` से मेल खाता है क्योंकि `\w` ने `d`, `i`, और `v` से मेल खाया है। यह आगे के स्पेस से मेल नहीं खाता क्योंकि स्पेस `[]` में शामिल नहीं है। कैप्चर ग्रुप का अर्थ है कि यह मेल परिणाम सरणी में दिखाई देगा। इसके विपरीत नॉन-कैप्चर ग्रुप भी होते हैं, जिनका सिंटैक्स `(?:)` है। उपरोक्त `([\w-:\*>]*)` यदि इस परिणाम की आवश्यकता नहीं है, तो इसे `(?:[\w-:\*>]*)` के रूप में लिखा जा सकता है।

तो परिणाम में नहीं दिखने के लिए, सीधे कोष्ठक का उपयोग न करना क्यों नहीं? कोष्ठक समूहीकरण के लिए होते हैं, और समूहीकरण का बहुत महत्व होता है। आप □□□□□ □□ □ □□□ □□□□□□□□□□ □□□□□? (?:) - □□□□□□□□□□□□□□□ को संदर्भित कर सकते हैं।

`div` के पहले सेट के नियमों को पूरा करने के बाद, अब हम यह समझाएंगे कि स्पेस क्यों सातवें सेट के नियमों को पूरा करता है।

`[\/, ]` का मतलब है कि यह इन चार वर्णों में से किसी एक से मेल खाता है, और + का मतलब है कि पिछला मिलान एक या अनेक बार होता है, और जितना संभव हो उतनी बार होता है। इसलिए, क्योंकि इन चार वर्णों में स्पेस भी शामिल है, यह हमारे स्पेस से मेल खाता है। और क्योंकि `div` के बाद अगला वर्ण > है, इसलिए यह सातवें समूह के नियम को पूरा नहीं करता है, और आगे मेल नहीं खाता है।

`div` का मिलान समझ में आ गया है। तो फिर दूसरे से छठे समूह के नियमों ने यहां के रिक्त स्थान का मिलान क्यों नहीं किया, बल्कि उन्हें सातवें समूह के लिए छोड़ दिया?

दूसरे भाग की व्याख्या:

सबसे पहले, `(?:)` यह दर्शाता है कि यह एक नॉन-कैप्चरिंग ग्रुप है। अंत में आया `?` यह दर्शाता है कि इससे पहले का मिलान 0 या 1 बार हो सकता है। इसलिए ऊपर दिया गया `(?:\#([\w-]+)|\.([\w-]+))?` हो सकता है या नहीं भी हो सकता है। बाहरी मॉडिफायर को हटाने के बाद, जो बचता है वह है `\#([\w-]+)|\.([\w-]+)`, जहां बीच का | "या" को दर्शाता है, यानी इनमें से कोई एक मिलान होना चाहिए। `\#([\w-]+)` में `\#` # कैरेक्टर से मिलान करता है, और `[\w-]+` अन्य कैरेक्टर्स से मिलान करता है। फिर, दूसरे हिस्से में, `\.([\w-]+)` में `.` . कैरेक्टर से मिलान करता है।

इसलिए 2 से 6 समूह संभवतः रिक्त स्थान के कारण संतुष्ट नहीं हो सकते हैं क्योंकि ये समूह रिक्त स्थान को अपने शुरुआती वर्ण के रूप में नहीं मांगते हैं। चूंकि इन समूहों में एक `?` संशोधक है, इसलिए यदि वे संतुष्ट नहीं होते हैं तो भी कोई समस्या नहीं है, और इसलिए यह सातवें समूह पर कूद जाता है।

`div > ul` के बाद आने वाला > भी वैसा ही है:

पहला नियम `([\w-:\*>]*)` ने > को मैच किया, और सातवां नियम `([\/, ]+)` ने स्पेस को मैच किया। फिर `ul` भी `div` की तरह काम करता है।

मिलान करें `#answer-4185009 > table > tbody > td.answercell > div > pre`

यह एक □□□ सेलेक्टर है जो □□□□ डॉक्यूमेंट में एक विशिष्ट एलिमेंट को टार्गेट करता है। यह सेलेक्टर निम्नलिखित तरीके से काम करता है:

1. `#answer-4185009` - □□ `answer-4185009` वाले एलिमेंट को चुनता है।
2. `> table` - उस एलिमेंट के सीधे बच्चे (□□□□□□ □□□□□) के रूप में `table` एलिमेंट को चुनता है।
3. `> tbody` - उस `table` एलिमेंट के सीधे बच्चे के रूप में `tbody` एलिमेंट को चुनता है।
4. `> td.answercell` - उस `tbody` एलिमेंट के सीधे बच्चे के रूप में `td` एलिमेंट को चुनता है जिसकी क्लास `answercell` है।
5. `> div` - उस `td` एलिमेंट के सीधे बच्चे के रूप में `div` एलिमेंट को चुनता है।
6. `> pre` - उस `div` एलिमेंट के सीधे बच्चे के रूप में `pre` एलिमेंट को चुनता है।

यह सेलेक्टर □□□□ डॉक्यूमेंट में एक विशिष्ट `pre` एलिमेंट को टार्गेट करता है जो उपरोक्त पदानुक्रम में स्थित होता है।

अब एक थोड़ा और जटिल चयनकर्ता (□□□□□□□□□) आता है `#answer-4185009 > table > tbody > td.answercell > div > pre` (आप इसे □□□□□://□□□□□101.□□□/ पर खोलकर वहाँ पेस्ट करके टेस्ट भी कर सकते हैं):

यह □□□□□□ से कॉपी-पेस्ट किया गया है:

पहला मिलान:

क्योंकि पहले समूह के नियम `([\w-:\*>]*)` में `[]` के अंदर के कैरेक्टर सेट में कोई भी `#` से मेल नहीं खाता, और फिर अंत में `*` 0 या अनेक बार मेल खाने का समर्थन करता है, यहां यह 0 बार है। फिर दूसरे समूह के नियम का विवरण है:

ऊपर पहले ही विश्लेषण किया जा चुका है। सीधे | से पहले के `\#([\w-]+)` को देखें, `\#` ने `#` को मैच कर दिया है, और `[\w-]+` ने `answer-4185009` को मैच कर दिया है। इसके बाद वाला `\.([\w-]+)`, अगर `.answer-4185009` होता तो यह मैच लागू होता।

अगला, `td.answercell` इस मिलान को देखते हैं,

पहले समूह का नियम `([\w-:\*>]*)` ने `td` से मेल खाया, और दूसरे बड़े हिस्से `(?:\#([\w-]+)|\.([\w-]+))?` के बाद वाले हिस्से, यानी `\.([\w-]+)`, ने `.answercell` से मेल खाया।

इस चयनकर्ता (□□□□□□□□□) का विश्लेषण यहीं समाप्त होता है।

a[href="http://google.com/"] **से मिलान करें**

यह □□□ सेलेक्टर उन <a> टैग्स को चुनता है जिनका href एट्रिब्यूट http://google.com/ के बराबर होता है। यह सेलेक्टर बहुत सटीक है और केवल उन्हीं लिंक्स को मैच करेगा जिनका href एट्रिब्यूट बिल्कुल http://google.com/ होगा।

उदाहरण:

```
<a href="http://google.com/">Google</a> <!--         -->
<a href="https://google.com/">Google</a> <!--            -->
<a href="http://google.com">Google</a> <!--           (    '/'    ) -->
```

इस सेलेक्टर का उपयोग करके आप सटीक रूप से उन लिंक्स को स्टाइल या मैनिपुलेट कर सकते हैं जो सीधे http://google.com/ की ओर इशारा करते हैं।

अगले चरण में हम सेलेक्टर a[href="http://google.com/"] को मैच करेंगे:

तीसरे बड़े ब्लॉक को देखें:

तीसरे ब्लॉक का एक्सप्रेशन है (?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?। सबसे पहले, बाहरी (?:) इंगित करता है कि यह एक नॉन-कैप्चरिंग ग्रुप है, और अंत में ? इंगित करता है कि यह पूरा बड़ा ब्लॉक 0 या 1 बार मैच कर सकता है। इसे हटाने के बाद यह \[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\] बन जाता है। \[[कैरेक्टर को मैच करता है। @? इंगित करता है कि @ कैरेक्टर वैकल्पिक है। अगला ग्रुप (!?[\w-:]+) है, जहां ! वैकल्पिक है और [\w-:]+ href को मैच करता है। इसके बाद का ग्रुप (?:([!*^$]?=)["']?(.*?)["']?) एक नॉन-कैप्चरिंग ग्रुप है, जिसे हटाने के बाद यह ([!*^$]?=)["']?(.*?)["']? बन जाता है। यहां ([!*^$]?=) में [!*^$]? इंगित करता है कि [] के अंदर के कैरेक्टर्स में से 0 या 1 को मैच किया जाए। फिर = सीधे मैच होता है। इसके बाद ["']?(.*?)["']? "http://google.com/" को मैच करता है, जहां ["']? इंगित करता है कि " या ' या दोनों में से कोई भी नहीं मैच हो सकता है। इस बाहरी हिस्से को हटाने के बाद (.*?) http://google.com/ को मैच करता है, जहां *? इंगित करता है कि जितना संभव हो उतना कम मैच किया जाए। इसका मतलब है कि अगर " या ' है, तो उसे बाद के एक्सप्रेशन ["']? को मैच करने के लिए छोड़ दिया जाए। इसलिए यह http://google.com/" को नहीं, बल्कि केवल http://google.com/ को मैच करता है। इस प्रकार, पूरा सेलेक्टर a[href="http://google.com/ हो जाता है।

"] □□□□□□□□

सारांश

आखिरकार समझ में आ गया! चलिए एक बार फिर से स्पष्ट करते हैं, पहले पूरे जटिल एक्सप्रेशन ([\w-:*>]*)(?:\#([\w-]+)|\.([\w-]+))?(?:\
]+) को चार मुख्य भागों में विभाजित करते हैं:

- □ ([\w-:*>]*)
- □ (?:\#([\w-]+)|\.([\w-]+))?

- (?:\[@?(!?[\w-:]+)(?:([!*^$]?=)["']?(.*?)["']?)?\])?
- ([\/,]+)

ये रेगुलर एक्सप्रेशन (□□□□□□□ □□□□□□□□□□□) हैं जिनका उपयोग विभिन्न पैटर्न को मैच करने के लिए किया जाता है। इन्हें हिंदी में अनुवाद करने की आवश्यकता नहीं है क्योंकि ये कोड के हिस्से हैं और इन्हें उनके मूल रूप में ही रखा जाना चाहिए।

सबसे जटिल तीसरा भाग इन कुछ हिस्सों से मिलकर बना है:

- \[
- (!?[\w-:]+)
- (?:([!*^$]?=)["']?(.*?)["']?)?
- \]

यह एक □□□□□ (रेगुलर एक्सप्रेशन) है जो किसी विशेष पैटर्न को मैच करने के लिए उपयोग किया जाता है। इसे हिंदी में समझाने का प्रयास करते हैं:

1. `\[` - यह एक स्क्वायर ब्रैकेट `[` को मैच करता है।
2. `(!?[\w-:]+)` - यह एक ग्रुप है जो एक या अधिक वर्ड कैरेक्टर्स (अक्षर, संख्या, अंडरस्कोर), हाइफन, या कोलन को मैच करता है। इससे पहले एक ऑप्शनल `!` हो सकता है।
3. `(?:([!*^$]?=)["']?(.*?)["']?)?` - यह एक नॉन-कैप्चरिंग ग्रुप है जो ऑप्शनल है। यह एक ऑपरेटर (=, !=, *=, ^=, $=) को मैच कर सकता है, और उसके बाद एक ऑप्शनल कोटेड स्ट्रिंग (सिंगल या डबल कोट्स में) को मैच करता है।
4. `\]` - यह एक स्क्वायर ब्रैकेट `]` को मैच करता है।

इस □□□□□ का उपयोग अक्सर □□□□ या □□□ जैसी भाषाओं में एट्रिब्यूट्स को पार्स करने के लिए किया जाता है।

इसलिए ये सभी छोटे-छोटे हिस्से एक-एक करके समझे जा सकते हैं। फिर और उदाहरण ढूंढें, देखें कि प्रत्येक उदाहरण कैसे मेल खाता है, और साथ ही □□□□□://□□□□□101.□□□/ की व्याख्या का उपयोग करके विश्लेषण करें। इस तरह, यह जटिल लगने वाला रेगुलर एक्सप्रेशन समझ में आ जाएगा, और पता चलेगा कि यह एक कागजी शेर है!