HAWK (HTML is All We Know) Language Proposal

Jonathan Adelson, Ethan Benjamin, Justin Chang,Graham Gobieski, George Yu
jma2215, jc4137, eb2947, gsg2120, gy2206

Introduction

HAWK (HTML is All We Know) is a play on AWK, and strives to accomplish for HTML web scraping what AWK accomplished for text processing. Web scraping describes the process of automatically extracting data from websites. For instance, one could write a web scraping program to look at the menu for John Jay dining hall each day and determine if bacon is being served. As another example, one could scrape IMDB to determine how many degrees of separation a given actor has from Kevin Bacon.

Though no two web scraping tasks are the same, most web scraping programs employ a similar workflow. For the most part, this involves finding relevant parts of a web page and performing some action in response. In practice, the most typical "relevant parts" are HTML elements that match some criteria or certain strings in the raw HTML document. Usually, in order to find these parts you must combine various search mechanisms including XPath, CSS selector search, and regexes. Often, these distinct search mechanisms are implemented in separate libraries which each have distinct abstractions that don't play well together.

HAWK unifies the disparate aspects of web scraping in a clean and coherent manner. Like AWK, HAWK mostly consists of pattern-action pairs. HAWK supports multiple types of patterns, and treats each as a first class citizen. For our project, we will support three types of search patterns for HTML documents: CSS selectors, regular expressions, and HAWK predicates. CSS selectors and HAWK predicates will allow users to match whole HTML elements. HAWK predicates are simply boolean expressions with the same syntax

and semantics as HAWK's action language. Regular expressions will let users cut right to the chase and scrape raw string data.

HAWK's action syntax and semantics will be bare-bones and dynamic, in the spirit of Lua. We hope to provide just enough features to make the large majority of tasks straightforward, and enough flexibility to make hard tasks possible. Like Lua, we will provide only one built-in data structure, a table, which is essentially just a key-value hash table. Just as in AWK, we will provide several built-in variables (for both patterns and actions) which will assist the programmer in performing common operations.

Code Examples

Below are some hypothetical HAWK programs. Each performs a web scraping task on the Wikipedia site listing the tallest mountains in the United States:

https://en.wikipedia.org/wiki/List_of_mountain_peaks_of_the_United_States

Example 1: Print Top 50 Mountains From Colorado

```
/*Beginning of program. This is a comment.
Like in AWK, code done at the beginning is marked with BEGIN.
No action taken in this case.*/

BEGIN{}

/*No beginning actions.
Since its empty, the {} is optional here and could be ommited.*/

{}

/*table.wikitable > tr is a CSS selector and
matches all <tr> elements within the mountain table.
With the {...}  is the action to perform when finding a pattern.
In this case, no action is taken when encountering the table.
Within the [...] are subpatterns to look for
```

```
within the table element, table.wikitable > tr*/

table.wikitable > tr
[
        /*First get the height ranking of the mountain
        corresponding to the first column.
        $1 is special pattern, meaning first element of current parent match
        (the first <td> of the parent <tr> in this case).*/

        /*Assign the $rank variable to the content of the <td>
        All variables have global scope*/
        $1 { $rank = str2num($content);}

        /*Next get the state of the mountain
        $3 = 3rd columnof tr = state <td>
        $elt is special variable corresponding to the matched element.
        It is an object that has methods similar to jQuery elements.
        Within 3rd column of tr, get the second child's title attribute.*/

        $3 { $state = $elt.nth_element(2).attr("title");}

        //Fetch the name of the mountain
        $2 { $mountain = $elt.nth_element(1).attr("title");}
]
//Ending action for <tr>.
//Print name of mountain if it is ranked highly enough and in Colorado
END{
        if ($rank <= 50 and $state == "Colorado")
                print($mountain)
        end
}
```

Example 2: Show Tallest Mountains By State

```
BEGIN{
        $counts_by_state = {}
}
```

```
table.wikitable > tr
[
        $3{
                $state = $elt.nth_element(2).attr("title");
                $counts_by_state[$state] = ($counts_by_state[$state] or 0) + 1
        }
]

END{
        for state in counts_by_state do
                print("STATE: " + $state + ", COUNT: " + $counts_by_state[$state]);
        end
}
```

Example 3: List mountains between 14,000 and 15,000 feet

```
table.wikitable > tr
[
        //Fetch the name of the mountain
        $2 { $mountain = $elt.nth_element(1).attr("title");}

        /*Get the height of the mountain using a regular expression pattern!
        $MATCH is a special array variable corresponding to the captured
        groups in the the regex. In this case it captures the feet*/

        /(\d+,\d+) ft/ { $height = str2num($MATCH[1]);}
]

//Ending action for <tr>.
//Print name of mountain if it is in the right height range
if height>=14000 and height<=15000
{
        print($mountain);
}
```