Hap-Shu
A Language for Locating Information in HTML Documents

by
Baris Temelkuran

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
May 20, 2003

© Copyright 2003 Massachusetts Institute of Technology. All rights reserved.
Hap-Shu
A Language for Locating Information in HTML Documents

by
Baris Temelkuran

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The World Wide Web (WWW) has evolved into an enormous knowledge base that has attracted researchers into exploring ways of accessing and indexing it. The goal of the START system is to let users access this knowledge using common written English. The amount of knowledge and the variety of its representation in WWW makes this goal hard to achieve. Hap-Shu is a language that unifies the representation of knowledge for START, abstracting away from HTML-specific vocabulary. Hap-Shu also gives more information about the failure of this abstraction when it fails to extract the knowledge due to various externalities such as major format changes. This information can later be used to automatically fix the scripts written in Hap-Shu. A system that includes an automatic repair system and an editor is also introduced.

Thesis Supervisor: Boris Katz
Title: Principal Research Scientist, MIT Artificial Intelligence Laboratory
# Table of Contents

1. Introduction ........................................................................................................ 5
2. Problem Definition .............................................................................................. 7
3. Literature Review ............................................................................................... 10
4. Hap-Shu ............................................................................................................. 14
   4.1. Introduction ............................................................................................... 14
   4.2. Command List .......................................................................................... 16
   4.2.1. table-with ......................................................................................... 16
   4.2.2. item-with ........................................................................................... 17
   4.2.3. picture-with ....................................................................................... 18
   4.2.4. nth-item ............................................................................................ 18
   4.2.5. all-till-next-title .............................................................................. 19
   4.3. Other Commands & Features .................................................................... 22
   4.3.1. get-all-results .................................................................................. 22
   4.3.2. Chaining ............................................................................................ 22
   4.3.3. The HTML Parser ............................................................................ 23
5. The Editor ........................................................................................................... 26
6. The Repair System ............................................................................................. 31
   6.1. Separation of the Content from the Format .............................................. 31
   6.2. The Design ............................................................................................... 35
   6.2.1. Cached Results Database .................................................................... 36
   6.2.2. Omnibase .......................................................................................... 37
   6.2.3. Users .................................................................................................... 38
   6.2.4. Command Mutators .......................................................................... 39
   6.2.5. Repair Results Handler ...................................................................... 40
   6.2.6. Main Repair System .......................................................................... 41
7. Future Work ......................................................................................................... 47
8. Conclusion ........................................................................................................... 48
9. References ........................................................................................................... 49
Table of Figures

Figure 1 : Snapshot from the Factbook page

Figure 2 : Snapshot from the IMDb page
http://us.imdb.com/Name?Tarantino,+Quentin, on April 24th. 2003

Figure 3 : Steps of a sample run for all-till-next-title command

Figure 4 : A snapshot of the editor after loading

Figure 5 : A snapshot of the editor after loading a URL for script creation

Figure 6 : A snapshot of the editor after submitting some keywords for command generation

Figure 7 : A snapshot of Frame #5 opened in a new window to see testing results

Figure 8 : The Diagram of the Six Major Parts of the Repair System
1. Introduction

Many systems being developed today are in need of unified access to the information residing in the World Wide Web (WWW). One such system, START (SynTactic Analysis using Reversible Transformations)\(^1\), is a natural language question answering system that uses the WWW as its knowledge base [1-3]. The unstructured representation and dynamic nature of the knowledge in the WWW renders the task of retrieving the sought-after knowledge from a web page difficult. The traditional ways to extract information from HTML documents, which form the biggest portion of the knowledge on the web, involve manually written wrappers containing low-level regular expressions\(^2\) [4] or high-level tag paths [5] and automatically or semi-automatically generated wrappers [6]. These wrappers are difficult to build and costly to maintain. They are often vulnerable to minor format changes such as capitalization of tags or addition of spaces in the HTML document and/or to major changes such as introduction of tables to the document. Occasionally, web sites change and/or enhance their designs for HTML documents, causing such wrappers to fail.

Currently, START uses a system called Omnibase [7] as its uniform access interface to the web. The biggest portion of Omnibase is a database of scripts that extract information from various web sites. These scripts are written in Guile\(^3\) and include many regular expressions to extract the information. As Omnibase expands to encapsulate more information, the increasing number of scripts with high complexity and of web sites Omnibase has access to causes more frequent failures of the system.

One way to solve this problem is to store a copy of the remote data or the parsed information from the remote data locally. This approach has three major problems. The first and the biggest problem is scalability. Considering the size of the web, storing the knowledge locally is far from being practical. The second problem is that web pages are often dynamic. Locally storing weather reports will likely result in wrong answers. A third problem is that the scripts dealing with locally cached data are no easier to write than those to parse fresh data.

\(^1\) http://www.ai.mit.edu/projects/infolab/
\(^2\) Refer to http://www.perldoc.com/perl5.6/pod/perlre.html for information about regular expressions
\(^3\) http://www.gnu.org/software/guile/guile.html
Hap-Shu⁴ is a language designed to define the location of the knowledge in an HTML document in a more abstract way than regular expressions. The main goal of the language is to ease the creation and maintenance of the scripts. An important property of Hap-Shu is the fact that the scripts written in Hap-Shu will be automatically repairable to some extent. The scripts written in this language will be able to define not only the location but also the type of the knowledge. By combining the location and type information of the knowledge, the system's built-in support of the language has a good chance of repairing the broken script.

⁴ Hap-Shu is the sneezing sound written in Turkish, and should be read as Hop-Shoe.
2. Problem Definition

The World Wide Web (WWW) is dynamic not only in its knowledge domain but also in its format domain. Thus, one of the problems in trying to access the dynamic knowledge is its dynamic shape on the web. For example, The CIA World Factbook changes its content and page layout every year. START is a system designed to answer queries that include queries about attributes of real world entities such as the age of a celebrity or the area of a country. Omnibase helps START by storing the entity names, called symbols, as well as what attributes each entity class has and scripts for each attribute that extract the information about that attribute from the WWW given a symbol. The scripts make extensive use of regular expressions.

A script that uses regular expressions to extract, for example, the area information of a given country from the Factbook generally fails to do so after the annual layout changes. This may be due to a stress change in the title area such as the substitution of B tags with STRONG tags, introduction of additional HTML elements like tables, or even as small a change as the addition of spaces to the document. The script then has to be rewritten/repaired manually as it does not contain any semantic information implied by both HTML tags and the sought-after knowledge. One can argue that the script writer can write more flexible scripts to handle these problems. This way of generating scripts has two problems, which are the replication of code and the human inclination for writing short scripts. Trying to write flexible scripts with regular expressions for every extraction task will introduce code replications. Also, the inclination to write shorter scripts that just get the job done and the misperceived chance of failure of one script will lure script writers to write short scripts with very little generalization.

One of the causes for the high cost of maintenance is the non-trivial process of using regular expressions to parse HTML fragments. The usage of these expressions is also a problem in the creation of such scripts. For example, one can use "<table[^]>+><tr>Area: " and "</table>" regular expressions to find the start and the end of the table below from an HTML fragment

<table width=600><tr>Area:</tr><tr>100 sq km</tr></table>

---

http://www.cia.gov/cia/publications/factbook/
One other example, the script in Omnibase that extracts the area information given a country name using The CIA World Factbook is given below:

```lisp
(lambda (symbol)
  (let* ((page (get "cia-country" symbol "PAGE"))
         (ml (match "$<td width="20%" valign="top"...
              ...class="FieldLabel">[^<]"*<div align="right">Area"
              page))
         (sl (make-shared-substring page
                         (match:start ml))))
    (string-append "<BASE 
" href="http://www.odci.gov/cia/publications/factbook/" 
"geos/">" 
"<table width="100%" cellspacing="0">" 
<tr> 
<td width="20%" valign="top" class="FieldLabel">Area:</td> 
<td valign="top" bgcolor="#FFFFFF" width="80%"> 
<a href="../../docs/notesanddefs.html#2147"> 
<img src="../../graphics/dictionary.jpg" alt="Definition"> 
</a> 
<a href="../fields/2147.html"> 
<img src="../../graphics/listing.jpg" alt="Field Listing"> 
</a> 
<br> 
    <i>total:</i> 780,580 sq km 
    <br><i>water:</i> 9,820 sq km 
    <br><i>land:</i> 770,760 sq km 
      </td> 
</tr></table>

The script will extract the following HTML fragment if given "Turkey" as its input.

```HTML

```

There are three problems with this approach. The script writer has to examine the HTML code carefully, the script will break with small alterations to the HTML fragment such as the addition of B or I tags around “Area:” for more stress, and the level of readability is greatly reduced due to complexity introduced by regular expressions.
Hap-Shu’s editing and automatic repair systems have achieved the following five main goals:

- The script creation process was made easier and more intuitive
- The scripts are more resistant to format and layout changes, reducing the frequency of repairs
- The scripts execute quickly
- The automatic repairing system can often repair the broken scripts
- The automatic repairing system generates detailed easy-to-understand error reports even if the automatic repair fails
3. Literature Review

Researchers have tried many ways to approach the problem of defining the location of information in the semi-structured HTML documents. In this section, some of these approaches will be discussed.

It is appropriate to reiterate the purposes of this thesis here before discussing any approach in detail:

1. Easier creation of wrappers
2. Wrappers that are resistant to format changes
3. Fast execution of wrappers
4. Useful feedback on failures
5. Automatically repairable wrappers

In the following discussions, a proper set of HTML documents is defined as a set of HTML documents that is expected to differ only by textual content visible to the human eye through browser but not by tag structure (layout). One example would be the set of HTML documents retrieved from Factbook using different country names.

Grieser et al. created a system called LEXIKON [8] that uses learning techniques in a feedback loop that involves a human user. The user is asked to point out the important sections of an HTML document chosen from a set of similar documents. The system uses the response to create a wrapper that will extract some HTML fragments which are shown to the user for modification. The user can add or remove parts from the HTML fragments the wrapper created or choose to try the wrapper on another document. The system uses the additional information supplied by the user to modify the wrapper and once again presents the results to the user continuing the feedback loop. The loop is broken by the user when the results are satisfactory. The created wrapper is intentionally hidden from the user with the aim of keeping the user in a simple feedback loop.

The system achieves the first goal of easy wrapper creation by giving the user a very simple intuitive interface. On the other hand, since the user does not have any insight to the wrapper created, the wrapper is likely to be vulnerable to future layout changes. The system is designed to produce wrappers that will work at their time of creation. The system is very convenient for information extraction from a set of documents that contain static information, or information
needed at a specific point in time. Using this system, for example, for Factbook will require the user to recreate wrappers every year.\(^6\)

Taniguchi et al. defined "association paths" \(^[5]\) using a pair of patterns, a tag pattern and a word pattern. A tag pattern is like a path of tags, between any two of which any \(k\) other tags can exist to form an actual tag path from the root of an HTML document. The word pattern is analogous to the tag pattern with words replacing tags in the pattern. One such association path formatted as \texttt{tag pattern \# word pattern} is:

\[
\langle i \text{ font p body html} \rangle \# \langle tsp \rangle
\]

In the paper, the pattern above is used in the sense that any HTML fragment that matches this pattern is considered important for data mining purposes. The word ‘tsp’ is the abbreviation for traveling salesman problem, which occurred frequently in their HTML document set.

Using such an association path for the purpose of this thesis would bring about three problems. The first of these problems is the necessity for the user to look at the HTML source to find an appropriate path. In a proper set of HTML documents, using the real path from the root, the \texttt{html} tag, to the fragment we need, will give us a wrapper that will work for the time being. Any layout change in the document would break this path, thus failing our second goal. The second problem is the repairability as the level of importance of each tag on the association path is missing in the association path. Trying to fix such a path where the shape (table?, list?, picture?) of the information sought after is not properly known is difficult. The third problem is the speed of execution, where parsing of the whole page may be necessary to find the matching HTML fragment. Besides, parsing the whole page may make the wrapper even more fragile to formatting errors in the HTML.

The same arguments for the previously discussed paper can be made for Hammer et al. \(^[4]\). The authors have developed a tool that extracts information from a web page and turns it into a database object, given a tag pattern for each required field. An example pattern is given below\(^8\):

\[
[\{ "root" ,
  "get(\"http://www.intellicast.com/weather/europe/\")\}]
\]

---

\(^6\) As stated before, the Factbook tends to change its layout every year.

\(^7\) \(k\) is the proximity parameter in the paper. Actually \(k-1\) other tags can exist. This is analogous to \(<[^]*[^>]*[^>]\rangle(0,k-1)\) between each tag in a regular expression.

\(^8\) The example is taken from the paper \([4]\).
This pattern is far more complicated than the tag pattern described by Taniguchi et al. This pattern structure is very powerful, but also is difficult to write. Automatically repairing a broken pattern is much more difficult than rewriting it manually since the patterns are long and complicated for an automatic system to repair. The purpose of this tool is to extract the data from a given web page, rather than to generate a reusable wrapper for a proper set of HTML documents. For that reason, the authors put more effort into a powerful information extractor than a very intuitive one that is also resistant to formatting changes.

Ashish and Knoblock presented a semi-automated wrapper generator [6] with the same purpose of use as Hap-Shu. Their wrapper is used to query attributes of a symbol using the WWW. They also use the Factbook as their example to query ‘area’ and other attributes of a country. Their tool generates a content structure of a given HTML page from a proper set of HTML documents using a heuristic described in the paper. The structure is then presented to the user for final editing. The difference between this tool and LExIKON is that the user modifies the content structure, which is designed to be intuitive, instead of modifying the output. Thus the user has more control on how to access a needed HTML fragment. According to the evaluation results, generating the correct content structure for a given HTML document takes only a few minutes. The only major drawback of this method is the speed of the parser for extraction. It is necessary to parse the HTML document to derive the content structure to extract any specific field, which may take too much time on larger documents, and is prone to formatting errors. Repair is also troublesome as any modification in the layout will most likely leave any old parser obsolete. However, given the short amount of time required to generate a
new structure and parser, one may choose to manually rebuild broken parsers and give up on an automatic repair facility.

Adelberg and Denny touched upon the robustness of wrappers [9]. They point out the importance of realizing that most templates are generated by humans, and thus are prone to errors. Most wrappers will fail on human errors such as unclosed tags or unopened ones in a document. The authors have built a system that reports errors on a statistical basis. For example, if a 200 character text is returned from a wrapper that generally returns a 10 character text, the system signals an error. Most of the systems discussed above are vulnerable to the following types of errors:

- The format of the wrapped source may change. Web sites are particularly guilty of this.

- Although a wrapper tests successfully on a set of example data, it is possible that it will fail in actual operation if it encounters a document with a previously unencountered feature, such as an infrequent field. Given that wrappers often need to be developed very quickly, particularly after a previously supported web site has changed formats, the likelihood of not using a fully representative set of pages to test increases.

- For sources where the documents are human generated or OCRed, typos or OCR errors can wreak havoc with the wrapper's parser.

---

9 In a statistical sense
10 Quoted verbatim from [9]
4. Hap-Shu

4.1. Introduction

Hap-Shu is a language designed to provide easier and more intuitive location definition for information in an HTML document. Let us now examine how a human would intuitively extract information for an attribute of a symbol from a web page.

Example 1:

If the user is looking for the attribute ‘Population’ of a symbol ‘Turkey’, he will go to the Factbook, find the page on Turkey and scroll down the page till he sees the word ‘Population’. He will reach the piece shown in Figure 1. Here he has to choose between two different rows, both of which seem to contain population information. These rows are labeled with ‘Population’ and ‘Population growth rate’. The user will look at the first row since he is looking for ‘Population’ and not ‘Population growth rate’. The smart heuristic approach to mimic human behavior is to pick the first row where the word ‘Population’ occupies more of the title. This process can be briefly described as finding an object with a title that contains ‘Population’ and as few other characters as possible.
Example 2:


Filmography as: Actor, Writer, Producer, Director, Miscellaneous Crew, Editor, Himself, Notable TV Guest Appearances

**Director - filmography**

1. Inglourious Basterds (2004) (announced)
5. Pulp Fiction (1994)
7. Reservoir Dogs (1992)

Filmography as: Actor, Writer, Producer, Director, Miscellaneous Crew, Editor, Himself, Notable TV Guest Appearances

**Miscellaneous Crew - filmography**

1. Doena (1999) (special thanks)
2. Seul contre tous (1998) (special thanks)
   ... aka I Stand Alone (1998) (USA)
   ... aka Dolph Lundgren: Maximum Potential (1987) (V) (USA)

Figure 2: Snapshot from the IMDb page
http://us.imdb.com/Name?Tarantino,+Quentin, on April 24th, 2003

The user now wants to find out which movies Quentin Tarantino directed, i.e., the 'Director - filmography' attribute of the symbol 'Tarantino, Quentin'. The user goes to imdb.com and fetches the page for Tarantino. He looks for the word 'Director' to reach the piece of HTML seen in Figure 2. Here, the user has one choice; there is only one 'Director - filmography' in the page. Now the question is when the information ends. Again, a smart user will notice that 'Miscellaneous Crew - filmography' is formatted the same way as 'Director - filmography'. So any information up to that point should be of his interest. Briefly, one can explain this process as: find a title with words 'Director - filmography' and get all the information till the next title of the same format. Of course, if the page ends or a table where the information resides ends, the user will cease to look for the next title.
The two examples above contain the two main ideas used in designing Hap-Shu:

1. Looking for some structure that contains the keywords
2. Looking for a title that contains the keywords, and getting all the information till the next title

Hap-Shu is designed for simplicity. One of the purposes of its design is to make the creation of wrappers easy. Hap-Shu was developed in Guile and could be thought of as a set of modules in Guile. The reasons Hap-Shu was not designed as a standalone language but as a set of modules are:

1. For any location Hap-Shu cannot express due to its simplicity, the user can use Hap-Shu partially to arrive at the whereabouts of the information and use his own routine in Guile to proceed.
2. Hap-Shu is designed to be part of a system, Omnibase, which was developed in Guile.
3. The scripts written with Hap-Shu will replace Guile scripts; thus for the ease of transition Hap-Shu is designed to be a module in Guile.

Run-time script generation, modification and evaluation power in Guile also make it a very good candidate for a repair system.

4.2. Command List

4.2.1. table-with

The command table-with takes three arguments: a list of case-insensitive keywords, a source HTML fragment, and a non-negative integer. The keywords cannot be HTML tags (refer to Section 4.3.2, Chaining, for more information). It returns one of the following HTML structures:

- Table : defined by the tag table
- List : defined by the tags ul, ol or dl
- Other : defined by the tags dir or menu

---

11 The word 'table' in this section does not only refer to the table tag unless specifically stated so. The word 'table' refers to any structure that resembles a table.
The structure returned has a titlefound in it. The titlefound has the following properties:

- It contains all the keywords in the keyword list.
- It preserves the order of the keywords.
- No text other than tags and whitespace characters exist between any consecutive two keywords. No tag can precede the first or follow the last keyword.
- None of the keywords is inside a tag declaration. The only exception to this is when all the keywords are in one image tag, in which case titlefound is the image tag.
- It is the longest string satisfying the rest of the conditions.

In a nested structure, the innermost table or list containing the titlefound is returned. If there is more than one such table, for example due to existence of multiple titlefound candidates (see Example 1), the third parameter determines the outcome.

If the parameter is 0, the title heuristic is used. The title heuristic gives each candidate a score, which is calculated using the following formula:

\[
Score = 100 \times \frac{\text{length(keywords)}}{\text{length(titlefound)}},
\]

The heuristic returns the table with the highest score. In case of a tie, the longer of the extracted fragments that tied will be returned. For Example 1, the first row would get 80.0 due to the extra ':' character in the titlefound whereas the second row would get 21.1.

If the user does not want to use the heuristic, he can reach other extracted tables by setting the third parameter to a positive integer less than or equal to the number of extracted tables. Note that these tables are still the innermost tables in their respective nested structures. If the third parameter is greater than that, an exception is thrown. Please refer to Section 4.3, Other Commands & Features, for information about accessing all the extracted tables at once.

### 4.2.2. item-with

This command is analogous to the table-with command with the exception that the structures returned are as follows:

- List item : Defined by the tag li
- Table row/cell : Defined by the tags tr/td
- Paragraph: Defined by the tag `p`

- Definition item: Defined by the tags `dt` and `dd`

`item-with` is useful for reaching finer HTML structures. Its parameters are the same as `table-with`.

### 4.2.3. picture-with

This command is analogous to the `table-with` command but returns an image defined by tag `img` that has the keywords in its `src` or `alt` attribute. The purpose of this command is to retrieve pictures. The parameters are the same with `table-with` command.

### 4.2.4. nth-item

This command has two arguments, an HTML fragment and a number `N`. It returns the `Nth` item in the HTML fragment. An item for this command is any structure defined by the following tags: `table`, `ol`, `ul`, `dl`, `dt`, `li`, `p`, `tr`, `td`. While finding the `Nth` item, the command only looks at the items on the top level, i.e., without entering any sub structure implied by tags like `ul`. For example, running `nth-item` on the fragment below with `N=2`:

```
<table ignore="This is the first item in this fragment">
  <tr ignore="nth-item will not enter this level">
    <td>This is a table cell</td>
  </tr>
</table>
<ol ignore="This is the second item">
  <li>List item 1</li>
  <li>List item 2</li>
</ol>
```

will return:

```
<ol ignore="This is the second item">
  <li>List item 1</li>
  <li>List item 2</li>
</ol>
```

In the example above, there are only two items, the table and the ordered list. If the user wants to pick an item inside the `table` tag, he has to call `nth-item` using the HTML fragment inside the table.
4.2.5. all-till-next-title

This command implements the second main idea shown in Example 2. It has three parameters, a keyword list, an HTML page or fragment, and a number analogous to the previous commands. The command uses the following algorithm (excluding the third parameter for now).

```plaintext
all-till-next-title(keywords, HTMLpiece){
    Tags<-NIL
    S<-find_title(keywords, HTMLpiece)
    while (surrounded_by_a_tag_pair?(S, HTMLpiece) 
          or 
          not(have_all?(html_piece_after(S, HTMLpiece), Tags) 
               and 
               info_found = TRUE) 
          or 
          not_reached_end_of_a_structure(S, HTMLpiece)){
        if(surrounded_by_a_tag_pair?(S, HTMLpiece))
          { 
              Tags<-Tags + surrounding_tag(S, HTMLpiece) 
              S<-expand_to_include_tags(S, HTMLpiece)
          }
        else {
            S <- S + html_piece_after(S, HTMLpiece)
            info_found = TRUE
        }
    }}
```

The algorithm starts by finding a title. It expands the title from both sides when there is a start/end tag pair surrounding the title, keeping the expanding tags. It adds more information to the end of the string when expansion from both sides cannot be done. The `html_piece_after` routine finds the next node after `S` where the node is either an unformatted string that ends just before a tag or a string that starts with a tag and contains the structure started by that tag. The reader can look at Section 4.3.3, HTML Parser, for more details.

Applying the algorithm to a web page at this point is useful as an illustration. The area information of a country is the target for this example. The HTML code surrounding the fragment of interest is below:

---

12 When looking for the title, the current implementation searches for the keywords and no other structure. The string found is the same as the titlefound described in Section 4.2.1, table-with. This is called a 'title' not because the algorithm is searching for a title, but because it treats what it finds as a title.
The target HTML fragment is the table row that contains the key word 'Area' and the detailed area information.

The execution steps for (all-till-next-title 'area' PAGE 0) are below:

<table>
<thead>
<tr>
<th>Step</th>
<th>Found String</th>
<th>Tag List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Area:</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&lt;div align=&quot;right&quot;&gt;Area:&lt;/div&gt;</td>
<td>Div</td>
</tr>
<tr>
<td>3</td>
<td>&lt;td&gt; &lt;div ... Area: ...&lt;/div&gt;...&lt;td&gt;</td>
<td>td, div</td>
</tr>
<tr>
<td>4</td>
<td>&lt;td&gt; &lt;div ... Area: ...&lt;/div&gt;...&lt;td&gt; &lt;td ... area info...&lt;/td&gt;</td>
<td>td, div</td>
</tr>
<tr>
<td>5</td>
<td>&lt;tr&gt;...&lt;td&gt; &lt;div ... Area: ...&lt;/div&gt;...&lt;td&gt; &lt;td ... area info...&lt;/td&gt; ...&lt;/tr&gt;</td>
<td>tr, td, div</td>
</tr>
<tr>
<td>6</td>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

Figure 3: Steps of a sample run for all-till-next-title command
In the first step, the algorithm finds the title ‘Area:’. In the second step, the algorithm sees the surrounding tag div, adding it to the tag list while expanding the string. The third step is also a tag list expansion step where this time td is taken into the tag list, expanding the string again to contain td. On the fourth step however, the string is not surrounded by a pair of tags, so the algorithm takes the next html piece without checking for the tags. No check was made at this step since no previous information was found, i.e., we only had tag list expansion steps. On the fifth step, the HTML fragment we have is surrounded by the tag tr. The algorithm adds it to the tag list, expanding the fragment to include the tag tr. The algorithm terminates here since there are no surrounding tags, and the next HTML piece which is the next row has the tags tr, td and div. Thus the algorithm thinks it found the next title and returns the fragment constructed so far.

The third parameter of all-till-next-title is analogous to the third parameter of previous commands. If there is more than one such title, setting the third parameter to 0 will let the algorithm use the heuristic described in the table-with command. Any positive integer will let user access the item of that number.

Ignoring the checking of tags in the tag list for the first information node found helps algorithm find at least some information that is not in the title, even if all the tags in the tag list exist in the information node found. This approach bears the risk of returning the next title from a list of titles with no information following, as within a navigation bar for in-page navigation. This is not generally an issue because, most of the time, if a title is both in a navigation bar and in another structure with some relevant information, the heuristic will pick the latter one, being the longer of the two (please refer to the heuristic in table-with command for further details on the heuristic). However, if the heuristic fails, one can still use the third parameter to extract the right structure.

It is interesting that, although designed for an example like Example 2, the algorithm works fine with the fields from the web page from Example 1. In its current simple form, all-till-next-title returns the needed information more often than any of the other commands. This is because most HTML documents use one or two tables that contain a big portion of the page, and the item-with command is too fine to capture a table row such as the example above. For example, item-with returns the string constructed at step 3, i.e., the td structure as it is the smallest item that contains ‘Area’, if it is called using (item-with ‘Area’ PAGE 0).
4.3. Other Commands & Features

4.3.1. get-all-results

Another command of Hap-Shu, get-all-results, allows the user to access all the extracted HTML fragments from a run of any of the Hap-Shu commands listed in Section 4.2, Command List. This method can be used for various purposes such as getting all the pictures in a page or using a different heuristic than the provided title heuristic.

The current implementation returns all the extracted fragments as a list of strings. Also, iterating the third parameter to access the results will cause multiple executions and may be slow. The script writers should prefer get-all-results to iterating the third parameter to access all the results, but if and only if more than one result is needed and multiple executions are slow. get-all-results was provided as a convenience for the cases when it is really needed. In the common script writing case, the author should not need to iterate through all outputs. He can figure out which number to use for iteration at the time of script creation. The current repair system implementation does not use any information about get-all-results, and will try to repair the Hap-Shu command associated with get-all-results, not the get-all-results command. Please refer to Section 6, The Repair System, for more information.

4.3.2. Chaining

So far, we have used a keywords list as the keywords parameter for the commands. Hap-Shu also allows the use of an HTML piece. In Section 4.2.5, all-till-next-title, we have seen in the example that item-with was returning only part of the information and table-with was returning too much information. One can get the row with the area information by calling item-with with 'Area' as the keyword and the whole page as the HTML fragment, and using its output as a keyword for another call to item-with again using the whole page as the HTML fragment.

The current implementation takes both the keyword list and the HTML piece in string format. The space character (ASCII code 32) is used as the delimiter for the keywords in the list. The
implementation considers any string with more than one pair of ‘<’ and ‘>’ as HTML code\textsuperscript{13}, in which case it will look for an exact case-insensitive match of the string in the page. It will then use the matched substring as the title found described in Section 4.2.1, \textit{table-with}. The design enforces a no-HTML-tag rule in the keywords parameter. If the script writer needs to use ‘>’ or ‘<’, he should use ‘&gt;' or ‘&lt;' respectively. Usage of any non-alphanumeric character is discouraged as they will most likely change on a redesign of the web page. In other words, they will significantly lower the expected life of a script. The script writers are discouraged from using HTML code not constructed by other Hap-Shu commands.

4.3.3. The HTML Parser

In implementing Hap-Shu, due to speed and robustness goals, a purpose-specific HTML parser was designed. This parser has one and only one command which is named \texttt{next-HTML-node}. The command takes one argument, a piece of HTML code. The command returns the node starting from the beginning of the argument, as a string. A node is defined as either a string that does not contain any HTML tag or a string that starts with an HTML tag and ends at the end of the structure defined by the tag. If the HTML piece starts with an end tag, the parser returns an empty string. Let us examine some examples to clarify the definition.

\begin{itemize}
\item \textbf{Input:} "This is one \texttt{<strong>example</strong>...}"
\item \textbf{Returned string:} "This is one"
\end{itemize}

In the example above, the first node is the string up to the first tag. Note that the returned string contains the whitespace character just before the tag.

\begin{itemize}
\item \textbf{Input:} " \texttt{<strong>Another</strong> example...}"
\item \textbf{Returned string:} ""
\end{itemize}

In the example above, the node is the two space characters before the tag.

\begin{itemize}
\item \textbf{Input:} "\texttt{<strong>Yet</strong> \textit{another} example...}"
\item \textbf{Returned string:} "\texttt{<strong>Yet</strong>}</item>
\end{itemize}

In the example above, the node is the piece defined by the tag \texttt{strong}.

\textsuperscript{13} The reason for allowing one HTML tag in the keywords parameter is to provide the script writer with a way to handle the cases where one tag must be present in the keywords list for the Hap-Shu command to work. It must be remembered that the HTML tags, excluding the one allowed tag in the keywords list, between the keywords in the document are ignored in finding the title found.
Input: "<li>badly formatted list item 1
<li>badly formatted list item 2</ol>"
Returned string: "<li>badly formatted list item 1"

In the example above, the node is the piece defined by the tag li. Many HTML code authors choose not to include the optional end tag for the list items. Here the structure, hence the node, is still well defined.

Input: "<img src=myimage.jpg> Here is an image ..."
Returned string: "<img src=myimage.jpg>"

In the example above, the node is the img tag.

Input: "<dt>term 1
<dd>definition for term 1
<dt>term 2 <dd>..." 
Returned string: "<dt>term 1
<dd>definition for term 1"

This example is different from the previous examples as it seems to return two nodes instead of one. Both the term being defined and its definition were returned. This was a design choice made only for these special HTML tags. These tags do not have to have matching ending tags, despite the fact that they modify the context of the text that follows them. These tags appear together most of the time, and if they do, each dt tag is conceptually connected to zero or more dd tags that succeed it by a term-definition(s) relation. Thus, together they were considered as a node.

The HTML parser considers HTML tags in three groups:

- Single tags: Tags that do not have a matching end tag. Ex: img, hr, meta
- Special tags: Tags that may not have an end tag but still are considered valid by browsers and tags that need special treatment like dt as described above. Ex: tr, p, li
- Normal tags: The rest of the tags, which are expected to have an end tag for validity. Ex: a, strong
The parser handles each of the three groups differently. For each description below, assume
the parser is parsing a string that starts with a tag and is trying to find the node associated with
this tag in the rest of the HTML string.

Parsing out a single tag is straightforward.

The HTML parser handles a normal tag as if it is the only tag in the HTML string. Thus, to
find the matching end tag, the parser looks for only the normal tag. The algorithm then is
analogous to finding the closing pair of an open parenthesis in a math equation or a lisp script.
Any other tag is ignored in the algorithm. If an end tag is not found, an exception is thrown.

To handle special tags, each special tag is associated with two lists of tags: the list of possible
closing tags and the list of recursion tags. If the HTML string starts with a special tag, the
parser finds the first occurrence of any tag from the two lists associated with the special tag. If
the tag found belongs to the first list, the parser checks if the tag is the actual end tag for the
special tag, i.e., if it is a slash character in front of the special tag. The parser then returns the
substring from the beginning up to the tag found, including the tag if it is the end tag of the
special tag. If the tag found belongs to the second list, the parser recursively finds the node
starting from the tag, and continues the search from the end of the node.

Example: For the tag tr, the closing tag list is ('tr', '/tr', '/table') and the recursion
tag list is ('table'). The first list means the tag tr can be closed by another tr
tag, the table closing tag /table, or its closing complement /tr. The second
list says that if a table opening tag is found before the other tags, the end of that
table has to be found before continuing the search.

Most of the HTML tags are not special. For example although tr is a special tag, table is a
normal tag. While finding a node of a normal tag; the HTML parser ignores all other tags that
are present in the HTML string. In the case of a node of a special tag, the parser deals with the
very few tags listed in the two lists of the special tag. This design greatly reduces the risk
involved in encountering a badly formatted HTML fragment, which is not uncommon.
Considering that all the commands in Hap-Shu use this parser to find their respective
structures, the design of the parser significantly improves the resistance of the scripts to
HTML formatting errors.
5. The Editor

The editor has been crucial in achieving the goal of easy-to-implement wrappers. It also allows the testing of any wrapper to verify its robustness across different pages from a proper set of HTML documents.

The editor is a set of HTML documents, CGI scripts and Guile scripts. The editor can be loaded using any browser that supports dynamic HTML content. Let us go through the steps of creating a simple wrapper using the editor to illustrate how it works.

**Step 1:** We load the editor to a browser.

![Figure 4: A snapshot of the editor after loading](image)

The editor is split into five frames. From now on, each frame will be referred by the number seen in Figure 4 above. At this stage only frame #1 is useful.
Step 2: We now enter the URL of the page we will work on in frame #1. For this walkthrough, the URL will be http://www.50states.com/massachu.htm.

After we enter the URL, the editor loads the URL into frame #4 and two forms to frames #2 and #3. In frame #2, we can enter a set of keywords. We will talk about frame #3 later.

Figure 5: A snapshot of the editor after loading a URL for script creation
Step 3: We now enter a set of keywords to specify an HTML fragment. For this walkthrough, we will use “border states”.

The editor submits a request to the back-end system that generates all possible Hap-Shu commands that uses the set of keywords we entered. The generated output is loaded in frame #5. In this frame, all the possible Hap-Shu commands and their results are shown. In Figure 6, we see that if we run

(all-till-next-title::run "border states" PAGE 0)

we will get what we wanted. We can see the rest of the commands and their results by scrolling down in frame #5. Since we are satisfied with the wrapper above, we can now test it or modify it according to our needs. We will talk about testing later. Let us modify this wrapper so that its output includes the area information as well.
Step 4: We copy the script to the text area in frame #3 by pressing the button labeled “Copy to Test Area” and we edit it in the text area to have the following script:

```
(string-append (all-till-next-title: :run "area" PAGE 0)
 (all-till-next-title: :run "border states" PAGE 0))
```

We assumed here that since all the fields on the web site seemed similar, running the same command with “area” will give us the area information. Please note that the modified script now uses both Hap-Shu and Guile commands. Just beneath the text area is a form where we can enter a symbol. We will enter “Massachusetts” into the symbol field for now. We will see the use of the symbol later in this section. We now check the checkbox next to it to tell the editor to use the page in frame #4 for testing and press the test button. The results of the test are given in frame #5 but to study them better, the results are shown in a new window in Figure 7.

![Figure 7: A snapshot of Frame #5 opened in a new window to see testing results](image-url)
The result turned out to be what we wanted. The actual Guile script that was run is also shown in the output so that we can copy it directly into a Guile program. The additional lines introduced by the testing code are for downloading the page and finding and appending the base for the HTML document so that links and pictures work properly.

The walkthrough above shows a simple wrapper creation with a simple test done only on the loaded URL. However, the editor has other features that will help script writers in generating and testing scripts. Let us go over these features.

The implemented editor has a more powerful testing mode. The editor can connect to the database Omnibase uses and pick a number of randomly sampled symbols for testing. For this functionality to work properly, the script writer must have already put the symbols and two scripts into the database. These two scripts are URL and PAGE scripts, which are called with a symbol and return the URL of the page for the symbol and the actual PAGE respectively. Please note that these scripts and symbols must be present in Omnibase regardless of whether Hap-Shu is used, i.e., putting them into the database is not an extra task necessitated by Hap-Shu. The editor also allows testing of any one symbol entered in frame #3 without loading its URL into the editor. The requirements for this functionality are the same as the requirements for random symbol testing. As we saw in the walkthrough, the script writer can still generate wrappers without a database, by loading a URL and testing the wrapper on it.

The editor has two other simple but useful features. One of them is the ability to remember the editor’s current state, which was achieved using HTTP cookies. This feature helps in browser crashes, unwanted reloads of frames and throughout the wrapper generation for one site. The second feature is a command index at the end of the command output generated when a keyword set is submitted. The index can be reached by clicking the command index link at the top of each command and serves as a navigation bar for the page. This feature helps with the long pages where the results of many commands are long as well.
6. The Repair System

The automatic repair system is an important part of the system; it increases the life time of scripts dramatically by repairing them in the case of major layout changes. The design of Hap-Shu and the run-time capabilities of Guile rendered this simple, fast and successful automatic repair system possible. The design principles of Hap-Shu will be described first with an introduction to how the automatic repair is done. The actual design will then be discussed.

6.1. Separation of the Content from the Format

Hap-Shu is designed to separate the content from the layout and/or formatting information. Each command in Hap-Shu contains the formatting information in the design of the command and obtains the content information from the keywords parameter. In other words, each Hap-Shu command in a script separately knows the expected content and formatting of its output.\(^\text{14}\) The ability to manipulate these two important features of the target information separately helps the repair system to cope with many major layout changes. As mentioned earlier, every so often websites change their format. On the other hand, it is less common for a website to serve new content or remove old content. In other words, trying only different layout structures is sufficient most of the time for the repair system to reach its target fragment.

A wrapper that uses regular expressions with the content and the format information intermixed with each other is generally difficult to fix. In the event of a change in layout, Hap-Shu, knowing what kind of content it is looking for, can find a new structure containing the old content much faster and more accurately than a system that only uses cached results of prior runs to figure out the new format.

For example, consider one Hap-Shu and one regular expression-based script trying to extract the director information from the following HTML piece:

```html
<table>
<tr><td> The movie was made in 1989. </td></tr>
<tr>
<td> The movie was directed by X Y. </td>
</tr>
<tr><td> The movie cost A$. </td></tr>
</table>
```

\(^\text{14}\) Note that the script itself does not know the content or the format of the output due to the writer's flexibility in combining Guile and Hap-Shu commands.
The Hap-Shu script is one simple command (item-with ‘directed by’ PAGE 0). The other one has a regular expression that matches the beginning and the end of the fragment. This regular expression would look something like:

```
"<td>[^<>]+\bmovie\W+was\W+directed\W+by\b[^<>]+</td>".
```

Both of the scripts are able to extract the information with the current layout. If the HTML piece changes its layout to become the following HTML piece, scripts will need repairing.

```
<UL>
  <LI><I>Made in :</I> 1989 <BR>
      <I>Movie was directed by :</I> <A HREF=XY.html>X Y</A><BR>
      <I>Cost</I> A$<BR> </LI>
  <LI><I>Sequel made in :</I> 1989 <BR>
      <I>Sequel directed by :</I> <A HREF=XY.html>X Y</A><BR>
      <I>Cost</I> B$<BR> </LI>
</UL>
```

The example is a tailored one to cause problems for both scripts on different areas. Assume that both scripts have an automatic repair system that by using previously sampled results can successfully detect which part of the new HTML page is the new target. From now on, assume the new target for the example is detected as the following:

```
<I>Movie was directed by :</I> <A HREF=XY.html>X Y</A><BR>
```

The repair system of Hap-Shu can try the other commands and other third parameters keeping the same set of keywords, ‘directed by,’ to see if any similar result is found. In this example (all-till-next-title ‘directed by’ PAGE 1) works perfectly. The repair is successful. Note that (all-till-next-title ‘directed by’ PAGE 0) would have given us the sequel director, due to the title heuristic described in Section 4.2.1, table-with. Execution of the automatic repair for a simple script like this example takes a few seconds on average, thanks to both the repair system’s knowledge about the content and the speed of the HTML parser.

The repair system for the regular expression script is in trouble. It knows where the target piece is and it has to create a new script that can extract the target piece. Modifying the old script is not a good option because understanding regular expressions enough to modify them is extremely difficult for a computer program, especially if the author of the script is a human.
Obligating the script writers to learn the parsing and working methods of the repair system to write repair-system-friendly expressions is neither a friendly nor a dependable approach.

Creating a new regular expression for the target fragment is not easy. A simple pattern built around the words ‘directed by’, which can be obtained with a simple heuristic of taking the alphanumeric parts outside the tags from the original pattern, may fail due to the existence of a very similar pattern around another list item containing ‘directed by’. The example shows another item giving the information about the director of the sequel. Three possible solutions are described below.

One way for the second repair system to solve the problem of multiplicity is to find a tag path from the beginning of the document to the beginning of the target piece. Two such paths for this example are \{ul \rightarrow 2^{nd} li \rightarrow 2^{nd} i\} and \{5^{th} i\}. From there, the system looks for the keywords ‘directed by’. It later pinpoints the end of the target piece using another tag path, like \{2^{nd} br\} or \{/li\} for our example.

A regular expression using the \{ul \rightarrow 2^{nd} li \rightarrow 2^{nd} i\} path followed by ‘directed by’, ending with \{/li\} will look like:

```
"<ul>.*<li>.*<li>.*<i>.*(<i>.*directed.+by.*</i>)</li>"
```

The expression is now much more complicated and will most likely run very slowly due to the numerous wildcards in it. A major problem with tag paths is the necessity of parsing a big portion of the HTML code to traverse them. The process may be quite slow and badly formatted HTML code could cause failure in automatic repair. Another problem with using tag paths is how to choose a path over another. Short paths such as the \{5^{th} i\} path above are very vulnerable to unexpected introduction of similar tags, even if the page is properly formatted. Long paths are exposed to many possible formatting errors across the document and generally have longer running times. The tag paths in general tend to break on minor layout changes due to their absolute dependencies on the existence of specific tags in specific order and number. The more repairs a script is put through in the past the higher chance the script will return a wrong fragment due to a non-zero possibility of an error introduced by the repair.

A second, smarter regular expression building solution is to examine several pages to arrive to the template of the page. Assuming the repair system knows the target for each of the several pages, the repair system can generate a common regular expression system that can extract
each of these targets successfully. The papers by Grieser et al. [8] and Ashish et al. [6] present the use of similar approaches. There are two problems with this approach. The first problem is to find the target HTML fragment for each of the several pages. The second problem is the broad range from not being able to find a template for the page to overfitting it. The solution to the first problem, assuming the system can find the HTML target for each one of them as described in the beginning of this example, reduces to finding the common template for the target fragments. Finding the common template for the target fragment is also exposed to the second problem listed for finding the common template for the page. The second problem arises from many web sites having different fields listed for different symbols. For example, some of the actor pages in the IMDb web site do not include a mini-biography section. Another example is Monaco, which does not have any airports; thus no “Airports - with paved runways” field is listed for it in the Factbook. To be able to arrive at a good template for both the page and the target fragment, the system needs to have access to a good set of examples from each different layout. A good set of pages means the set of pages that is enough for the template to be extracted. For example the set of pages for some countries in Asia from the Factbook where the neighbor list for each country includes China, is not a good set to construct a template for the neighbor information. The template will overfit the data set in this case by looking for China in the neighbor list. Construction of the set has to be done by humans, who can oversee a detail like the China example. Even with no human error involved, there may not be enough pages for a template to be made for a special case, in which case the template will overfit. Overfitting decreases the resistance of scripts to layout changes.

The third solution is to modify the content information to include ‘movie’. This method is highly dangerous as it introduces words that were not previously in content information. To distinguish the original set of words from the newly introduced words, the repair system can keep track of the new words, which increases overhead for the system. Without a tracking mechanism, the pattern may end up having too many words if no word gets deleted or end up losing the original words and hence the ability to find the target HTML fragment in a new HTML layout. Even with the tracking of new words, the system may still end up with many words, i.e., very complicated expressions.

Without the explicit separation of the content and the formatting, the repair system cannot deduce the content from the broken scripts. The content is what the scripts use as a landmark and a repair system that cannot obtain the content from the scripts has to guess it using the only other source available, the cached results from old runs. With only cached results in its
arsenal, the automatic repair system can only hope to create a new wrapper that returns a result “similar” to the cached result. There may be small mistakes at each repair step that may accumulate to cause future wrappers to extract irrelevant HTML pieces.

Hap-Shu eliminates the use of HTML tags or any other literal formatting input. The only time it allows the use of tags is described under Section 4.3.2, Chaining. The script writer thus has no choice but to provide solely content, which will be the easy option for him anyway. Finding the content is much easier than dealing with the format–content mixture in HTML documents.

6.2. The Design

The repair system has the following goals:

- Detection of broken scripts
- Automatic repair of broken scripts
- Detailed error reports
- Report submission to various recipients through multiple channels
- Extensibility

The repair system utilizes six major parts. Two of these predate Hap-Shu. These are Omnibase and Users. Four other parts were developed to meet the goals:

- Cached-Results Database
- Hap-Shu Commands’ Mutators
- Repair Results Handler
- Main Repair System

The following diagram shows the interaction of all six parts:

---

15 The script is considered broken if for a given symbol it fails to return the result it had returned before regardless of whether it is an HTML fragment or an error.
In Figure 8, a solid line between two parts represents a mandatory communication link, and a dashed line represents an optional communication link. For example, the link between Users and the Main Repair System is optional and can only be initiated by Users.

Below are the descriptions for and responsibilities of each major part and its communication links, as shown in Figure 8.

**6.2.1. Cached Results Database**

The Cached Results Database is responsible for managing prior results of scripts. These results are used in the repair process. The Cached Results Database supports the following functionalities:

- **List of repair-disabled scripts**

  The list of repair-disabled scripts holds the information about which scripts of which classes should not be repaired. This functionality is useful for disabling automatic repairing for scripts that have Hap-Shu commands. The scripts without any Hap-Shu commands will be detected and not be repaired.
• **Storage and retrieval of results**

The database is responsible for storing and retrieving the results of scripts. Each result is stored along with the page it was extracted from, the symbol that the script used to get the page, the class the script belongs to, and the attribute the script is designed for, i.e., the name of the script. The stored results are unique on their <class, attribute, symbol> triples. In other words, there can be at most one entry for a given class, an attribute and a symbol. There can be multiple results for the same script given that the symbol is different. The storage of results can be requested by both the Main Repair System and the Users. The retrieval of results is requested only by the Main Repair System.

• **Tracking the error detection date for each result**

The database is also responsible for storing and retrieving, for each result the last day an error was detected if the script could not be repaired. If the script was repaired or no error has been detected yet, the error detection date is set to -1. The request for storage and retrieval are requested only by The Main Repair system.

### 6.2.2. Omnibase

Omnibase [7] is the system that stores and uses the symbols and the scripts for production purposes. Each script in Omnibase is associated with a class and an attribute. There can be only one script for each <class, attribute> pair. As mentioned before, Omnibase is used by START as the only interface for the WWW. Omnibase is responsible for the execution of all wrapper scripts and the WWW communication needed to obtain the data. In the Hap-Shu repair system, Omnibase acts as the trigger for both caching and repairing procedures. Omnibase chooses to cache the successful result of any script if no other result was cached for the class and the attribute associated with the script. It requests the storage of the result through the Main Repair System. Omnibase also notifies the Main Repair System of errors in the execution of scripts, triggering automatic repair.

The functionalities provided by Omnibase to the repair system are:

• **Storage and retrieval of scripts**

Omnibase stores and retrieves scripts for only the Main Repair System. The request for the storage of scripts is an optional communication link, which is only used when the production database is allowed to be modified.
Execution of scripts

Omnibase executes scripts, which may connect to a server in the WWW, for only the Main Repair System. It is the Main Repair System’s only access to the WWW.

Storage and retrieval of both symbols and scripts, execution of scripts and creation and deletion of classes

Omnibase can be directly modified by Users with all privileges. However, this link should not be used unless manual intervention is the last remaining option.

6.2.3. Users

Users are the human module of the system. Users submit all of their requests through the Main Repair System.

Their abilities in the repair system are listed below:

- Creation of a Test Suite

Users can force caching of the result of a specific script executed with a specific symbol from the same class with the script. This allows Users to cache the results of some special runs using some special symbols whose information pages have slightly different layouts than the other pages in the proper set of HTML documents of the class. For example, if the ‘Area’ information was in different tables for European and Asian countries, the script responsible for ‘Area’ would have two execution paths depending on the continent of the country. In this case, Users should cache the script’s results for one country from Asia and another from Europe. This lets the repair system successfully repair all execution paths.

- Forced repair

Users cannot trigger a repair event directly, but they can use the production system to run the broken script. This will cause Omnibase to trigger the repair procedure.

- Modification of the list of repair-disabled scripts

Users can select which scripts are in the list of repair-disabled scripts, through the Main Repair System. This is useful for excluding scripts that have any Hap-Shu commands.
 Modification of Omnibase

As mentioned in Section 6.2.2, Omnibase, Users can modify Omnibase directly by adding new classes or fixing broken scripts that could not or cannot be fixed through Hap-Shu. After each manual repair, a user should update the cached results, as the repair system can only detect the modification of the script. It cannot know whether the new script is correct or not.

The Repair Results Handler maintains a list of users to inform about its activities. The users in this list are notified of every repair attempt and its result via email.

6.2.4. Command Mutators

Each command in Hap-Shu has a mutator. A mutator is a script that uses the broken command with its old parameters and the new page to create a list of commands and parameters that may return a result similar to the old result. This list is the list of mutated commands. Each mutator is responsible for one command. The Main Repair System is the only part that requests this functionality. Note that the mutator does not know the old result of the command. It has to create all reasonable mutations for the given command. Comparison of the new results from the list and the old result is done by the Main Repair System. This abstracts the comparison function from the rest of the system, making any future improvement to the comparison function easy to implement. However, the mutator has access to the keywords parameter. The mutator can use this parameter to make intelligent and targeted mutations.

The mutators in the current implementation are very simple. Instead of using the content information, formatting used by other commands and the page to find possible smart mutations, the mutators just mutate the command and the third parameter to other possibilities, leaving keywords parameter untouched. Here are some examples from the list of mutated commands that the mutator for table-with returns when called with (table-with 'Area' PAGE 0):

(item-with 'Area' PAGE 0)
(all-till-next-title 'Area' PAGE 3)
(all-till-next-title 'Area' PAGE 0)

There is another useful mutation that is used by the mutators. Given the broken command:

(item-with 'Content1' (table-with 'Content2' PAGE 0) 0)
the mutator for `item-with` will also return:

```
(item-with 'Content1' PAGE 2)
```

In this mutation, the second parameter is replaced by the actual page, with the hope that one command may be sufficient to extract the target.

Using simple mutations is useful for various reasons. One reason is that it allows the separation of implementation details among commands. For example, `item-with` does not have to know which structures are in `table-with` to use it in a mutation. Knowing that `table-with` returns a table-like structure is sufficient. On the other hand, to make more complex informed, i.e., smart mutations, `item-with` needs to know what kinds of HTML structures are returned by `table-with`. Letting mutators use such information will damage the extensibility of the whole system. In its current form, any mutator or command can be modified without the necessity for changes in other parts of the system. Another reason for using simple mutations is the speed of their creation which is much higher than that of smart mutations. Trying to find targeted mutations using other commands’ implementation knowledge will cause the mutator to try many different formats in different parts of the page. This will lead to mutators being more exposed to formatting errors. This method will also cause code duplication in every mutator, with slight local modifications. Maintaining the mutators will then be very difficult, damaging the robustness and the extensibility of the system.

### 6.2.5. Repair Results Handler

The Repair Results Handler is the module that is responsible for handling the collection of repair reports and delivering them to their intended targets. The Repair Results Handler receives only two requests, addition of more sub-reports to the current report and sending the report out. Both of these requests are sent only by the Main Repair System. Once the send-out request has been given, the Repair Results Handler uses the reporting channels enabled for submission to send the collected report. The Repair Results Handler has three channels it can submit the report to. Each of these channels can be enabled or disabled easily from within the code, but modifications after the system is up are not supported in the current implementation. The first of these channels is standard error output. This channel is useful for debugging and proper logging of repair attempts in the system. The second channel is the email list mentioned in Section 6.2.3, Users. The email list is also easy to change offline, but is not
modifiable once the system is up. If enabled, an email with the report will be sent to the email list for each send-out request. The third channel is a CVS\(^{16}\) repository. The scripts in Omnibase are stored also in a CVS repository. If enabled, the report will be the commit log for the new scripts file for the class that was repaired. After a report is sent out, the report buffer is cleared for a new fresh report.

### 6.2.6. Main Repair System

The hub of the repair system is the Main Repair System. It is involved in almost all communications in the system. With the help of the five other major parts, the Main Repair System is simple but powerful. Here are the functionalities it provides to the system:

- **Checking the list of repair-disabled scripts**

  This service is provided only to Omnibase, for it to decide what to cache automatically. It is basically the access method for the list stored in the Cached Results Database.

- **Caching the result of a script**

  This service is provided only to Omnibase and Users. With the help of Omnibase, the Main Repair System runs the script with a given symbol, or a random one if no symbol is provided. If the script successfully terminates, the result is stored via a request to theCached Results Database.

- **Comparing the result of a script with the cached results**

  This service is called only by Omnibase with a script when the script fails. The Main Repair System retrieves all the cached entries for the given script. Each entry is a result from a different symbol. For each entry, the Main Repair System first checks the last error detection date. If the entry was already examined today, it is skipped to prevent multiple reports for the same class in a single day. If the entry was not examined today, the page for the symbol in the entry is downloaded to check for errors like relocation of the page or removal of the page. If there is no such error, the system runs the script with the symbol of the entry and compares the result with the result in the Cached Results Database. If the results are similar, the error detection date for the entry is set to -1, as described in Section 6.2.1, Cached Results Database.

---

\(^{16}\) [http://www.cvshome.org/](http://www.cvshome.org/)
The similarity of two results is tested as follows:

- All HTML tags are removed from both results with only the values of the attributes of image tags kept, such as the values for src and alt attributes.
- All non-alphabetic characters are removed including numbers. Removal of numbers is the easiest way to deal with the dynamic nature of numbers in HTML pages. Most of the numbers in an HTML page change very often, which damages a similarity test.
- All multiple consecutive whitespace characters are reduced to one space character.
- For every word of cached result that is also in the new result, a score, which is initially 0, is increased by the length of the word found (+1 to account for spaces between words) divided by the length of the new result. The score is basically the ratio of the new result that is also in cached result.
- The previous step is repeated, this time using the new result as the cached result and vice versa. A second score, which is initially zero, holds the ratio of the cached result that is also in the new result.
- If both of the scores pass a certain threshold, the two results are considered similar.

The actual implemented similarity algorithm has two thresholds, one for long and one for short results. For short results, which are shorter than 30 characters after all the removal process, the threshold is 80%. For long ones, the threshold is 70%.

Going back to the Main Repair System loop, if the new result is an error or the results are not similar then the actual repair code within the Main Repair System is called with the new page, the cached page, the symbol, the class, and the attribute. The repair code first gets the script itself from Omnibase using the class and the attribute. The first use of the run-time power of Guile is made at this step. The $\texttt{get}$ method in Omnibase that is called to run the scripts is modified so that if the page script, which downloads the page from the WWW, is called for the symbol in the entry, the cached page is returned instead of downloading the new one from the WWW. From here on in this section, any execution of any script on the cached page implies the prior application of this modification. A similar
modification is made on any script used with the new page so that the script will use the already downloaded new page when instead of downloading the page from the WWW.

The repair code runs the script on the cached page to see if the result matched the cached result. The only time when they do not match is when a modification to the script was done by another source, possibly a user, and the cache was not updated. In such a case, an error sub-report is sent to the Repair Results Handler; asking Users to manually cache the modified script (refer to the Modifications to Omnibase item in Section 6.2.3, Users).

If the script was not modified by another source, the repair code modifies the script so that each Hap-Shu command is replaced by the command repair-subresult with two parameters, an ID, which is unique in the script, and the Hap-Shu command whose execution is delayed with the help of another powerful feature of Guile. The delay of the execution lets repair-subresult execute the Hap-Shu command in a controlled environment.

Here is a script before the modification described above:

```scheme
(lambda (symbol)
  (let* ((URL (get "test-class" symbol "URL"))
         (PAGE (get "test-class" symbol "PAGE"))
         (base (helper::find-base URL PAGE)))
    (string-append base
      (begin
        (picture-with::run "flag"
          (table-with::run "Flag" PAGE 0)
          0))))
)
```

The Guile script above is a function that takes one parameter, `symbol`. Using the get method of Omnibase, the "URL" script and the "PAGE" script are called. The name of this class is 'test-class'. After getting the URL of the page, and the PAGE itself, an algorithm to find the BASE tag from the HTML is called. If there is no BASE tag, it will be constructed from the URL. The rest of the script is two chained Hap-Shu commands. The script first looks for a table with 'Flag' in it, and then takes the picture that has 'flag' in its name or alt text from the table found. Please note the readability of Hap-Shu commands. This script after the modifications described above will be the following:

```scheme
(lambda (symbol)
  (let* ((URL (get "test-class" symbol "URL"))
         (PAGE (get "test-class" symbol "PAGE"))
         (base (helper::find-base URL PAGE)))
    (begin
      (picture-with::run "flag"
        (table-with::run "Flag" PAGE 0)
        0))))
)
```
Both of the Hap-Shu commands are delayed using "lambda ()" and are parameters to two repair-subresult calls with two unique IDs, unique within the script.

repair-subresult is a very simple function that runs the delayed command inside an exception catcher and throws any successful result inside a repair-subresult exception attached with its unique ID parameter. If the result is an error, it gets caught by the exception catcher. The type of error is checked. If it is a repair-subresult exception, the exception is re-thrown to upper levels without modifying the attached ID. Otherwise, the error is wrapped in a repair-subresult exception attached with the unique ID parameter. In other words, the result of the first executed Hap-Shu command is thrown inside a repair-subresult signed by the immediate wrapping repair-subresult function with its unique ID.

The modification of the script allows the repair system to access the result of each Hap-Shu command in the script. If the modified script is called, the result of (table-with::run "Flag" PAGE 0) will be thrown in a repair-subresult exception with the unique ID 2 coming from the innermost repair-subresult. The exception will then get caught by the other repair-subresult which will re-throw it since it is coming from another repair-subresult node. The result of the innermost Hap-Shu command is successfully passed to the caller of this script, with the ID 2 attached for tracing the location the result was sent from. The ability to modify scripts and delay their execution allows the system to reach any intermediate result, which is a tool that the repair system uses extensively. I will call these exception throwing nodes error nodes as their purpose is to catch internal errors cause by Hap-Shu commands.

The repair process from here on is very simple. The repair code starts with two copies of modified scripts. It runs one on cached page and the other on the new page. The returned
results are compared. If they come from the same error node, which is verifiable by the IDs attached to the exception, and are strings and similar, the error nodes that threw the results are removed from the script, so that the next execution will return the results from other error nodes. If the results are not similar but are from the same node, the mutator for the command encapsulated in the error node is run for a list of possible mutated commands. For each mutated command, the repair code substitutes the original one with mutated one and runs the script to get the mutated subresult to compare it to the subresult from the cached data. If any of the mutated commands is successful in returning a similar subresult, the error node is removed and substituted with the mutated command. If there is no successful command, the error node is substituted with an exception throwing code that reports the failure for the Hap-Shu command from the node.

If the results are coming from two different nodes, then their closest common ancestor node, which is well defined as Guile scripts are trees in parenthesis representation, is found. If there is no such node, the repair attempt is reported as failure. This is the case where the two scripts ran on two different execution paths, probably due to a Guile command introduced by the script writer. If there is a common ancestor, all error nodes in the sub-trees rooted by the common ancestor node in both scripts are removed. This allows a chance to match the results from the two scripts of a common Hap-Shu command when the repair for the sub-tree rooted at that common command has failed.

If the repair system manages to remove all the error nodes with success, a last check is made with the new mutated script and the result is compared for similarity. If the results are similar, the result of the script is cached; a success sub-report with the old script, the new script, the old result and the new result is added to the current report. The error detection date of entry is set to -1, as described in Section 6.2.1, Cached Results Database. If the flag for production database modification is enabled, the new script is put into the production database for seamless repair. If the flag for CVS is enabled, the scripts file gets checked out and modified to reflect the changes. The file however is not committed yet as the system may modify the script again using other entries, for example for different

17 The possible mutations are sorted in the repair code to try the commands with the smaller third parameters first. Thus, the code first tries the commands with 0 as the third parameter, and then 1, and so on. This helps find the good command faster if it exists.
execution paths. Committing is done by the Repair Results Handler as described at the time described in this section.

If the repair failed, the failure sub-report is added to the report. The error detection date of the entry is changed to the current date.

When the repair code finishes its repair attempt regardless of the outcome, it proceeds to the next entry in the Cached Results Database for the script till it goes over all of them and checks each one for repairs. Any repair modifications made to the script in any entry are kept, so that the rest of the repairs done with the remaining entries will be done on the modified script, cumulating the repair modifications as the repair system proceeds.

When all entries are finished, the repair code checks if any error was detected that initiated the repair. If an error was detected, all the scripts of the class are given to the comparison script and each script is checked for repairs. At the end of a run where errors were detected and maybe repaired for a whole class, a send-out request is sent to the Repair Results Handler, which will commit the CVS file with the repair report as the commit log if CVS is enabled. Other channels of error reporting for the Repair Results Handler are given in Section 6.2.5, Repair Results Handler.
7. Future Work

The editor can be improved so that after the user creates one prototype script for an attribute in the page, the editor can generate a set of similar scripts that extract other similar structures in the page. These scripts can be presented to the script writer for modification. This idea is presented in the paper by Ashish and Knoblock [6]. For example, all the fields in a Factbook page can be extracted using the all-till-next-title command. When the user creates the script for ‘Area’, by analyzing the Factbook page, the editor should be able to generate the scripts for all other fields, which will differ from each other by just the keywords parameter. The scripts after being modified if necessary by the script writer can be directly integrated into the code repository as well as the production database if necessary.

Hap-Shu can also be used as a passage retriever. It is successful in getting the HTML fragments related to given keywords, even from badly formatted HTML documents. Using it to retrieve relevant pieces from a set of HTML documents obtained by a document retriever, like Google\textsuperscript{18}, may provide a good set of passages for a question answering system to play with.

\textsuperscript{18} http://www.google.com/
8. Conclusion

Hap-Shu achieved all of its goals set in the beginning of this thesis.

- The script creation process was made easier and more intuitive. The editor helps script writers create scripts without looking at the actual HTML code. A person with no HTML, Guile or regular-expression knowledge can still create scripts. The created scripts are easier to read and modify than the old regular-expression based scripts.

- The scripts are more resistant to format and layout changes, reducing the frequency of repairs. Each Hap-Shu command include the information for similar structures such that any change from one structure to another similar one covered by the same command, will not affect the script. For example, using list items instead of table cells will not break the script using item-with. HTML parser increases the resistance of scripts to badly formatted HTML.

- The scripts execute quickly. Hap-Shu commands do not execute faster than script-specific regular expressions, however, Hap-Shu commands still execute in very close speeds\textsuperscript{19} with the regular expression based scripts.

- The repairing system may repair the broken scripts.

- The automatic repairing system generates detailed easy-to-understand error reports even if the automatic repair fails. Different types of reports include source page errors such as page-not-found or page-moved, success reports with old and repaired versions of both the scripts and results, failure reports as detailed as the errors occurring deep inside the script and where they specifically occurred.

\textsuperscript{19} No controlled speed test was performed.
9. References

1. B. Katz and J. Lin, “START and Beyond”, In 6th World Multiconference on Systemics, Cybernetics, and Informatics, 2002


