SeMole:
A Robust Framework for Gathering Information from the World Wide Web

by

Hyung-Jin Kim

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 25, 1998

Author_____

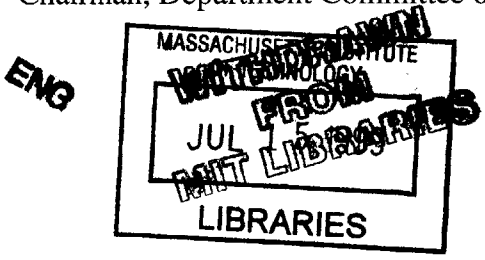
Department of Electrical Engineering and Computer Science
September 25, 1998

Certified by_____

J. Lee Hetherington
Thesis Supervisor

Accepted by_____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses



SeMole:
A Robust Framework for Gathering Information from the World Wide Web

By

Hyung-Jin Kim

Submitted to the Department of Electrical Engineering and Computer Science
on Sept 25, 1998, in partial fulfillment of
the requirements for the degree of Master of Engineering in Computer Science

Abstract

This thesis describes seMole (*se*mantic *Mole*), a robust framework for harvesting information from the World Wide Web. Unlike commercially available harvesting programs that use absolute addressing, seMole uses a semantic addressing scheme to gather information from HTML pages. Instead of relying on the HTML structure to locate data, semantic addressing relies on the relative position of key/value pairs to locate data. This scheme abstracts away from the underlying HTML structure of Web pages, allowing information gathering to only depend on the content of pages, which in large part does not change over time. We use this framework to gather information from various data sources including Boston Sidewalk and the CNN Weather Site. Through these experiments we find that seMole is more robust to changes in the Web sites and it is simpler to use and maintain than systems that use absolute addressing.

Thesis Supervisor: I. Lee Hetherington
Title: Research Scientist

Acknowledgements

This thesis would not have been possible without the support of many people in the Spoken Language Systems Group. First, I would like to thank Victor Zue for his patience, understanding, and his inspiration. Without his leadership, this thesis would have never been completed. To Lee Hetherington, my supervisor, I owe my entire thesis. He has always been crucial supporter of my work and his insights into seMole have been invaluable.

I would also like to thank Giovanni Flammia for introducing me to the Spoken Language Systems Group and giving me the chance to develop seMole. It was his interest in a bridge between the Web and machines that inspired me to work on seMole. Without his spark, seMole would have never been made. I am indebted to Ray Lau for his help in developing seMole. I have relied on his systems knowledge (which is second to none) many times in my work. I would also like to thank Stephanie Seneff, Joe Polifroni, and Philipp Schmid for their courage to merge seMole into their systems.

Finally, I would like to thank my parents, Young Hwa Kim and Young Il Kim, for being who they are. Without them, nothing would have been possible. For that, I am indebted to them more than I can imagine.

This research was supported by DARPA under contract N66001-96-C-8526, monitored through Naval Command, Control and Ocean Surveillance Center and a research contract from Bell Atlantic.

Contents

List of Figures and Tables

Chapter 1

Introduction

1.1 The seMole framework

This thesis describes *seMole* (*se*mantic *Mole*), a novel framework to retrieve information from the World Wide Web. Previous methods of harvesting HTML data used the underlying HTML tree structure combined with regular expressions. We found this method simple to implement but vulnerable to changes in the HTML structure. Thus, we developed a method of harvesting data that only uses the *content* of the Web page, referred to as *semantic addressing*. Since the content of HTML pages remains relatively constant through time, semantic addressing proves to be a more reliable and simpler way of harvesting data.

The need for this new method becomes evident as companies and research institutions realize the potential of the Web as a large, time-critical database. Unfortunately, two issues prevent organizations from using the Web for this purpose. First, the Web lacks an interface to allow machines to retrieve information from Web pages. HTML, the dominant language for most of today's Web pages, lacks a mechanism to label data on Web pages. (META tags can be used in HTML to accomplish this, but META tags are cumbersome and are rarely used.) HTML's cousin language, XML [1], aims to fix this problem by structuring data on Web pages with semantic tags. XML will allow machines to easily harvest data. Both of these languages, HTML and XML, will coexist for many

years. HTML will be used on Web sites where fast page prototyping is important, whereas XML will be used where data structure is more important than the data's façade. Second, HTML pages are updated on a frequent basis to either revise data or to refresh a Web's page's façade. Many of these updates drastically change the underlying HTML structures that represent these pages. Since current frameworks utilize the underlying HTML structure to mine data, these frameworks are not robust to frequent HTML revisions.

Many commercial applications, such as WebMethod's B2B [2], are available. These products have been developed to meet organizations' need to gather information from HTML and XML pages. However they are still vulnerable to frequent HTML revisions because they rely heavily on the HTML structure to find their data. Thus organizations lack a mechanism that provides both a robust bridge to the Web.

The seMole framework tackles both of these issues by providing a *semantic proxy* to the Web. By using semantic addressing, seMole is independent of all changes in the HTML structure and most changes to data. The seMole framework provides an easy to use interface as well as a robust harvesting system. The seMole framework allows users to develop machine-to-Web bridges quickly without constant upkeep.

1.2 Thesis Overview

In this thesis, we develop two frameworks to test the performance of semantic addressing. The first of these frameworks, called the *seMole* framework (*se*mantic *Mole*) allows users to gather HTML data through semantic addressing. The second is the *absMole* framework (*abs*olute *Mole*), which shares many of the components of seMole but uses absolute addressing instead.

First, we describe the motivation behind the development of these two frameworks. We next describe these two systems in detail with emphasis on how seMole gathers its data.

Then, we evaluate and compare the performance, ease of use, and robustness of these frameworks and WebMethod's B2B product. We find that although seMole is slower, it proves to be more robust and simpler to use than absolute addressing frameworks.

Chapter 2

Background

2.1 Web Galaxy

In 1994, the MIT Spoken Language Systems Group introduced GALAXY, a client-server architecture for accessing on-line information using spoken dialogue [3]. Since then, GALAXY has served as the testbed for our research and development of human language technologies, resulting in systems with expertise in various *domains.* A domain is one type of information such as airline information or weather information. In 1996, we made our first significant architectural redesign and introduced WEBGALAXY, which had a Java GUI to permit universal access via any web browser [4]. While the underlying human language technologies were the same across all domains, the information access/retrieval mechanisms for WEBGALAXY domains were developed separately and therefore lacked several key features:

- *Code re-use*: Because each domain was written with a different computer language, the work used to develop one domain could not be re-used to develop a new domain. Code was largely task-specific and difficult to port from one system to another.
- *Robustness*: All information was gathered using an absolute addressing scheme similar to the addressing scheme of the WIDL language developed by WebMethods [5]. Data was gathered from each Web page by using the absolute HTML position of the data on the page. Therefore, if the page changed even slightly, the data retrieval would likely fail.

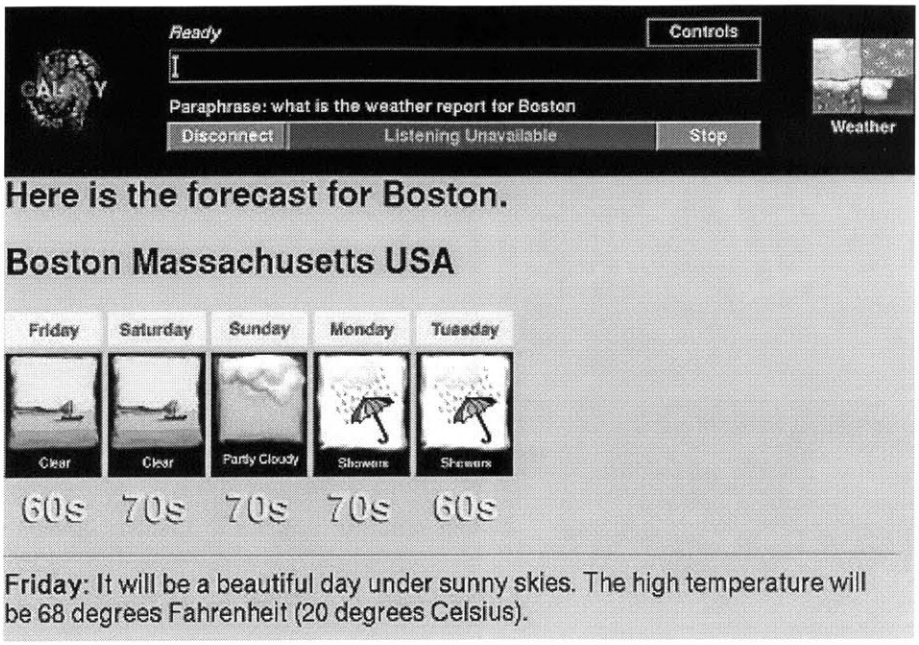


Figure 2-1: WebGalaxy's interface. This screenshot shows WebGalaxy and its subsystem Jupiter, which is responsible for weather information.

- *Timeliness of Data*: Because systems lacked a simple, robust interface to the Web, most Web data was collected and stored in off-line databases (i.e., when a user requested information, our systems would go to a database rather than the on-line source on the Web).

Two of WEBGALAXY's main subsystems are Jupiter and Pegasus. Jupiter is a weather retrieval system that relied on C programs to retrieve data from the CNN Weather site [6] and store them in an off-line database. This system harvested weather information through regular expressions and stored them in an Oracle database. When users requested weather information, Jupiter referred to this off-line database to find information. However, because Jupiter relied only on regular expressions to find its data, it was very prone to fail due to changes in the CNN Weather site. If, for example, the order of the HTML data changed or if the Weather site changed its façade, these C programs failed.

Pegasus is WEBGALAXY's subsystem that provides access to information in spoken or written natural language. One component of this subsystem gather airlines information from various Web sources. This information was largely time-sensitive (e.g., current flight information) and therefore had to be gathered from on-line databases such as CO.O.L. Travel Assistant [7]. Most of the Pegasus' scripts were based on Tcl [8]. Similar to Jupiter's scripts, Pegasus' scripts were based heavily on regular expressions and required the order of the data on the page to remain persistent through time.

Because these two systems were developed separately, much of the code each used to interface to Web pages was redundant. Further, without any centralization of the data retrieval mechanisms, the work to maintain the harvesting of this data was spread over various systems and people.

2.2 AbsMole and Absolute Addressing

We first tackled these issues using a centralized framework called absMole (*abs*olute *Mole*). This system used an absolute addressing scheme similar to WebMethod's B2B product. To harvest information with absMole, users created *absolute templates*: scripts that harvested text on an HTML page through *absolute addressing*. An absolute address is a location of a piece of text on the HTML structure that represents that page. Many of today's commercial Web harvesting packages (e.g., WebMethod's B2B) use an *absolute addressing* scheme to gather information from Web pages. An example absolute address for a hypothetical Web page is "The TEXT element matching the regular expression 'high forecast' under the second table element in the HTML structure":

```
table:2.regex("high forecast")
```

These addresses are very susceptible to small changes in the HTML structure. If, for example, another table were inserted at the beginning of our hypothetical Web page, then the absolute address would be pointing to the wrong data. For absolute addressing schemes to be reliable, the position of all pieces of data in the HTML structure must be

consistent from one version of the page to the next. Unfortunately, since the pages that we target are usually updated often (to update the appearance or to change data on the page), these absolute addressing schemes tend to fail on a regular basis.

Since absMole was server-based, all of these absolute templates resided on the server, allowing code to be re-used amongst all users. Further, templates were modular: scripts can call other scripts. For example, a template that was responsible for gathering information about a ski resort could rely upon a weather template to gather information about the current conditions of that resort. AbsMole was an on-line gateway to the Web. It did not store anything offline (except in its proxy cache which could be turned on or off) and therefore most, if not all, of the data it retrieved was on-line. Although the absMole provided a common interface into the Web, it still lacked robustness because it used an absolute addressing scheme.

2.3 SeMole and Semantic Addressing

SeMole (*se*mantic *Mole*) is an information retrieval framework that addresses many of the problems of absMole and B2B. Since seMole is a server-based framework in which all code resides on the server, it allows code to be re-used by anyone using the seMole framework. For example, a ski information harvester that needs information about the current weather can also invoke a weather harvester that resides on the server.

With seMole, users define *semantic templates*, XML-based scripts that use *semantic addressing* to gather information from a Web page. Unlike absolute addressing, semantic addressing relies on the content of the page to find data rather than the HTML structure by itself. For example, to find information about the Wednesday high temperature forecast on a weather page, a semantic address might be "the integer nearest the word 'high' nearest the word 'Wednesday'":

```
<near>
   "Wednesday";
   <near>
        "high";
        "[0-1]*";
   </near>
</near>
```

This scheme is entirely independent of the HTML structure and only depends on how key/value pairs are placed in the two-dimensional HTML space. Further, notice that no matter how much the Web page changes, the content of the page will remain the same: there will always be a Wednesday high forecast. Therefore, even if the façade of this weather site changes, semantic addressing will still retrieve the correct data.

In the following chapters, we first describe the seMole framework in depth, detailing its various components and the processes it uses to harvest data. Next, we investigate the robustness, speed, and usability of this system when compared to absMole and WebMethod's B2B product.

2.4 Related Work

2.4.1 Commercial Software: WebMethod's B2B

Another framework we considered for WEBGALAXY was WebMethod's B2B framework. This framework is a commercial system with a simple user interface designed to allow users to develop harvesting scripts quickly. B2B offers advanced capabilities to connect to Web sites that use forms, usernames, and cookies. (These capabilities are all described in the WIDL white paper.) Further, B2B have many exception mechanisms to inform users of connect failures, form failures, and other failures related to Web harvesting. B2B offers two methods for developing scripts. Users can use the included graphical interface to point and click at the data they wish to harvest, or they can directly manipulate the text scripts to gather information from HTML pages. If users use the

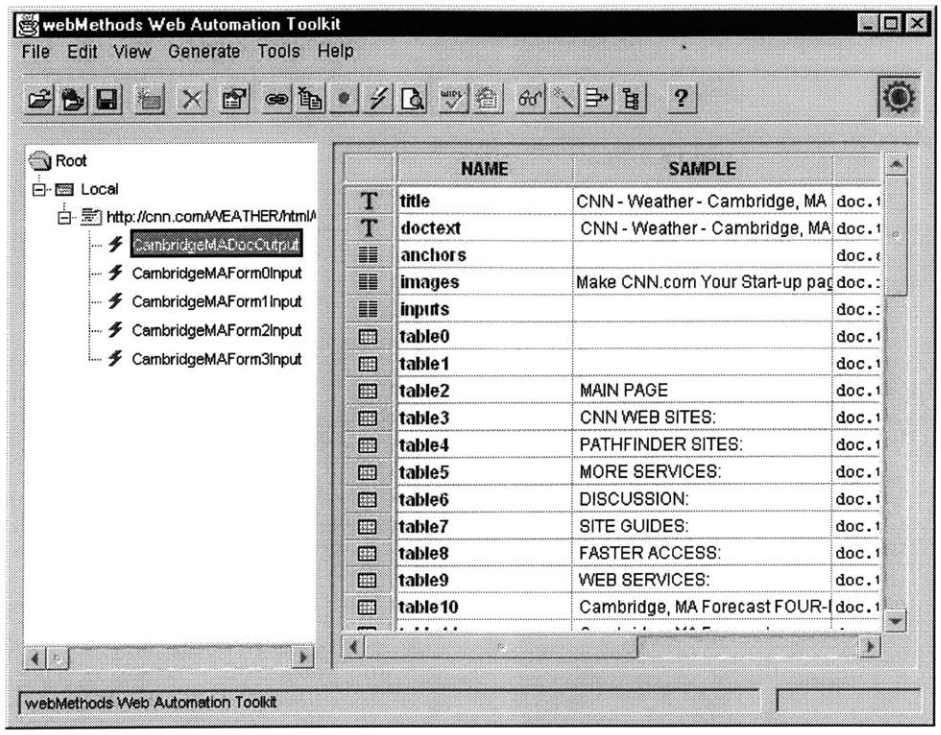


Figure 2-2: WebMethod's B2B interface. Users can easily point and click the data they wish to retrieve from a Web site.

graphical interface (see figure 2-2), the users are presented with a simple menu of all the text elements on the user's target Web page. Once the user chooses the text elements he wishes to harvest, B2B automatically creates the script necessary gather the information. This script is based on the WIDL language, therefore all of the harvesting is based on an absolute addressing similar to absMole's addressing system.

The user can also develop harvesting scripts by directly editing the scripts. Instead of simply choosing the pieces of text he wishes to harvest, by editing the scripts, the user can use regular expressions to locate data or keys that label the data. However, unlike semantic addressing, B2B lacks a mechanism for searching in the rendered space of the HTML page. Like absMole, since B2B is unaware of how HTML tags are rendered on screen, it is restricted to work in the HTML space of the page.

Once a script is complete, it resides on the server-side portion of B2B. This component takes HTML requests for data, allowing any system with access to the Web immediate access to all harvesting templates. Further, responses to the user are also defined in HTML. Responses are user-definable and are dynamically created upon completion of a script.

2.4.2 XML/XSL

One of the main intentions of seMole and absMole is to provide a link between machines and data on a Web page. A new Web language can potentially allow machines to find data on the Web without an intermediary link. This new language, called XML (*E*xtensible *M*arkup *L*anguage), is currently being developed by the Web Consortium. XML requires all text on a Web page to be tagged with semantic labels. This allows anyone (or a machine) who reads a Web page to know what each piece of text represents. A raw XML document is very structured, making it simple for any machine to read and find data.

Unlike HTML, XML is not intended for formatting. Instead, its purpose is to provide for a general language that can be used as a ground work for other languages, including formatting languages. XSL (*E*xtensible *S*tylesheet *L*anguage) [9], a formatting language developed on top of XML is a formatting proposal by the Web Consortium. Together, XML and XSL can potentially redefine how people and machines use the Web. However, since both of these languages have yet to be fully completed and because of HTML current dominance over the Web, it is unlikely that XML and XSL will be pervasively used for some time to come.

Chapter 3

System Description

3.1 Introduction

To evaluate the performance of semantic parsing, we developed two systems: the seMole and the absMole. The seMole is a Java Web Server-based component [10] that uses semantic parsing to not only harvest data from today's HTML-based pages but also future pages developed with XML. It also makes heavy use of XML for its Web interface. The absMole framework is similar to seMole in that it shares most of its components with seMole. However, instead of using semantic addressing, absMole uses absolute addressing to find data on Web pages. The absMole framework's absolute engine and language are derived heavily from WebMethod's WIDL language (used in their B2B product). Both the seMole and the absMole frameworks were developed using Java and Perl.

The seMole and absMole frameworks share three main components: the XML Web parser, the Java Web Server, and the cache. The XML Web parser is responsible for interfacing both frameworks to the Web. The XML Web parser, along with all other components, resides on the Java Web Server. The Java Web Server is a repository for all components. The cache also resides on the Java Web Server. Its main responsibility is to cache requests to and from the seMole framework to reduce latency.

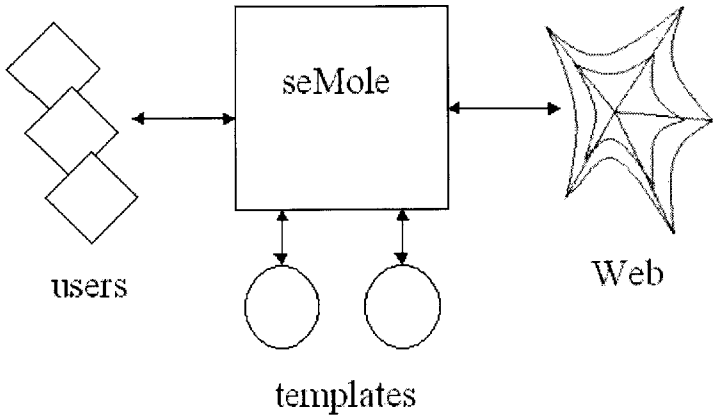


Figure 3-1: Overall view of the seMole framework.

The seMole and absMole frameworks each have two other components that they do not share. The seMole framework has the semantic engine and semantic language. These two components are responsible for interfacing the user to the seMole framework and also for providing semantic addressing. The absMole framework has an absolute engine and an absolute language. These are analogous to seMole's two semantic components, except absMole's two components provide the user with absolute addressing in lieu of semantic addressing.

3.2 The seMole Framework

3.2.1 The XML Web parser

The XML Web parser is seMole's interface into the World Wide Web. Its main responsibility is to retrieve HTML documents from the Web and translate them into well-formed, machine-readable XML documents. The Web parser is developed from Micro-

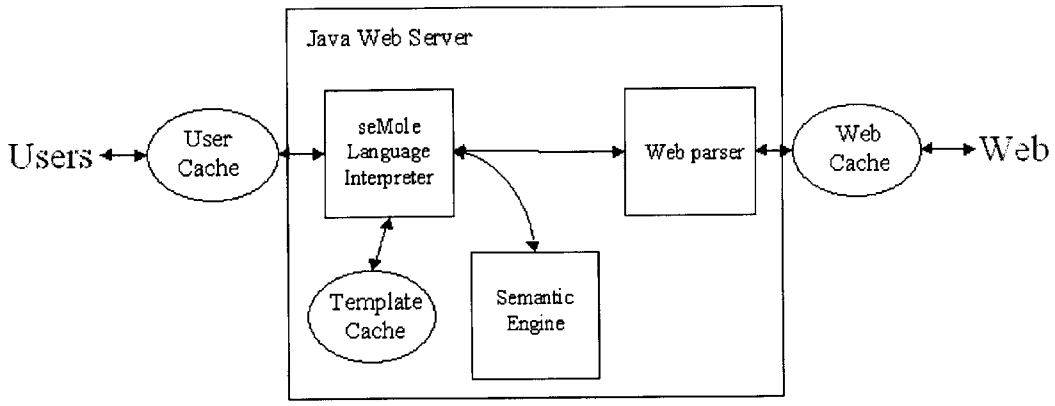


Figure 3-2: The five components of the seMole framework: The Java Web Server, the Web parser, the cache, the seMole language, and the semantic engine.

soft's XML parser [11]. Unfortunately, today's HTML documents do not conform to the Web Consortium's XML conventions that the original XML parser followed. For example, in any XML document, all tags must be closed off with a corresponding close tag. On the other hand, HTML allows a variety of tags to be left "unclosed" (e.g. the break tag in HTML does not require a close tag to indicate a break in the page). Further, in the many cases, HTML documents are simply poorly written, with missing close tags. Even though these documents do not follow HTML conventions, they still are parsed and displayed properly by many popular Web browsers. The original Microsoft XML parser was extended to parse today's often poorly written HTML documents into well-formed XML documents. A library of parse rules was added to the parser to allow it to parse these HTML documents. For example, in the case of missing close tags, it either immediately inserts a close tag after the opening tag (for BR) or it inserts the close tag in the most probable location of the document (in the case of a missing TABLE close tag, the Web parser inserts it after the last row element TR before the next TABLE element in the document).

Once fetched and parsed, the XML document is then readable by any other component of seMole. The XML document is accessible through an interface defined in the *DOM*

(*D*ocument *O*bject *M*odel) [12] model paper by the World Wide Web Consortium. This interface allows each part of the XML document to behave as an object. Each object is related to other objects through a hierarchical tree. Elements of the XML document are accessible through their parents and elements can be rearranged and deleted through a specified set of DOM interface methods. The Web parser heavily depends on this interface to navigate through the XML space to find data.

The Web parser is also responsible for interfacing seMole to a variety of Web data sources that require more than a simple URL request. For example, many Web sites have forms, redirects, and user/password information. Unfortunately, these mechanisms are not completely functional in a multi-user situation. For example, since the seMole is one server that serves many users, it is possible that multiple users need to access different pieces of information from one Web site using the *same username*. This often times confuses the Web site, and it will return garbage data. Later iterations of the seMole will fix this problem (see Section 5.5).

3.2.2 The Java Web Server

The Java Web Server is responsible for holding all of the components (see figure 3-2). All of the other components of the framework reside on the Java Web Server as servlets. Servlets are Java applications that are invoked in much the same manner as cgi-scripts: through a URL. They are fast and easy to prototype. Users of the seMole framework access the Java Web Server through a *dispatcher* servlet. This servlet is responsible for parsing an incoming request for a template (with arguments) and then executing the template. Once completed, the dispatcher then send back the result of that template back to the user.

The Java Web Server can also be daisy-chained with an unlimited number of other Java Web Servers to accommodate more capacity or to organize servers by semantic content. For example, we could organize seMole servers by categories such as "weather information", "ski information", and "restaurant information". Three separate servers can exist

for each one of these categories and each server will be aware of the other two servers. If one of these servers receives a request of a certain type of data that it does not know about (e.g. the "weather information" server receives a request for a "ski information" template), then it can route the request to the correct server. This routing ability allows servers to be coherently organized and therefore, load can be distributed amongst a group of servers without mirroring templates or content between different servers.

3.2.3 The Cache

The cache is the component of the seMole framework that is responsible for making the seMole fast. There are three main components of the seMole cache: the user cache, the template cache, and the Web cache (see figure 3-2). The user cache resides between the users and Java Web Server. This cache intercepts all template requests. If this template catches a template request that has been executed recently, the user cache will be quickly returned the cached copy. The second cache is the template cache. This cache resides in the seMole and caches all the results of the template that are executed from within the seMole (i.e., those templates that are invoked by other templates). The final cache is the Web cache. This cache resides in between the Web parser and the Web. Since Web latency is very high when compared to template execution time (see section 4.4 Performance), this cache is heavily relied on to cache pages as long as possible.

Although some Web sites do not change their content frequently, many Web sites are updated on a regular basis. To differentiate between these two types of Web sites, a cache policy manager is used to keep track of how long to store a cached version of an HTML page and its corresponding template execution results. This manager keeps a user-defined policy for each Web site that the seMole harvests from.

3.2.4 The SeMole Language

The seMole framework is entirely controlled by small scripts called templates. Each template represents one *datum* of information. A datum of information is one piece or a

```
<third_command>
   <first_command>
        first_arg1;
        first_arg2;
   </first_command>
   <second_command>
        second_arg1;
   </second_command>
   third_arg3;
</third_command>
```

Figure 3-3: Execution flows from the bottom of the XML tree hierarchy upward. The "`first_command`" (which has two arguments) is executed first and then the "`second_command`" is executed next (it has one argument). The last command to be executed is "`third_command`". It has three arguments: one is a result of "`first_command`", another is a result of the "`second_command`", and one is a text argument.

group of data that can be identified with one semantic label. For example, in a Web page giving the weather forecast for a certain city, a group of numbers under the heading Tuesday may be considered one datum of information with the semantic label "*day forecast*". Further, the entire page can also be considered one datum of information with the semantic label "*city forecast*".

These templates are defined in a language derived from XML called the seMole language. Each tag in the template represents a command that requests an action from some component of seMole. For example, a FETCH command asks the Web parser to request a new page from the Web and parse it into a valid XML document. Each command usually requires a number of arguments (e.g., the FETCH command requires a URL text argument). These arguments are the children of the command. These arguments can be a raw string, or they can be other commands. Execution flows from the bottom of the XML hierarchy upward. Commands on the same level of the XML hierarchy are executed top to bottom. Therefore, children of commands are executed first and their results are passed as arguments to the parents (see figure 3-3). The only exception is the topmost parent of the XML template. The topmost parent tag is the name of the script and is not executed. Once its children are executed, their results are then added as children of this top most parent. Then, the top most parent and its sub-tree are returned to the agent

```
<template_one>
  arg1;
  arg2;
  arg3;
</template_one>
```

Figure 3-4: Invoking another template from within a template. The command name ("`template_one`") is simply the name of the template.

who invoked the template. In this way, the XML template also defines the structure of the template result.

Templates can also invoke other templates. Templates are called in much the same manner as commands. A tag with the template's name is used, with arguments that the template expects (see figure 3-4). Template tags are first assumed to be command names. If seMole cannot find the command, then it assumes that the tag represents a template call.

In the seMole language, there is no semantic mechanism for w2understanding paragraph text and sorting it into small pieces of data. In most cases, the smallest item of data that the user can access is one element of the HTML structure (e.g., a TEXT element, or a TABLE is one element of an HTML structure). Therefore, if there is a large amount of text on a Web page that falls underneath one TEXT element, the seMole can only retrieve all of that text at once. This problem falls under the domain of natural language understanding, which we do not approach in this thesis.

The seMole language offers three types of commands. The first type of commands is *structural commands*. These commands largely affect how the elements of the template tree. For example, the SET command allows users to insert an element within the template tree (figure 3-5). This allows users to organize results of other commands within the template tree. The arguments to the SET command are other commands. Once the semantic engine executes these arguments, the results are placed as children underneath the set command and returned to the user.

```
;; Template                    ;; Template Result

<set forecasts>               <forecasts>
     <near>                        <near_result>
     .                             .
     .                             .
     </near>          →            </near_result>
     <near>                        <near_result>
     .                             .
     .                             .
     </near>                       </near_result>
</set>                        </forecast>
```

Figure 3-5: The execution of a SET command and its result.

The next type of commands is *retrieval commands*. These commands are responsible for retrieving information from the Web. The most important of these commands is the FETCH command. This command is the seMole's language interface into the Web parser. It is responsible for fetching an HTML document and then parsing it into a machine-readable form.

The last type of seMole commands is *semantic commands*. These commands make up the bulk of the seMole language. These commands are the user's interface to the semantic engine. The most important of these commands is the NEAR command. The NEAR command allows users to find semantic key/data pairs in the HTML hierarchies retrieved by the FETCH command.

Since each template is responsible for one datum and since templates can invoke each other, most Web sites are harvested with a group of hierarchically organized templates. In our weather forecast example, one template may be responsible for gathering a day forecast, while another template uses this smaller template to compile all the day forecasts on that page.

Variable naming is also included in the seMole language. At any point in the execution of the template, users can access any other part of the XML template through an absolute

```
<template_one>
      <set template_argument_one>
            "http://www.mit.edu";
      </set>
      <set template_argument_two>
            "Cambridge, MA";
      </set>
      <fetch>
            *template_one.template_argument_one;
      </fetch>
</template_one>
```

Figure 3-6: The first argument to the FETCH command is an absolute address to the first argument to this template: "http://www.mit.edu"

address. Usually, users will access a part of the tree that is above the point of execution since the commands that follow that point of execution have yet to be run.

These templates are all written with a standard text editor and are stored on the server-side component of the seMole. This allows all templates to be shared amongst all users.

3.2.5 The Semantic engine

The semantic engine is the heart of seMole. It is responsible for robustly finding data in HTML/XML structures through semantic addressing. It heavily relies on *semantic keys* to find its data. Semantic keys are any pieces of text that labels a piece or group of data. For example, on a web page with airline flight information, text that labels arrival times, gate number, and the flight number are all semantic keys.

The user uses the semantic engine through a number of semantic commands in the seMole language. These include NEAR, AFTERNEAR, and BEFORENEAR. Most semantic commands are derived from the command NEAR. This command takes, as arguments, a regular expression for a semantic key and any number of regular expressions for data. The key and the data are assumed to relatively near each other in the rendered version of the HTML structure. Once invoked, this command attempts to find the best match to this key/data regular expression pair. First, it searches the entire document for

all text elements that match the semantic key regular expression. Once this list has been compiled, the NEAR command then attempts a breadth-first search in the local *rendered space* of each of the semantic key matches for the data regular expression. The rendered space is the two-dimensional area that is viewed by a user when he renders the HTML structure with a Web browser. This differs from the *HTML space* that is a tree structure that does not include information about how elements should be placed on screen. The semantic engine knows how the HTML elements are placed in the rendered space by interpreting the tags in the HTML structure. For example, by evaluating the TR and TD elements of a TABLE element, the semantic engine is able to interpret how text should be aligned with each other and approximately what distance they are apart. The semantic engine assumes that most key/data pairs are in cardinal directions (e.g., north, south, east, and west) from each other. In HTML space, the four cardinal directions are toward the top of the screen (north), the bottom of the screen (south), the left of the screen (west), and the right of the screen (east). (Key/data pairs that are not in cardinal directions of each other in TABLE elements are very rare.) Therefore, this NEAR search in the rendered space of the HTML structure only occurs in cardinal directions from the semantic key.

Once found, the key/data pairs are ranked to find the best match to the regular expression pair. We use a ranking because we are not guaranteed that our regular expression pairs will return correct data: the regular expression may also match false positive data on the HTML page. The semantic engine ranks the possible pairs by a number of heuristics derived from patterns in key/data pairs on HTML pages. First, the semantic engine considers the distance between the key/value pairs. The shorter the distance, the better the match. Second, the semantic engine considers the *formatting* of the pairs. Formatting refers to the font, size, boldness, etc. of the key and data. Given two pairs with equal HTML space distance, that pair with unequal formatting between the key and data is ranked higher. Most keys have *higher formatting* than values: either keys are bolded, italicized, or they have a larger font size than their data. Keys/data pairs with high formatting have higher rankings.

```
<NEAR>
      "m/HIGH/i";
      "m/[0-9]*/i";
</NEAR>
```

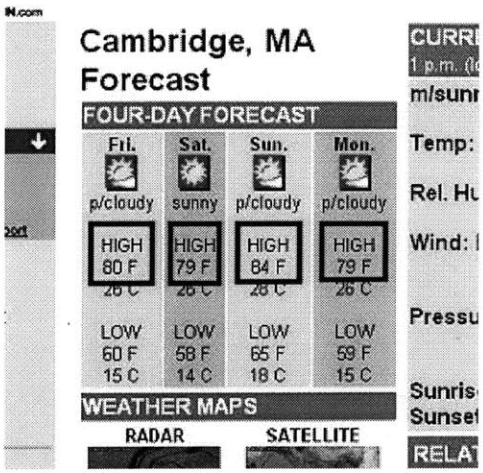


Figure 3-7: Semantic groups on the CNN weather forecast page. The key value pairs that match the NEAR command are outline in rectangles.

Finally, if a set of key/value pairs have the same distance and formatting, *semantic groups* are assumed. A semantic group is a group of key/data pairs that all match a template equally well. Therefore all are assumed to be of the same rank and all are returned by the NEAR command. For example, in figure 3-7, we attempt to find the key data pair "HIGH"/"[0-9]*". Four pairs match this template, all with equal distance and formatting. Therefore, all returned to the user as four separate semantic groups.

Semantic groups can also be hierarchical. For example, in figure 3-8, we have nested groups of NEAR commands for the same page presented in figure 3-7. Each of the two children NEAR commands will return at least four semantic groups each (there are four day forecasts on the page, each with a high and low temperature forecast). These two sets of semantic groups are returned to the parent NEAR command which then sorts the two set of semantic groups into *another* set of parent semantic groups. Each one of these parent semantic groups will contain two children: one "HIGH" forecast semantic group, and one "LOW" semantic group. These children semantic groups are paired up by a distance measure: a *center of mass* is calculated for each of the "HIGH" and "LOW" semantic groups and pairs are chosen by distance. The center of mass of a semantic group is the point on the rendered screen that is in the middle of all the key/data text elements

```
<NEAR>
      <NEAR>
            "m/HIGH/I";
            "m/[0-9]*/i";
      </NEAR>
      <NEAR>
            "m/LOW/i";
            "m/[0-9]*/i";
      </NEAR>
</NEAR>
```

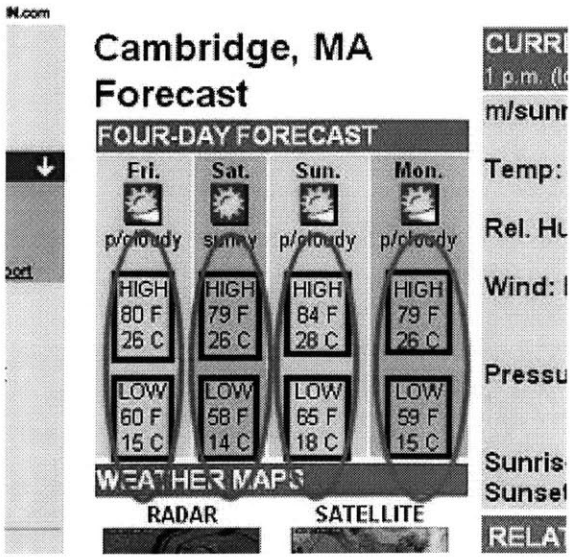


Figure 3-8: Nested semantic groups within a weather forecast page. The pairs that match the two children NEAR commands are outlined in rectangles. The semantic groups are outlined in ellipses.

found. Once the parent NEAR command is completed, at least four semantic groups are returned (there are *at least* four returned because there may potentially be some false positive matches).

3.3 The AbsMole Framework

AbsMole is an absolute addressing framework that we developed to test seMole's performance. It shares all the components of seMole, except for the semantic engine and the semantic language. In the semantic engine's place is an *absolute engine* modeled after the WIDL language developed by WebMethods.

As specified in the WIDL whitepaper, WIDL uses a mix of regular expression matching and absolute addressing to find data. The absMole language only allows for regular expression matching and absolute addressing. Although the language may differ from WIDL, the fundamental mechanism for finding data remains the same. The absMole's language also differs from the seMole language. Instead of a bottom-up hierarchical lan-

guage, the absMole language is a traditional top-down scripting language. Many of the structural and retrieval commands are the same, but in place of the semantic commands are commands to address the HTML hierarchy through absolute addressing and regular expressions.

3.4 Summary

The seMole is a robust framework for harvesting information from the Web. The seMole offers three main features: ease of use, speed, and robustness. The seMole language and the seMole's web parser allow users to quickly develop and easily maintain templates to harvest various HTML pages through the Web. The Java Web Server and the cache allow users to quickly harvest information once their templates are on-line. Finally, the semantic engine allows templates to gather HTML information through semantic addressing. Semantic addressing, in turn, makes templates robust to many changes in HTML pages, reducing the amount of time users spend in maintaining seMole templates. In later chapters, we compare the seMole framework to the absMole framework (which is the same as the seMole framework except it uses absolute addressing in lieu of semantic addressing) and WebMethod's B2B to test for its ease of use, speed, and robustness.

Chapter 4

Results

4.1 Introduction

In this section we evaluate the performance of seMole for its ease of use, robustness, and speed. We compare the seMole framework to the absMole framework and WebMethod's B2B product. These two other frameworks both use absolute addressing schemes to gather information from the Web.

First, we compared seMole's robustness to absMole's robustness by creating templates for various Web sites. Next, we completed an interface study that compared the length of time it required to developed scripts/templates for seMole, absMole, and B2B. Finally, we compared seMole's, absMole's, and B2B's performance characteristics.

4.2 Robustness

To measure the robustness of seMole's semantic engine, we compared it to absMole's absolute engine. We did not include B2B in our study because its basic mechanism for retrieving data was mirrored by absMole. We created seMole and absMole templates for various Web sites. A Web site refers to any source of on-line information that presents HTML data about one domain (e.g., stock information). Each Web site that we considered had more than one page of information, with each page describing its own target

(e.g., Microsoft stock, or Intel stock). Each of these target pages is called a frame of data. Each frame was not necessarily static. Many pages, such as those that presented weather information, were updated on a frequent basis.

Each Web site was categorized in terms of topology (persistent or transient) and data (persistent or transient). "Topology" refers to how the data is presented in a frame. If a site has persistent topology, then the data is in the same place across all frames on the Web site. If the data is located in different places from one frame to the next, the topology is said to be transient. "Data" refers to the amount of data present on the frame. When the amount of data is the same across all frames, the site is said to have persistent data. If there is a discrepancy in the amount of data from one frame to the next (the key, value, or both may be missing), the site is said to have transient data. While a Web site's persistence and transience refer to the arrangement and the amount of data across frames at one point in time, a Web site's *temporal* nature refers to how that Web site changes over a period of time. For example, on-line databases about restaurants may seem persistent, but over a period of a few months or years, restaurants close and open and therefore, their entries in the database change. This type of change is a *temporal* change and is usually difficult to overcome because of its unpredictability.

4.2.1 CNN Weather Site: Persistent Topology, Persistent Data

One of the first sites that we considered was the CNN Weather Site. This site has over 6,100 frames of data for various cities worldwide. Each frame included four day forecasts as well as information about the current condition for its city. These frames largely have persistent topology as well as persistent data. We attempted to gather weather data using seMole and absMole.

Almost all of the interesting data on the CNN Weather site were labeled with semantic keys. For example, every day forecast was labeled with a name of a day (see figure 4-1) and each temperature forecast was labeled with the semantic key "HIGH" or "LOW".

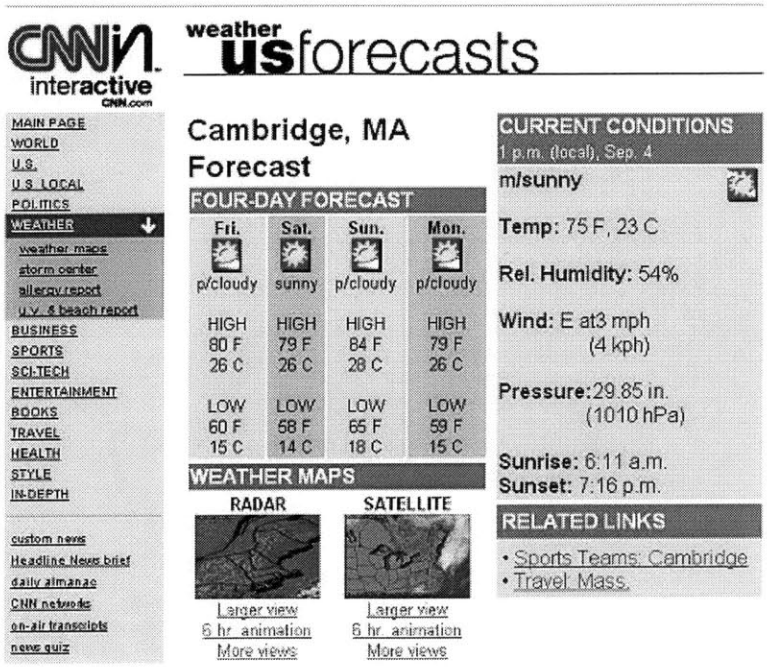


Figure 4-1: A frame from the CNN Weather site. This page presents four day forecasts as well as information about current conditions for over 6,100 cities throughout the world.

Our seMole template was therefore very straightforward. First, we created a sub-template for one of the day forecasts. This consisted of retrieving the day name, the "HIGH" for that day and the "LOW" forecast for that day. This sub-template was then applied across the entire frame using semantic groups to fetch all of the day forecasts. Then, NEAR commands were used to fetch data about the current condition, humidity, and wind. The AbsMole template was similarly straightforward. Each day forecast on CNN Weather were represented by one table. Therefore, for each of the day forecasts, absMole template extracted one table from the HTML structure. Each of these trees was then passed to a sub-template to filter out the target data. Care was taken while forming

the absolute addresses to the day forecasts. Instead of directly accessing each of the tables directly:

```
table:10  ;; first forecast
table:11  ;; second forecast
table:12  ;; third forecast
table:13  ;; fourth forecast
```

These tables were accessed through a master table found with a regular expression:

```
regex("day forecast").table:1  ;; first forecast
regex("day forecast").table:2  ;; second forecast
regex("day forecast").table:3  ;; third forecast
regex("day forecast").table:4  ;; fourth forecast
```

Therefore, even if another table were inserted before the master table this longer addressing scheme would still retrieve the correct data. Data regarding the current condition were retrieved in a similar manner.

Because the topology and data were persistent, both seMole and absMole consistently gathered the same data. The only discrepancy between the two frameworks occurred when the CNN Weather Web site changed its façade in March of 1998. This temporal change involved changes only to the topology of the site. All of the weather information persisted through the change. However, banners were added and tables were rearranged to update its appearance. Tables were inserted before and inside the master table. Although the tables before the master table did not effect absMole template, the tables within the master table renumbered the day forecast tables within the master table. The addresses had to be modified to:

```
regex("day forecast").table:2  ;; first forecast
regex("day forecast").table:3  ;; second forecast
regex("day forecast").table:4  ;; third forecast
regex("day forecast").table:5  ;; fourth forecast
```

Other similar changes rendered the absolute addresses to the current condition data. Because of this, absMole's template failed entirely.

Although absMole's template failed, seMole's template continued to properly gather information from the site. This is largely because all of the changes to the frames were cosmetic. The semantic keys and the data still remained. Although they now had different fonts and typing, keys such as "HIGH", "LOW", and "WIND", still stayed relatively near the data that they labeled.

4.2.2 Cool Travel Assistant: Transient Topology, Persistent Data

Another Web site that we gathered data from is CO.O.L. Travel Assistant. This site provided flight information for various airlines. Each frame presented on this site held information about one flight (see figure 4-2). Unfortunately, this site had transient topology: each of these frames contained a variable number of legs for each flight (each flight had at least one leg). For each leg of the flight, there existed one table describing where the flight originated, ended, its estimated flight time, and its current status. Data was persistent within each leg, allowing us to predict what kind of data each leg had. Unfortunately, there was no information on each frame to determine how many legs to expect on each frame. This information had to be determined automatically.

We first attempted to gather data from this site using absMole. Because each leg of each flight was presented using the same table structure, we were able to define a simple subroutine template to gather information from one leg. However, to apply this subroutine to a variable number of legs, we needed to rely on the master table encapsulating the legs. From this table, we directly count the number of legs presented on the frame, and applied the subroutine to each leg's table. Unfortunately, this approach is vulnerable to many possible inconsistencies of the frames. For example, if banners or

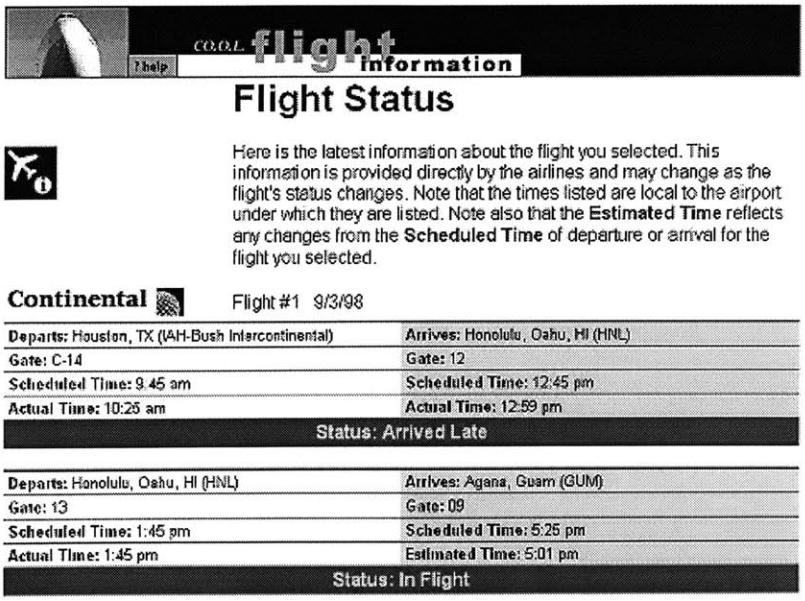


Figure 4-2: A frame from the Cool Travel Assistant site. A flight schedule for Continental flight #1 with two legs.

advertisements were added between the legs, this template would incorrectly determine the number of legs per flight. Further, we relied on the fact that one HTML node is a parent of the tables representing each leg. Therefore, if the parent node moved in the frame, or if each leg were presented using more than one table, this template would again fail.

We then gathered the same information using seMole. As with absMole, we first created a simple "subroutine" template to gather information from one leg on the frame. Like other seMole templates, this subroutine template used semantic tags to find airline information. Therefore, even if the information for one leg of a flight were spread over more than one table, this subroutine would still be able to properly group the leg's data. To capture all the legs within the frame, we took advantage of seMole's semantic grouping ability. Instead of explicitly finding the number of frames by looking at the HTML structure, we simply created another template that applied the subroutine template across the entire frame. Since all the information for one leg was always grouped together, seMole was able to differentiate between two different legs and independently apply the subroutine template to each leg. This was accomplished without the aid of the HTML

substructure. Since this template was created without any dependence on the HTML structure, it was very robust to any changes in the topology of the frames.

4.2.3 Boston Sidewalk: Persistent Topology, Transient Data

Another domain that we attempted to harvest is restaurant information. For this domain, we used the Boston Sidewalk site [13]. This site contains a repository of restaurants in the Boston area. The frames are organized under various categories (Chinese, Italian, etc.) and provide information about hours, payment options, parking availability, as well as reviews by Boston Sidewalk. Although the frames are topologically similar, not all the data is available for all restaurants. For instance, some frames lacked information about parking and therefore lacked a key/value pair for parking. Further, the Sidewalk review on each page lacked a key, making it difficult to locate.

Again, we first attempted to harvest each frame using absMole. The sidewalk review was always located in a certain location within the HTML structure, so retrieving the review was straightforward. However, retrieving the transient data (e.g., payment options and parking) proved to be more problematic.

To tackle this problem, we used several conditional statements to search for the transient data. Unfortunately, these conditional statements, as well as the statements used to retrieve the review, depended heavily on the structure of the HTML. The conditional statements queried a certain portion of the HTML structure. If the HTML structure changed even slightly, the queries would return garbage data.

Retrieving the transient data proved to be simpler with seMole. Since seMole only returns data that match the template, we created a strict template that never returned false data. For example, for parking availability, we queried the entire frame for the key "parking" followed by the regular expression "yes|no|none|street|garage". When the data was not present, this key/value pair was so strict that no false matches were returned.

ERROR: ioerror
OFFENDING COMMAND: image

STACK:

-dictionary-

Figure 4-3: A frame from the Boston Sidewalk site. This page shows information about a restaurant, including its costs, location, hours, and a review.

However, since there was no semantic key labeling the review, fetching the review from the frame proved to be difficult with semantic addressing. Since we lacked a generic solution to find the review on the page, we used an absolute addressing approach and fetched a certain part of the HTML structure that we expected to be the review.

4.3 Interface

No matter how robust a Web harvesting framework may be, it must be simple enough for user to be able to prototype templates quickly. We compare seMole's, absMole's, and B2B's ease of use by observing users prototyping templates from two Web services: CNN Weather and Boston Sidewalk.

SeMole and absMole are similar in that they require users to create scripts through text editors. Writing these scripts require the user to have prior knowledge of the Web site's

	seMole		absMole		B2B	
	Time to Prototype (hrs)	Template length (lines)	Time to Prototype (hrs)	Template length (lines)	Time to Prototype (hrs)	Template length (lines)
CNN Weather	~0.75	127	~1.5	151	~0.5	N/A
Boston Side-walk	~1	110	~1	157	~0.75	N/A

Table 4-1: A comparison between seMole, absMole, and B2B for ease of use. Note that B2B's template length is not included since the WIDL scripts generated by B2B include a large amount of information not used directly for finding data (e.g., information about forms embedded in the page).

frames, including the relative position of key/value pairs as well as the location of potentially false positive data (i.e., data that can be confused with the real target data).

Further, with absMole, writing the script also requires familiarity with the frame's HTML structure. While writing absMole templates, it is often the case that users refer back to the HTML structure many times to find the target data. With seMole, however, no knowledge of the HTML structure is needed since seMole commands only rely the text that is visible on a rendered version of the frame. Therefore, seMole templates can be prototyped much faster than absMole templates (see table 4-1).

In the case where the user is attempting to create a template that will harvest data across templates that have transient topology or data, seMole has proven to be the superior framework. For example, in creating templates for the CO.O.L. Travel Assistant site, users were able to use seMole's semantic grouping ability to easily harvest groups of data from each frame, no matter the number of legs. However, with the absMole, users had to use counters and looping constructs and counters to find each leg. (Copies of seMole and absMole templates for the CNN Weather site can be found in the Appendix.)

B2B, on the other hand, automates many of the processes that absMole requires the user to do. B2B offers a complete graphical front end that automates the script building process. Users are only burdened with clicking on the data that they want. They usually do not need to refer back to the HTML frame or the HTML structure. Therefore, users were

able to prototype scripts much faster with B2B than with either seMole or absMole (see table 4.1). However, an interface similar to B2B's interface would be inappropriate for the seMole. Many of the semantic commands offered by the seMole language require the user to create regular expressions to find the semantic keys and their respective values. It is unclear as to how a system would automatically create regular expressions for the seMole.

4.4 Performance

The last issue that we considered was how fast these frameworks were able to parse Web frames and collect target data. AbsMole and B2B use the same addressing scheme, so we expect that these two systems will have similar execution times. The seMole framework, however, processes many more operations per item of data than absMole or B2B. This is largely because of the fact that seMole must do many more regular expression searches per data fetch. We consider only the CNN Weather Site to compare the performance of our three frameworks. This site has large number of table structures, semantic keys, and semantic groups. Therefore, our three framework's performance on CNN should be representative of their performance on other Web sites.

We take the total running time for each framework and we separate it into two times: 1) the total time for the framework to fetch and parse the HTML frame into an HTML structure and 2) the total time to execute the template on the HTML structure. Seek and parse time includes sending an HTML request to the CNN server, waiting for its reply, and then parsing the reply through the Web parser into a machine-readable HTML structure. In large part, the amount of time to seek and parse a frame is consistent for all the frameworks since this mostly dependent on the status of the CNN server and Internet traffic. However, the time to actually execute a template on a frame is distinctly higher for seMole than for absMole. Further, if we postulate that B2B's average seek time is

	seMole	absMole	B2B
Avg. Web frame seek and parse	1.5 sec	1.5 sec	N/A
Avg. template execution	4.0 sec	3.5 sec	N/A
Avg. total	5.5 sec	5.0	4.3 sec
Avg. time per operation (1 data fetch)	0.1 sec	< 0.1 sec	N/A

Table 4-2: A comparison between seMole's, absMole's, and B2B's performance in harvesting data from the CNN Weather Site. SeMole's operation time is averaged over all GREP and NEAR operations. AbsMole's operation time is averaged over all absolute address executions. The total time for B2B could not be separated because we do not have direct access to their HTML parser or their absolute engine.

similar to the other two, then, the average template execution time for B2B is also distinctly lower than seMole's time (4.3 - 1.5 = 2.8 sec, note that this is an approximate figure since we do not have access to B2B's Web parser).

The reason why template execution time is higher for seMole is that seMole commands require many more HTML walkthroughs than absMole. Without an absolute address to seek its data, seMole must do a complete walk of the HTML hierarchy to find its keys and values. For example, to find the current humidity on the CNN Weather site, seMole must first do a complete tree search for all the text elements that match the semantic key regular expression. Once a list of potential semantic keys is found, it must complete a local search around each of the candidate keys for a text element that matches the value. In comparison, absolute addressing requires very few regular expression searches (they are only done if the absolute address includes a regular expression). Further, these

regular expression searches are only executed on a sub-tree of the HTML hierarchy. For example, given the address:

```
table:1.regex("HIGH")
```

the regular expression for the key "HIGH" is only executed on the tree underneath the first table element. Further, seMole framework has a larger space requirement than absMole. This is because for each data fetch, seMole's semantic engine must keep record of multiple contexts within the HTML structure. This is especially true when we have multiple false positive matches to semantic keys. In comparison, for each data fetch with absMole, the absolute engine only needs to keep track of one context: the path in the HTML structure traced by the absolute address.

Although these measurements are accurate representations of seMole's, absMole's, and B2B's performance, they do not completely measure their performance when they are actually deployed. This is because all of the caches usually installed in the seMole and absMole were turned off for this experiment. When installed in a system, the seMole and absMole would have their front-end and mid-level caches installed, making their response time negligible or much smaller than those listed in table 4.1.

4.5 Summary

Our experiments have shown that seMole is able to more robustly harvest data from various Web sites. Our efforts to harvest data from the CNN Weather site, CO.O.L. Travel Assistant, and Boston Sidewalk have shown that seMole is a superior framework to use when faced with transient topology or transient data. However, seMole is not superior to absMole and B2B for all frames. For example, it is sometimes simpler and faster to use

absMole and B2B where a site is guaranteed to be static through time or when a site lacks semantic keys. Fortunately, most Web sites that present interesting information have a large number of semantic keys. Web sites that do not have many semantic keys usually lack visual/HTML structure and therefore will be hard to harvest with any one of these systems.

Chapter 5

Future Work

5.1 Introduction

Although the seMole framework is a complete system, there is still room for improvement on many of its components. These additions to the seMole framework have yet to be implement because of lack of time or because their additions would require a large architectural change.

5.2 The Rendering Engine

The seMole's semantic engine was originally designed to be compatible with today's HTML pages. Therefore, all of the HTML formatting rules that the seMole needs to find semantic keys were integrated into the semantic engine (i.e., the "NEAR" command understands many of the formatting tags of HTML, allowing it to know how items are aligned in HTML space). However, if we wish to be compatible with future formatting standards, we could create an abstraction layer between the semantic engine and how the page is rendered on screen.

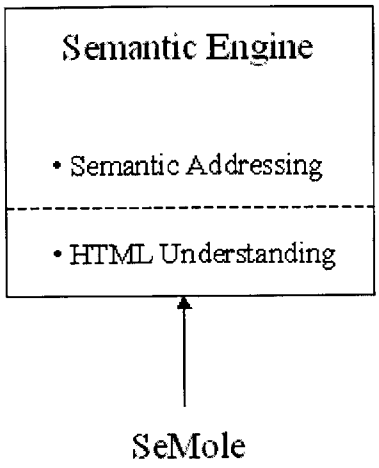


Figure 5.1: The current implementation of the semantic engine. HTML specific code is hard coded into the semantic engine to find semantically labeled information on Web pages.

In this new implementation of the semantic engine in figure 5.1, the semantic engine will have no HTML specific code. Instead, it will rely on a new component, the *rendering engine*, to understand how information is laid out on the screen. This rendering engine will be modular. There can be one rendering engine for HTML, another for MathML [14], and any other formatting language developed in the future. The rendering engine will provide an interface to allow the semantic engine to understand the physical relationship between the semantic components of a Web page. Since future pages will likely be formatted with style sheets (such as XSL), text on Web pages will no longer be only aligned in cardinal directions (i.e., north, south, east, and west). This is because style sheets allow a greater freedom to the Web page designer: it allows them to position text by pixels and it also allows text to overlap. Therefore, the interface in between the rendering engine and the semantic engine will likely be based on the pixel location of data rather than cardinal directions (which the semantic engine currently relies on).

Further, since many Web sites have a large amount of text in images, future iterations of seMole must be able to read text in images. Our rendering engine should include this capability.

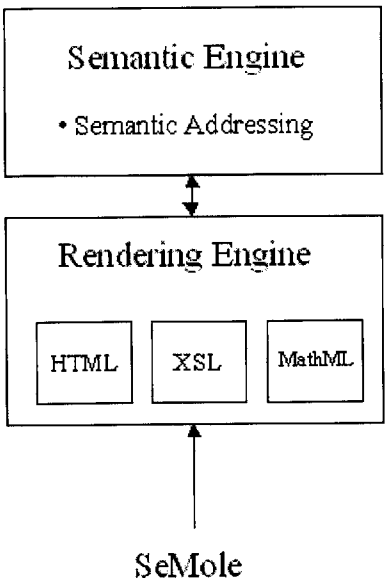


Figure 5.2: The new implementation of the semantic engine with the rendering engine. Web formatting code is hard coded into the rendering engine and the semantic engine is abstracted away from Web languages.

5.3 The Semantic Language

The semantic engine has proven to be much more robust than other mechanisms currently used by commercial systems. However, it fails to approach commercial frameworks' ease of use. In an effort to make the seMole framework easier to use, regular expressions should be eventually phased out of future iterations of the seMole language. In its place will be *semantic types*: broadly labeled groups of regular expressions that the user can use to identify semantic keys and data. For example, to find day forecast semantic keys on the CNN Weather site, one would simply use the semantic type "DAY" (which includes a

```
<try>
     <near>
          arg1;
          arg2;
     </near>
<catch bad_argument_exception>
     ..
</catch>
<catch exception>
     ..
</catch>
```

Figure 5.3: An example try-catch mechanism to catch exceptions for the NEAR command. The first catch statement catches incorrect arguments passed to the NEAR command, and the second catch statement catch all other errors.

large library of pre-defined regular expressions) instead of finding it with the regular expression:

"MON | TUE | WED | THUR | SAT | SUN"

If the seMole framework had such a library of semantic types, future template development will be even quicker and simpler.

The seMole language also lacks a mechanism for detecting and returning exceptions for various retrieval failures. Currently, whenever a template or command fails in its retrieval, it returns a NULL result. If, for example, a NEAR command fails within a template, nothing is returned for that command and the template will continue its execution as if the command were never executed. If all commands within a template fail, an empty template result is returned to the user. The seMole language should incorporate a sophisticated method of catching and returning exceptions. A try-catch construction (similar to Java and C++) could provide this functionality (see figure 5.3).

Some of the more important exceptions that the seMole cannot currently catch are network exceptions, web server exceptions, and time-out exceptions. The seMole framework should incorporate rules for catching these exceptions and reporting them to the user.

5.4 The Web parser and the Cache

Although the Web parser can handle all Web requests from single users simultaneously, it cannot handle multi-users requests for Web sites that use cookies. This is because some Web sites, such as MapQuest [15], keep track of what to return to the Web parser through cookies. Therefore, when the Web parser handles two simultaneous requests from MapQuest, the responses can be confused between the two users. For example, if two users simultaneously request different maps from MapQuest, the MapQuest server will assign *one username* to each of the two incoming requests since the two requests came from the same Web agent: the address of the Web parser. Although the two initial requests will return proper results (each of the two users will receive two different maps), any requests thereafter will be garbled. If, for example, $user_1$ (the user whose request reached the MapQuest server first) then requested a blown-up version of the map image that he sees on his page, he will receive a blown-up version of the map that $user_2$ sees. This is because the MapQuest server assumes that the requested image is the one that the user (Web parser) currently sees, that is, the $user_2$'s image.

This problem can be circumvented in a number of ways. First, the Web parser can serialize template requests that go to a Web site that uses cookies. Although simple, this solution also increases the latency for each template requests by a large amount. Another solution is to expand the Web parser to use multiple IP addresses and multiple cookies at the same time. Therefore, whenever two or more requests comes to the Web parser for the same piece of information, the Web parser will fetch these pages through multiple sockets, avoiding any confusion as to which cookie belongs to each request.

This solution requires the Web parser to start taking over some of the roles of the seMole cache: it has to expand to fetch image data instead of simply returning HTML text back to the user (as it does now). This is because the Web parser will be the only agent that will be aware of which image URL request belongs to which cookie. If the user simply

used the raw IMAGE ALT tags that it receives from the Web parser to retrieve images, the user will get corrupted data from MapQuest. This is because the user will then be retrieving data through an IP that MapQuest has yet to see.

5.5 Semantic Abstraction to the World Wide Web

As the seMole framework is used by more users, it will eventually develop a large library of templates. With such a large library of templates, the seMole framework has the potential to abstract away information retrieval from the user entirely. The user will only be burdened with asking the seMole framework for retrieving a certain type of data, without having to write and debug templates. This is, in large part, is due to the fact that seMole templates are generic enough to apply to a large amount of Web sites. For example, the template designed for the CNN Weather site (see section 4.2.1), is can easily be applied to any other Web site to find Weather forecast information. Some data may be missing (not all weather sites have four day forecasts), but a large proportion of the template data will be found.

There will be multiple Web sites for any one type of data. For example, for airline information, the seMole framework will have access to templates for CO.O.L. Travel Assistant as well as United Airline's and other carrier's Web sites. When the user asks for information about a flight, the seMole will search through all these templates to find the necessary information. This will make seMole even more robust and easier to use than the current iteration of the seMole.

To accomplish this, the seMole framework has to be extended: it has to be able to choose which Web site to retrieve information from and it also has to be able to rank template results to choose the best match. A Web site policy manager (akin to seMole's cache) can accomplish the first of these goals. However, a new mechanism will be required to rank the template results. This ranking must take into account the timeliness of the data

and how well the data matches the templates. Once done, however, the seMole framework can become a semantic gateway to the World Wide Web. Users will only be burdened to ask the seMole framework the type of information they wish to retrieve, and seMole will be able to intelligently find the information for them.

Chapter 6

Conclusion

In this thesis, we have presented the seMole framework. This framework provides users a simple interface to build robust bridges between machines and the Web. By using semantic addressing, seMole is able to gather information from HTML pages even if the page's façade or data change. Since many of today's Web pages change constantly to update data or to update users' interest in the site, semantic addressing promises to be invaluable for future machine to Web interfaces.

By having users develop various harvesting templates for various sites on the Web, we have shown that seMole is more robust than competitor Web harvesting packages. Even if the underlying HTML structures of a Web page change drastically, seMole's templates still harvested the correct data although other frameworks that use absolute addressing have failed. Even though the seMole framework has proven to be more robust, it is not necessarily faster nor is it easier to use. However, in the future, seMole will incorporate various new components and enhancements to make the seMole faster and easier to use.

As the Web continues to evolve, new languages will likely make data harvesting even simpler in the future. For example, XML promises to empower machines with knowledge of exactly what each piece of Web data represents. XML can potentially displace the need for frameworks such as seMole, absMole, and B2B. However, since almost all of today's Web pages are still developed using HTML and since XML is still a nascent

technology, the need for a bridge between machines and HTML will continue to exist for quite some time.

SeMole represents our efforts to allow machines to "see" HTML pages as humans do. Humans, when presented with an HTML page, rely on semantic keys/labels to find data. SeMole and semantic addressing offers machines this same ability: to locate data using labels and labels' positioning in the HTML space. Therefore, seMole not only a bridge between HTML and machines, it is also a bridge between today's human-oriented data and structured machine code.

Appendix

In the robustness experiments, we created two templates for the CNN Weather site. One template was written in the seMole framework and the other was written the absMole framework. This seMole template was created using only AFTERNEAR and BEFORENEAR commands. This template is listed below:

```
;;
;; get_cnnweather.template
;;
;; seMole template to gather information from
;; the CNN Weather Site (http://www.cnn.com/WEATHER/)
;;
;; Note that the DEFUNC command defines this template
;; as an executable template.  Usually, templates are
;; automatically executed by seMole.  When templates
;; are defined usin the DEFUNC command, the template
;; is stored in seMole's namespace and only run when
;; it is called by another template.

<defunc cnnWeather>
  "htmlinput";
  <set result>
        ;; first create the proper URL, and then
        ;; fetch the page from the CNN site
        <defunc cnnaux>
                inf;
                inc;
                <set text:1 concat>
                        *inf;
                        " (";
                        *inc;
                        ")";
                </set>
        </defunc>
        <let html fetchpage>  ;; retrieval is done here
                get;
                *htmlinput;
        </let>

        ;; find the four day-forecasts found on the page
        ;; using semantic groups
        <let daycondition afternear>
                *html;
                "day";
                "Mon\.|Tue\.|Wed\.|Thu\.|Fri\.|Sat\.|Sun\.";
                "weather"; "m/.*/";
        </let>
```

```
;; fetch four day high forecasts in F
<let highsF:0 afternear>
        *html;
        "let"; "HIGH$";
        "highf"; "[-]?[\d]+ F";
</let>

;; fetch four day high forecasts in C
<let highsC:0 afternear>
        *html;
        "let"; "HIGH$";
        "highc"; "[-]?[\d]+ C";
</let>

;; fetch four day low forecasts in F
<let lowsF:0 afternear>
        *html;
        "let"; "LOW$";
        "lowsF"; "[-]?[\d]+ F";
</let>

;; fetch four day low forecasts in C
<let lowsC:0 afternear>
        *html;
        "let"; "LOW$";
        "lowsC"; "[-]?[\d]+ C";
</let>

;; combine four day forecasts in a temporary
;; variable called HI and LO.  Will combine
;; these variables at the end of the template
<let hi foreach>
        "cnnaux";
        *highsf;
        *highsc;
</let>
<let lo foreach>
        "cnnaux";
        *lowsf;
        *lowsc;
</let>

;; fetch the city namne
<set city:0 beforenear>
        *html;
        "let";"forecast";
        "city";"(.*\, .*)\s\S+$";
</set>
```

```
;; fetch the current time
<set time:0 afternear>
        *html;
        "let"; "m/current conditions/i";
        "time";"m/\d* [ap]\.m\./i";
</set>

;; fetch the current date
<set date:0 afternear>
        *html;
        "let"; "m/current conditions/i";
        "date";"m/\S* \d*$/i";
</set>



;; fetch the current weather
<set weather:0 afternear>
        *html;
        "let"; "m/current conditions/i";
        "date";"m/.*/i";
        "weather";"m/.*/i";
</set>


;; fetch the current temperature in F
<let tempF:2 afternear>
        *html;
        "let";"m/Temp/i";
        "tempF";"[-]?[0-9]+\s*F";
</let>

;; fetch the current temperature in C
<let tempC:2 afternear>
        *html;
        "let";"m/Temp/i";
        "tempF";"[-]?[0-9]+\s*C";
</let>

;; combine the high and low temperatures
;; so the user is returned one temperature string
<set temp concat>
        *tempF;
        " (";
        *tempC;
        ")";
</set>
```

```
        ;; fetch the current humidity
        <set humidity:2 afternear>
                *html;
                "let";"humidity";
                "humidity";"[0-9]*.*";
        </set>

        ;; fetch the current wind.  Some formatting
        ;; was required to fetch this data, so a
        ;; temporary variable (windtemp) was used.
        <let windtemp afternear>
                5;
                *html;
                "let";"wind";
                "child";"[NWES]*\s*\S*\s*\S*|n\/a";
                "child";"\d+ \w\w\w|NULL";
                "child";"\(|NULL";
                "child";"\d+ \S+\)|NULL";
        </let>



        ;; format the current wind conditions
        <set wind concat>
                *windtemp.child:1;
                " ";
                *windtemp.child:2;
                " ";
                *windtemp.child:3;
                *windtemp.child:4;
        </set>

        ;; fetch the current barometric pressure
        <set pressure:2 afternear>
                *html;
                "let";"pressure";
                "pressure";"[0-9]*.*|n\/a";
        </set>

        ;; add the four day-forecasts to the end of template
        <set forecasts interleave>
                "forecast";
                <set lists>
                        *daycondition.day;
                        *daycondition.weather;
                        *hi;
                        *lo;
                </set>
        </set>
  </set>
</defunc>
```

The absMole template consisted of two separate templates. One template (FORECAST_AUX) was responsible for gathering all of the four day forecasts. The other template (GET_CNNWEATHER) gathered all of the current condition information and was also responsible for calling the first template.

```
;;
;; get_cnnweather
;;
;; This template harvests information from the CNN Weather
;; site (http://www.cnn.com/WEATHER). It takes in a
;; city name for an argument, fetches the page, and gathers
;; all of the current condition information. It also calls
;; "forecast_aux" to gather the four day forecasts.

;; fetch the page
newurl = text "http://cnn.com/WEATHER/html/"
$get_cnnweather.city ".html"
HTML = newurl.text.follow

;; a rudimentary exception mechanism: check if found city
if HTML.grep -i "not found"
        res_CNNWeather_excpt = makecontext
        temp = text "The city " get_cnnweather.city " was
not found"
        res_CNNWeather_excpt.addchild temp.text:1
        return res_CNNWeather_excpt
        quit
endif

;; setup... find the master table which wraps around all of
;; the information that we wish to retrieve
mastertable = HTML:1.table:10.copy
res_CNNWeather = mastertable:1.makecontext

;; get city name from the page
test = mastertable.text:1.copy
test = test.replace " Forecast" " "
test = test.replace "Forecast" " "
res_CNNWeather.makechild "city" -with test

;; get current conditions
conditions = mastertable:1.grep -i "condition"
conditions = conditions.movelist -up
conditions = conditions.movelist -up
conditions = conditions.movelist -up
conditions = conditions.movelist -up
conditions = conditions.movelist -up
```

```
;; get current time
temp = conditions:1.text:2.match ".*\m\."
temp = temp.text:1.replace " " " "
temp = temp.replace "  " " "
temp = temp.replace "  " " "
res_CNNWeather.makechild "time" -with temp



;; get the date
temp = conditions:1.text:2.match "\,\ .*"
temp = temp.text.replace ", " " "
res_CNNWeather:1.makechild "date" -with temp


;; current condition
res_cnnweather.makechild "weather" -with condi-
tions.tr:2.text:1

;; current temp
temp = conditions.tr:2.text:3.copy
temp1 = temp.match "^.*\,"
temp2 = temp.match "\,.*$"

temp1 = temp1.text:1.copy
temp2 = temp2.text:1.copy
temp1 = temp1.replace "," " "
temp2 = temp2.replace "," " "
temp1 = temp1.replace "&deg;" " "
temp2 = temp2.replace "&deg;" " "
temp1 = temp1.replace "  " " "
temp2 = temp2.replace "  " " "
if temp1.grep -i "c"
temp = temp2:1.text temp2 " (" temp1 ")"
else
temp = temp:1.text temp1 " (" temp2 ")"
endif
temp = temp.text.replace "  " " "
temp = temp.replace "\( " "\("
temp = temp.replace " )" ")"
res_cnnweather.makechild "temp" -with temp

;; get humidity info
if humid = mastertable:1.grep -i "humid"
humid = humid.movelist -up
humid = humid.movelist -next
res_cnnweather.makechild "humidity" -with humid:1.text:1
else
res_cnnweather.makechild "humidity" -with "n/a"
endif
```

```
;; get wind info
if wind = mastertable:1.grep -i "wind"
temp = text conditions.table:2.text:2 " " condi-
tions.table:2.text:3 " " conditions.table:2.text:4 condi-
tions.table:2.text:5
res_cnnweather.makechild "wind" -with temp.text
else
res_cnnweather.makechild "wind" -with "n/a"
endif

;; get barometric pressure
if press = mastertable:1.grep -i "pressure"
press = press.movelist -up "td"
  if test = press.grep -i "n/a"
        res_cnnweather.makechild "pressure" -with "n/a"
  else
        press = press.movelist -next
        res_cnnweather.makechild "pressure" -with
press:1.text:1
  endif
else
res_cnnweather.makechild "pressure" -with "n/a"
endif

;; get the forecasts by calling a sub procedure:
;; "aux_forecasts"
forecasts = mastertable.table:2.copy
removechild forecasts.tr:1
forecasts = forecasts.tr.map "aux_forecast"
res_cnnweather.addchild forecasts

;; return the result
return res_cnnweather

;;
;; aux_forecast template
;;

forecast = TD:1.makecontext
forecast.makechild "day" -with td.text:1

;; get the condition
forecast.makechild "weather" -with td.text:2

;; get hi
temp1 = td.text:4.copy
temp2 = td.text:5.copy
if temp1.grep -i "c"
temp = temp2:1.text temp2 " (" temp1 ")"
else
temp = temp:1.text temp1 " (" temp2 ")"
endif
forecast.makechild "hi" -with $temp
```

```
;; get lo
temp1 = td.text:7.copy
temp2 = td.text:8.copy
if temp1.grep -i "c"
temp = temp2:1.text temp2 " (" temp1 ")"
else
temp = temp:1.text temp1 " (" temp2 ")"
endif
forecast.makechild "lo" -with $temp

;; return the result
return forecast
```

```
;;
;; aux_forecast
;;
;; Called by get_cnnweather.  Responsible for finding
;; and returning the four day forecasts found on each
;; CNN weather page.
;;

forecast = TD:1.makecontext
forecast.makechild "day" -with td.text:1

;; get the condition
forecast.makechild "weather" -with td.text:2

;; get hi forecasts
temp1 = td.text:4.copy
temp2 = td.text:5.copy
if temp1.grep -i "c"
temp = temp2:1.text temp2 " (" temp1 ")"
else
temp = temp:1.text temp1 " (" temp2 ")"
endif
forecast.makechild "hi" -with $temp


;; get lo forecasts
temp1 = td.text:7.copy
temp2 = td.text:8.copy
if temp1.grep -i "c"
temp = temp2:1.text temp2 " (" temp1 ")"
else
temp = temp:1.text temp1 " (" temp2 ")"
endif
forecast.makechild "lo" -with $temp

;; return the result
return forecast
```

Bibliography

[1] "Extensible Markup Language," World Wide Web Consoritium, 1998. (http://www.w3.org/XML/).

[2] "WebMethods: B2B," WebMethods Corporation, 1998. (http://www.webmethods.com/solutions/products/index.html).

[3] D. Goddeau, E. Brill, J. Glass, C. Pao, M. Phillips, J. Polifroni, S. Seneff, and V. Zue, "Galaxy: A Human-Language Interface to On-Line Travel Information," *Proc. International Conference on Spoken Language Processing*, pp. 707-710, Yokohama, Japan, Sep. 1994.

[4] R. Lau, G. Flammia, C. Pao, V. Zue, "WebGALAXY - Integrating Spoken Language and Hypertext Navigation," in *Proc. Eurospeech 1997*, pp. 883-886, Rhodes, Greece, Sep. 1997.

[5] C. Allen, "WIDL: Application Integration with XML," in *World Wide Web Journal*, Vol.2, Issue 4, Fall 1997. (http://www.agentsoft.com/whit.html).

[6] "CNN - Weather," Cable News Network, 1998. (http://www.cnn.com/WEATHER).

[7] "CO.O.L. Travel Assistant," Continental Airlines, 1998. (http://cooltravelassistant.com).

[8] "Tcl/Tk Information," The Tcl/Tk Consortium, 1998. (http://www.tcltk.com/)

[9] "W3C Publishes First Public Working Draft of XSL 1.0," World Wide Web Consortium, 1998. (http://www.w3.org/Press/1998/XSL-WD).

[10] "Java Web Server," Sun Microsystems, 1998. (http://jserv.java.sun.com/products/webserver/index.html)

[11] "Microsoft XML Parser in Java," Microsoft Corporation, 1998. (http://www.microsoft.com/workshop/c-frame.htm#/xml/default.asp)

[12] "Document Object Model," World Wide Web Consortium, 1998. (http://www.w3.org/DOM/)

[13] "Boston Sidewalk," Microsoft Corporation, 1998. (http://boston.sidewalk.com).

[14] "Mathematical Markup Language (MathML) 1.0 Specification," World Wide Wed ConsortiWum, 1998. (http://www.w3.org/TR/REC-MathML/).

[15] "Mapquest," GeoSystems Global Corporation, 1998. (htttp://www.mapquest.com).