WebL — a programming language for the Web

Thomas Kistler[a] and Hannes Marais[b]

[a]*Information and Computer Science, University of California, Irvine,*
Irvine, CA 92697-3425, USA
kistler@ics.uci.edu

[b]*DIGITAL Systems Research Center*
130 Lytton Avenue, Palo Alto, CA 94301, USA
marais@pa.dec.com

Abstract

In this paper we introduce a programming language for Web document processing called WebL. WebL is a high level, object-oriented scripting language that incorporates two novel features: service combinators and a markup algebra. Service combinators are language constructs that provide reliable access to Web services by mimicking a Web surfer's behavior when a failure occurs while retrieving a page. The markup algebra extracts structured and unstructured values from pages for computation, and is based on algebraic operations on sets of markup elements. WebL is used to quickly build and experiment with custom Web crawlers, meta-search engines, page transducers, shopping robots, etc.

Keywords

Scripting language; Service combinators; Markup algebra

1. A computation model for the Web

When we think of the Web, we immediately think of a planet-wide distributed hypertext. However, it is also compelling to regard it as a distributed database, from which information is queried and processed. In fact, XML was designed to make the Web an information repository accessible by program [XML]. However, whereas traditional databases have specialized programming languages like SQL to query and process their content, there are no specialized languages yet for performing such tasks on the Web. Our premise is that the architectural, physical and administrative constraints of the Internet require new models for computing over planet-wide structures such as the Web.

Some of the characteristics of the Web, like its wide area distribution, unreliable services, lack of referential integrity, security model, and lack of data typing, differ immensely from those of traditional programming models, which presupposes a non-distributed, well-structured, and predictable infrastructure. Furthermore, because of the Web's geographical distribution, latency and bandwidth — not CPU speed and memory size — become the limiting factors that need to be addressed. So what kind of programming models and programming constructs

are needed to compute on the Web? To understand this question, we first have to study typical Web computations. In our view, a typical Web computation can be divided into three phases.

The input phase involves fetching one or more Web pages for processing. During this phase we have to contend with the Web's geographic distribution and architectural inefficiencies. For example, one or more of the following situations might apply when retrieving a page from a Web service:

- The page is available and can be retrieved successfully.
- The server is unavailable or provides intermittent service due to a high load.
- The page is (perhaps temporarily) unavailable or was redirected to another server.
- The connection is unexpectedly terminated or the data transfer speed varies, stalling or dropping to an unacceptable rate.
- The page is mirrored geographically, perhaps on servers with different capacity.

Consequently, a programming model for the Web not only has to expect several modes of failure (for which many programs are typically not designed) but should also provide functionality to overcome these problems, for example, to exploit the inherent parallelism of replicated servers. Our approach is to use *service combinators* to make access to services more reliable and to simplify the handling of failures (see Section 3).

The processing phase of a typical Web computation involves extracting data values from pages and performing computations on these data values. We assume pages to be marked-up in either XML or HTML, so as to exploit the structural content of the page. The difficulties encountered in this phase include poorly structured pages, HTML markup that does not reflect the document structure (for example, misuse of tables for presentation effects), and pages that change format regularly or unexpectedly. Our data extraction technique is based on a *markup algebra* that performs operations on sets of elements in a page (see Section 4).

The output phase of a typical Web computation covers the generation of Web documents from values computed during the processing phase, and storing them back on the Web (for example, by publishing the page on a Web server).

Figure 1 depicts this general model of a Web computation. Web pages flow through a pipeline of service combinators for fetching pages, a markup parser, the markup algebra for extracting (or "searching" for) data values from (on) a page, computing on those values, and page manipulation. Searching, computing and manipulation is repeated as often as needed. Finally the page is regenerated from its internal representation by the markup generator, and stored back on the Web.
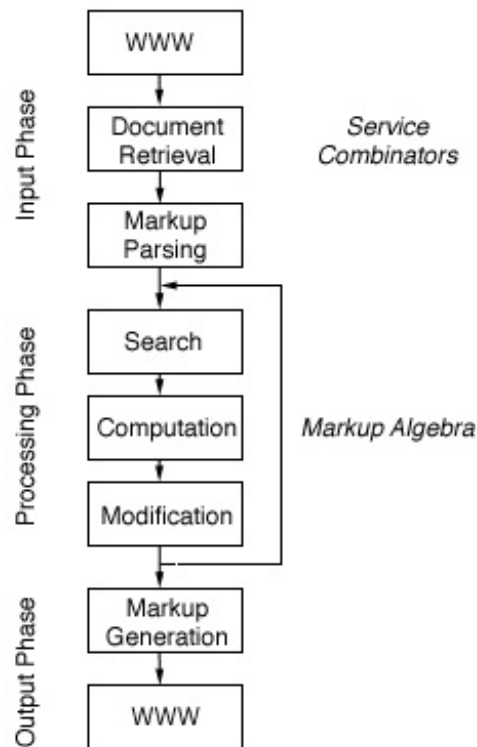
Fig. 1. A model for Web computation.

2. The programming language WebL

Our implementation of this computation model is called *WebL*, a scripting language that was specially created to experiment with *service combinators*, *markup algebras*, *page modification operations*, and in general the automating of Web-related tasks.

We implemented our WebL prototype in Java. The core functionality consists of libraries that implement the parsers, service combinators and markup algebra. The idea is that programmers that don't want to use the WebL language, could eventually use these libraries directly in their Java programs without penalty (except for giving up the compact syntax).

Although we regard the language as secondary to the novel features we introduce, the language has grown over time to support many conventional languaging features like dynamic typing, objects, modules, closures, exceptions, sets, lists, associative arrays, multithreading, etc. These features make it a convenient language for prototyping computations on the Web and quite a useful tool for Web masters.

Some of the applications we constructed with WebL so far include:

- Customizable Web crawlers.
- Meta-search engines for popular search engines on the Web.
- Meta-newspapers that collect articles from several sites according to interests.
- Tools to build a newspaper from CDF-based descriptions [CDF97].
- Shopping robots that shop for the cheapest books at several electronic bookstores.
- Tools to extract financial information from stock pages.
- Tools to extract and compute project information from DIGITAL's intranet.
- Tools to concatenate Latex2HTML-generated documents for printing.
- Tools to validate links in Web pages.

In the remainder of the paper we will concentrate on only two novel aspects of WebL, namely service combinators (Section 3) and the markup algebra (Section 4). These sections are followed by related work (Section 5) and conclusions (Section 6). An Appendix lists example programs.

3. Service combinators

An experienced Web surfer exploits a repertoire of behaviors when confronted with the situations introduced in Section 1 (e.g. server failure, stalling or dropping service rates, etc.). We call these behaviors *Web reflexes*. For example, users may

- reload a page on a stalled link,
- retry requests, taking short pauses in between requests,
- terminate a request that takes too long,
- switch to less used servers with the same information,
- switch to alternate sources of information,
- monitor the transfer rate and decide whether to wait for the page to arrive,
- run fetches in parallel, waiting for the first to finish, and stopping the other requests.

A strategy for making computations on the Web more reliable is to use programming constructs called *service combinators* introduced by Luca Cardelli and Rowan Davies [CD97]. The main purpose of service combinators is to mimic these reflexes or, in a more general way, to make any algorithmic behavior of Web users scriptable. Therefore, under the basic premise that by providing the programmer with easier ways to express these reflexes and it becomes easier to write robust scripts, service combinators provide explicit language constructs to automate handling of time-out and failure, exploitation of replicated data, etc. As in the approach suggested in [CD97], WebL maps service combinators directly onto operators of the language. As will be noticed from the following examples, service combinators are also convenient language constructs for handling exceptions.

For the remainder of this section *S* and *T* denote operands (called *services*), which may contain primitives to fetch pages or general WebL computations.

Services

```
getpage(string, [. param1=val1, param2=val2, ... .],
        [. header1=val1, header2=val2 .])
postpage(string, [. param1=val1, param2=val2, ... .],
        [. header1=val1, header2=val2 .])
```

The *getpage* function fetches with the HTTP GET protocol the resource associated with the string URL. The result returned is a page object that encapsulates the resource. The function fails if the fetch fails. The second and third arguments to getpage are optional — when specified, they provide the server with query arguments and HTTP headers respectively. A similar function called *postpage* uses the HTTP POST protocol, used to fill in Web-based input forms.

```
// This program simply attempts to fetch the named URL.
page := getpage("http://www.digital.com")


// This program looks up the word "java" on the

// AltaVista search engine.

page := getpage("http://www.altavista.digital.com/cgi-bin/query",
    [. pg="q", what="web", q="java" .])
```

Sequential execution   S ? T

The "?" combinator allows a secondary service to be consulted in case the primary service fails for some reason. Thus, the service *S ? T* acts like the service *S* except that if *S* fails then it executes the service *T*.

```
// This program first attempts to connect to AltaVista

// in California, and in the case of failure, attempts to

// connect to a mirror in Australia

page := getpage("http://www.altavista.digital.com") ?
        getpage("http://www.altavista.yellowpages.com.au")
```

Concurrent execution   S | T

The "|" combinator allows two services to be executed concurrently. The service *S | T* starts both services *S* and *T* at the same time and returns the result of whichever succeeds first. If both *S* and *T* fail, then the combined service also fails.

```
// This program attempts to fetch a page from one of the two

// alternate sites. Both sites are attempted concurrently, and the

// result is that from whichever site successfully completes first.
```

```
page := getpage("http://www.altavista.digital.com") |
        getpage("http://www.altavista.yellowpages.com.au")
```

Time-out   `timeout(t, S)`

The time-out combinator allows a time limit to be placed on a service. The service *timeout(t, S)* acts like *S* except that it fails after *t* milliseconds if *S* has not completed within that time.

```
// This program attempts to connect to alternative AltaVista mirror sites,
// but gives a limit of 10 seconds to succeed.
page := timeout(10000, getpage("http://www.altavista.digital.com") |
                       getpage("http://www.altavista.yellowpages.com.au "))
```

Repetition   `repeat(S)`

The repeat combinator provides a way to repeatedly invoke a service until it succeeds. The service *repeat(S)* acts like *S*, except that if *S* fails then *S* starts again. The loop can be terminated by writing *timeout(t,repeat(S))*.

```
// This program makes repeated attempts in the
// case of failure, alternating between two services.
page := repeat(getpage("http://www.altavista ...") ?
               getpage("http://www.altavista ..."))
```

Non-termination   `stall()`

The stall combinator never completes or fails.

```
// This program repeatedly tries to fetch the URL, but
// waits 10 seconds between attempts.
page := repeat(getpage("http://www.digital.com") ? timeout(10000, stall())
```

4. Structured text search on Web pages

4.1. An algebra for text search

One of the challenges in structured text search is to support a unified model of different views of a document. In one view we are interested in the linear text flow of the page (without tags), for example to locate words and character patterns. In another view, we are interested in the hierarchical organization of the document, for example to use markup as "landmarks" for guiding data extraction. Other views, such as a publisher's view that divides the document into lines, paragraphs, and columns, are also imaginable. In addition, we observe that different

views of the document are not always properly nested, as are rows in tables, or words in titles. Rather, elements in different views might overlap. Sentences usually go across multiple lines and images might span multiple columns. As a consequence, a unified model has to allow searching on several views, mixing of query results from different views, and handling of overlapping elements in the same or different views.

WebL's data extraction language addresses these problems with the notion of a markup algebra. The *markup algebra* is based upon the concepts of *pieces*, *piece-sets* and algebraic *operators* that are applied to piece-sets.

First, we define a *piece* as a contiguous text region in a document, identified by the starting and the ending position of the region. For this paper we can imagine positions as indices that indicate a character offset in the page, which makes it easy to determine by numerical comparison the relationship between two regions, such as whether two pieces overlap, are contained in each other, or follow each other. (Our WebL implementation uses a more complicated data structure for pieces that simplifies searching and page modification.) We further define a *piece-set* as a collection of pieces. Pieces within piece-sets may overlap, be nested, or may belong to different pages. However, unlike mathematical sets that do not impose a particular ordering on their elements, piece-sets are always in a canonical representation in which pieces are ordered according to their starting position, and then their ending position in the document. This allows iterating over pieces in a set in the sequence they appear in the document, and also to pick the *n*th occurrence of a pattern (by indexing into the piece-set). Both pieces and piece-sets are mapped to special objects in WebL, which means that they can have attributes and be manipulated by programs.

A common way to create a piece-set is to search for all the HTML or XML elements with a specific name (we call this a *structured search*). For example, the following program returns all the anchors (hyperlinks) that occur on the DIGITAL homepage by calling a method called *Elem* of the page object *P*:

```
P := getpage("http://www.digital.com/");
links := P.Elem("A")    // returns a piece-set of "A" elements
```

After the method invocation, the variable `links` contains a piece-set that, for every matching HTML or XML element, contains a piece that points to the starting and ending position of the element. In addition, all the element names and attributes are made visible to the programmer by associating them with the appropriate piece object.

Another way to create a piece-set is to search for character patterns, ignoring the markup (we call this *unstructured search* or *pattern search)*. The *Pat* method of a page object extracts all the occurrences of a Perl 5-style regular expression [Fri97] in the text of a page. The following example extracts the occurrences of the word "Digital" or "digital" in the Digital home page.

```
P := getpage("http://www.digital.com/");
words := P.Pat("(D|d)igital")
```

If the regular expression contains Perl-5 groups, the matching groups are accessible as attributes of the piece. Like the Elem method, the Pat method computes a set that, for every match, contains a piece that points to the starting and ending position of the match.

Finally, we define a *set operator* *S* ¤ *T* as an algebraic operation ¤ between two piece-sets *S* and *T* that returns a third piece-set as a result.

For the remainder of this section, *S* and *T* denote piece-sets, the elements of *S* and *T* are referred to as *s* and *t*, and *P* stands for a page object. WebL divides set operators into groups of *basic set manipulation* operators, *positional* set operators, and *hierarchical* set operators, which will be discussed in the following sections. In the interest of conciseness, we will not describe the negated operators (those starting with an exclamation point), as their behavior is easy to deduce.

4.2. Basic set operators

Union	`S + T`
Intersection	`S * T`
Exclusion	`S - T`

Basic set operators are used for basic set manipulation. They contain a set union operator, a set intersection operator, and a set exclusion operator. The set union operator merges the two sets *S* and *T* and eliminates duplicate pieces. The set intersection operator returns the set of all pieces that are contained both in *S* and *T*, and the set exclusion operator calculates the set of pieces that are contained in *S* but not in *T*. As an example, the following program retrieves all the level one and level two headings in a page:

```
titles := P.Elem("H1") + P.Elem("H2")
```

4.3. Positional operators

`S before T`	`S !before T`
`S after T`	`S !after T`
`S directlybefore T`	`S !directlybefore T`
`S directlyafter T`	`S !directlyafter T`
`S overlap T`	`S !overlap T`

Positional operators provide functionality to query on the locality property of pieces, such as searching for pieces that are located above or below other pieces in the linear text flow of the document.

The `before` operator computes the set of pieces in *S* that are located before some piece in *T*. We define a piece *s* to be located before a piece *t* if the ending position of *s* precedes the starting position of *t*. Correspondingly, the `after` operator returns the set of pieces in *S* that are located after some piece in *T*. Although they are very effective, these two operators are not always sufficient. As an example, in some cases we might not be interested in all the occurrences of a link after a special keyword, but only in the very first occurrence of a link after the special keyword. In this case, we use the stronger operators `directlybefore` and `directlyafter` that return the set of only the closest

pieces in *S* that follow or precede some piece in *T*. We also call the latter *non-transitive* versions of the `before` and `after` operators. The following example depicts the differences between these operators:

```
...
<IMG SRC="napa.gif">
<I>Fig. 1. Sonoma and Napa</I>        // I1

...
<IMG SRC="tahoe.gif">
<I>Fig. 2. Lake Tahoe</I>             // I2

...
<I>Northern California</I>            // I3

...
<IMG SRC="mendocino.gif">
<I>Fig. 3. Mendocino</I>              // I4
...
```

In order to retrieve the title of the first image we write the following program, assuming that the first text stretch in italics in our excerpt is also the first text stretch in italics in the whole document:

```
// retrieve the first italic text stretch -> {I1}
title := P.Elem("I")[0]
```

Searching for all the words in italics that follow an image yields the result set containing pieces I1, I2, I3, and I4.

```
// retrieve all the titles that follow an image -> {I1, I2, I3, I4}
titles := P.Elem(I") after P.Elem("IMG")
```

To retrieve all the titles of the figures we use the `directlyafter` operator that does not return the phrase "Northern California", since it does not directly follow an image.

```
// retrieve all the titles of the figures -> {I1, I2, I4}
titles := P.Elem("I") directlyafter P.Elem("IMG")
```

Finally, the `overlap` operator returns all the pieces in *S* that overlap with some piece in *T*.

4.4. Hierarchical operators

```
S in T                          S !in T
```

```
S contain T              S !contain T

S directlyin T           S !directlyin T

S directlycontain T      S !directlycontain T
```

In contrast to positional operators that provide functionality for expressing locality relationships between pieces, hierarchical operators provide functionality for expressing containment and inclusion relationships between piece.

The `in` operator returns the set of pieces in *S* that are contained in some piece in *T*. We define a piece *s* to be contained in a piece *t* if the starting position of *s* follows or is equivalent to the starting position of *t*, and the ending position of *s* is at most the ending position of *t*. Equivalently, the `contain` operator returns the set of pieces in *S* that contain some piece in *T*. As an example, to search for all the rows in the third table of a page, we write,

```
rows := P.Elem("TR") in P.Elem("TABLE")[2]
```

and to search for all the level two headings that mention the word UCI we write

```
titles := P.Elem("H2") contain P.Pat("UCI")
```

As for positional operators, we define two stronger, non-transitive operators, `directlyin` and `directlycontain`, that address direct containment and direct inclusion properties (between parents and their direct children in the parse tree of a page). For example, `directlyin` reduces the result *x* of the `in` operator to the "outermost" pieces of *x* that are not contained in other elements of *x*. The following example depicts the differences:

```
...
<UL>
    <LI>First Section</LI>              // LI1
    <LI>Second Section</LI>             // LI2
    <LI>Third Section</LI>              // LI3
        <UL>
            <LI>First Subsection</LI>   // LI4
            <LI>Second Subsection</LI>  // LI5
        </UL>
    </LI>
    <LI>Fourth Section</LI>             // LI6
</UL>
...
```

To retrieve all the list items in this unnumbered list, we write the following program, assuming that there is no other unnumbered list

preceding this section in the document:

```
// retrieves all the subsections -> {LI1, LI2, LI3, LI4, LI5, LI6}
subsections := P.Elem("LI") in P.Elem("UL")[0]
```

However, in many cases we are not interested in nested lists and would only like to retrieve the list items of the top-level list. Therefore we use the `directlyin` operator and write:

```
// retrieve only the toplevel subsections -> {LI1, LI2, LI3, LI6}
subsections := P.Elem("LI") directlyin P.Elem("UL")[0]
```

5. Related work

Cardelli and Davies [CD97] include an extra service combinator, *limit(t, r, S)*, that acts like the service *S*, except that each connection is considered to have failed if the rate ever drops below *r* Kbytes/sec after the first *t* seconds of the connection. The consequence is that their service combinators can only operate on Web services, and not general computations (which don't have a "rate"). Thus by removing *limit* from WebL, we can apply combinators to any computation. For example, a failure might occur when a page was fetched successfully but the content of the page is invalid or unexpected. We can reintroduce the rate limit feature as part of the *getpage* and *postpage* primitives themselves, which fail appropriately when the rate is not met.

A popular technique for searching in text is *regular expressions* [Fri97, IEEE92]. The limitations of regular expressions are twofold: they lack information about the structure of the document and they apply a "leftmost longest match" rule that is inappropriate for nested data structures. Searching for a table, for example, only returns a correct match if there is only one table in the document. This problem is described by Clarke and Cormack [CC97].

The prominent techniques for information extraction from structured text are tree matching, grammar parsing, and set algebras.

In *tree matching*, the search problem is reduced to searching a subtree in a parse-tree [Kil92]. The disadvantage of tree matching is the lack of orthogonality and compositionality regarding different views. Queries that search for character patterns cannot be mixed with searches for special structures in the document. Many of the tree matching problems cannot be solved in linear time.

Context free grammars pursue an approach in which the search pattern is specified as a context free grammar [ST96]. The result of a search query are all the substrings in the document that are accepted by the specified grammar. Context free grammars are very expressive in that they allow the definition of recursive search queries, but they suffer from the same problems as tree matching: they cannot express view-spanning and overlapping queries.

Several new techniques are based on set algebras [ST92, JK95, CCB94]. The *Standard Document Query Language* (SDQL) of the *Document Style Semantics and Specification Language* [DSSSL96] introduces the concepts of nodes and node-lists, which are loosely related to our pieces and piece-lists. Some of the WebL operators are provided and the user can also program new ones in a Lisp-like language. The data

structures on which SDQL operates — called *groves* — are trees of nodes corresponding to elements in the document, and thus multiple views and overlapping elements cannot be modelled. *PAT expressions* [ST92] use a set-at-a-time algebra for manipulating sets of match-points and sets of regions. In contrast to the WebL search algebra, PAT expressions do not support an orthogonal and unified model. Sets of match-points and sets of regions cannot be arbitrarily composed and, in regard to document transformation, match points are not very practical since only the starting position of a match is recorded. However, most of these problems can be avoided. Clarke, Cormack, and Burkowski propose a compositional *structured text algebra* that is based on the notion of sets and ranges [CCB94]. Apart from the WebL set-algebra, this is the only other approach that supports overlappings between views. Although the model supports overlappings, the language does not. Additionally, nestings are avoided by selecting the minimal segments from those set elements that nest. Concerning runtime complexity, all of the set algebra problems can be solved in linear time if no two elements in a set overlap [NY96]. In the unlikely worst case, when all the elements in the set overlap with each other, the runtime complexity is quadratic in the number of elements in the set.

There are also efforts to specify low-level programming APIs that provide users with the functionality for document navigation and manipulation. The prominent activity in this area is W3C's document object model [DOM97]. In contrast to WebL, DOM is restricted to manipulating and searching single HTML and XML elements, does not provide a notion of character patterns, does not support multiple overlapping views, and inherently cannot perform computation.

There are also recent proposals for automating tasks on the Web. The Web Interface Definition Language (WIDL) [MA97] enables automation by mapping Web content into program variables using declarative descriptions of resources. WIDL provides features to submit queries and to extract features from the resulting pages. WIDL does not determine itself how search is to be done, but rather uses the Java Page Object Model [JS] or the Document Object Model [DOM97]. Page manipulation is not supported. WebSQL [AMM97] is a declarative query language for extracting information from the Web. The language emphasis is on extracting connectivity information from pages (i.e. pages that are two hops away from a specific page). WebSQL regards HTML documents as monolithic objects, and its analyses are limited to simple text matching techniques.

6. Conclusions and future work

We presented a novel programming language WebL for document processing on the World Wide Web. WebL features two distinguishing features, namely service combinators and a markup algebra. Service combinators allow the construction of reliable services and the markup algebra supports the extraction of data values from Web pages in a straightforward manner. The language provides features to perform computations on data values, and generate or manipulate Web pages appropriately. The resulting tool is well-suited for automating tasks on the Web and for building and experimenting with Web computations.

We currently plan to build larger applications with WebL and to extend it with additional libraries in the domain of information retrieval. One of the extensions we are currently investigating is to generate WebL scripts automatically by demonstration.

Acknowledgements

Not only did we receive valuable input from our colleague Luca Cardelli regarding service combinators, but we also adopted the name he coined for the language. The research on service combinators and the markup algebra were perform by Rowan Davies and Thomas Kistler respectively as part of the intern program at DIGITAL's Systems Research Center.

Appendix

We include two listings to give a flavor of WebL. Listing 1 implements a simple function to retrieve a stock quotation from a service. Listing 2 implements a more complicated function to search for books by title or author at the electronic book store *Amazon.com*. It fills in the query form, analyses the results, and returns a list of book objects for each book found.

Stock Quotes								
Symbol	Date	Time	Open	High	Low	Last	Change	Volume
DEC	11/20/97	16:03	48 9/16	49 1/4	48 5/16	49	+7/16	933,600

Fundamental Data				
Name	Earn/Shr	P/E	Div/Shr	52-wk Price Range
DIGITAL EQUIPMENT CORP	1.2710	38.5523	0.0000	53.81 - 25.00

```
stockQuote := fun(symbol)
   page := getpage("http://fast.quote.com/fq/quotecom/quote",
      [. symbols=symbol .]);
   (page.Elem("B") in
      (page.Elem("TABLE") contain page.Pat("Stock Quotes"))[0])[1].Text()
end;

s := stockQuote("DEC")
```

Listing 1. Retrieving a stock price from Quote.com

The Chamber ~ Ships in 2-3 days
John Grisham / Paperback / Published 1995
Our Price: $6.00 ~ You Save: $1.50 (20%)
Read more about this title...

The Client ~ Ships in 2-3 days
John Grisham / Hardcover / Published 1993
Our Price: $16.45 ~ You Save: $7.05 (30%)
Read more about this title...

The Client ~ Ships in 2-3 days
John Grisham, Blair Brown / Audio Cassette / Published 1993
Our Price: $16.45 ~ You Save: $7.05 (30%)
Read more about this title...

```
shopAmazon := fun(title, authorfirst, authorlast)
  books := [];
  params := [. .];
  params["author"] := authorfirst + " " + authorlast;
  params["author-mode"] := "full";
  params["title"] := title;
  params["title-mode"] := "word";
  params["subject"] := "";
  params["subject-mode"] := "word";
  page := postpage("http://www.amazon.com/exec/obidos/ats-query/", params);
  items := page.Elem("dd");
  every book in items do
    info1 := substring(book.Text(),
        `\w*([^/]*) (/ ([^/]*))?(/ [^\d]*(\d*))?`)[0];
    info2 := substring(book.Text(), `Our Price: \$(\d*.\d*)`);
    if (size(info2) > 0) and (info1[3] != "Audio Cassette") then
      books = books + [[.
      title = (page.Elem("a") directlybefore book)[0].Text(),
      link = (page.Elem("a") directlybefore book)[0].href,
      author = info1[1],
      type = info1[3],
      year = (select(info1[5],-2,-1) ? "N/A"),
      price = info2[0][1]
    .]]
    end
  end;
  books
end
```

Listing 2. A WebL function to shop for books

References

[AMM97] G.O. Arocena, A.O. Mendelzohn, and G.A. Mihaila. Applications of a Web query language, in: *Proceedings of WWW6* , 1997, Santa Clara, California,
http://atlanta.cs.nchu.ed u.tw/www/PAPER267.html

[CC97] C.L.A. Clarke and G.B. Cormack, On the use of regular expressions for searching text, *ACM Transactions on Programming Languages and Systems,* 19(3): 413–426, March 1997.

[CCB94] C.L.A. Clarke, G.V. Cormack, and F.J. Burkowski, An algebra for structured text search and a framework for its implementation, Department of Computer Science, University of Waterloo, Canada, Technical Report CS-94-30, August 1994.

[CD97] L. Cardelli and R. Davies, Service combinators for Web computing, Research Report 148, Digital Equipment Corporation Systems Research Center, Palo Alto, California, June 1997,
ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-14 8.html

[CDF97] Channel Definition Format (CDF), published by Microsoft Corp., September, 1997,
http://www.microsoft.com/stand ards/cdf.htm

[DOM97] W3C DOM working group, Document Object Model Specification, October 1997,
http://www.w3.org/TR/WD-DOM/

[DSSL96] Document Style Semantics and Specification Language (DSSSL), ISO/IEC 10179, 1996,
http://www.jclark.com/dsssl/

[Fri97] J.E.F. Friedl, *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools (Nutshell Handbook)* . O'Reilly and Associates, 1997.

[IEEE92] IEEE 1992. Standard for information technology – Portable Operating System Interface (POSIX) – Part 2 (Shell and utilities) – Section 2.8 (Regular expression notation). IEEE Std 1003.2, Institute of Electrical and Electronics Engineers, New York, NY, 1992.

[JK95] J. Jaakkola and P. Kilpelainen, SGREP, University of Helsinki, Department of Computer Science, 1995,
http://www.cs.helsinki.fi/ ~jjaakkol/sgrep.html

[JS] Netscape Corp., JavaScript Guide,
http://developer.netscape.com/library/documentation/communicator/jsgu ide4/index.htm

[Kil92] P. Kilpelainen. Tree matching problems with applications to structured text databases, Ph.D. Dissertation, Department of Computer Science, University of Helsinki, Report A-19992-6, Helsinki, Finland, November 1992.

[MA97] P. Merrick and C. Allen, Web Interface Definition Language (WIDL), published by webMethods Inc., September 1997,
http://www.w3.org/TR/NOTE-widl

[NY96] G. Navarro and R. Baeza-Yates, A class of linear algorithms to process sets of segments, in: *Proceedings of PANEL'96,* Vol. 2, 1996, pp. 671–682.

[ST92] A. Salminen and F. W. Tompa, PAT expressions: an algebra for text search, *Acta Linguistica Hungarica,* 41(1–4): 277–306, 1992 –93.

[ST96] A. Salminen and F.W. Tompa, Grammars++ for modelling information in text, Department of Computer Science, University of

Waterloo, Canada, Technical Report, CS-96-40, November 1996.

[XML] World Wide Web Consortium, Extensible Markup Language (XML), http://www.w3.org/XML/

Vitae
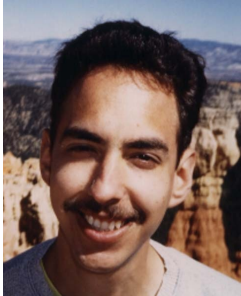
Thomas Kistler received the M.S. degree in computer science in 1995 from the Swiss Federal Institute of Technology (ETH) in Zurich, Switzerland. He is currently a Ph.D. student in the Department of Information and Computer Science at the University of California, Irvine. His research interests include programming languages, compilers, and dynamic runtime optimization.

Hannes Marais is a member of the research staff at DIGITAL's Systems Research Center (SRC) in Palo Alto, California. He holds a Ph.D from the Swiss Federal Institute of Technology (ETH) in Zurich. His interests include user interfaces, programming languages, and information retrieval applications on the Web.