A LANGUAGE FOR THE INTEGRATION OF WEB SOURCES

Cristina Bogdanschi[†] Alexandra Dumitrescu[†] Simone Santini[‡]

[†] Department of Software Engineering, University of Craiova, Romania
[‡] Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain

Abstract

This paper presents a language for the creation of data sources in mediation systems through navigation in web sites. The general idea of the language is to "thread together," so to speak, existing systems that parse and analyze single web pages into a navigation procedure spanning several pages of a web site, pages from different web sites, or web pages and other data repositories.

1 INTRODUCTION

Many data access problems today require, more that the creation of new repositories, the integration of existing ones. Traditionally, there are two classes of techniques for doing so. On one hand, one can bring all the necessary data into a single data base, using techniques such as *data warehousing*; on the other hand, one can leave the data in their original repositories and send, so to speak, the query to the data, using a *mediator* system—or something to that extent—to break the query into parts that the individual data repositories (*data sources,* in the mediation jargon) can answer, and then composing the partial results into a single answer, as required by the user (Chawathe et al., 1994; Garcia-Molina et al., 1995; Wiederhold, 1992)

In this paper we are mainly interested in the point of view of data mediation, which has several impoerant advantages over warehousing when it comes to managing web data. Since data mediation uses the data *in situ,* there is no unnecessary replication, and no periodic update so that, no matter how often the sources are updated, each query will access the latest version of the data. On the other hand, a mediation system has to deal with a plethora of data structures, and with data sources of widely different query capabilities, so that decomposing and optimizing a query is in general a rather formidable problem, whose solution is much more uncertain and problematinc than the corresponding data base problem.

Mediation systems are seen as an interesting solution for the integration of web data, because their characteristics adapt well to the volatile nature of the web substratum (Liu *et al.*, 2002). One important issue to be considered in this respect is that, while the mediator system typically assumes a logical and declarative model for a data source, web sites are quintessentially procedural. Representing a web source as, say, a collection of relations, as done, among others, by da Silva et al. (2002) and Zadorozhny et al. (2002) requires a method to map declarative queries against the conceptual relational model into a procedural plan to navigate the web site. The theory of such a mapping was developed, e.g., by Santini & Gupta (2004), and it relies on the definition of "navigation procedures" that would recurr the web site extracting data that would then be integrated into an answer to the query. While Santini & Gupta (2004) showed how to assemble a group of navigation procedures to answer queries expressed against a conceptual model of the web site, they left the definition of the navigation procedures open. In this paper we shall present a language for such a definition.

A number of systems have been developed in the past decade to analyze the content of web pages, select some desired data from them, and place the data in some specified structure. One can take as prototypical systems such as *Nodose* (Adelberg, 1998) or the more recent *Roadrunner* (Crescenzi et al., 2001). These systems are, by and large, oriented towards the syntactic analysis of single web pages, and take the form of declarative statements about the web page and the structure in which the data are to be mapped.

What we are after in this paper is of a different nature.

In a significant number of cases, the data that are needed to answer a single query are not conained in a single page, but can be collected by traversing links and obtaining several interconnected pages of the same web site, obtaining several interconnected pages from different web sites, or accessing web pages and other types of data repositories. For example, asking for the price of a given book on a bookseller's site consists of entering an author's name in a suitable query field, obtaining a page with a list of titles from that author, searching the desired title in the page, traversing the link attached to the title to retrieve the book's page, and searching the price figure in the book's page.

In this paper we present the design and teh implementation of a language for searches such as this one. The main idea here is to provide a system that will take care of the procedural aspects of a "navigation," but that will rely on existing systems to analyze individual pages or to access other data repositories.

2 LANGUAGE GENERALITIES

We set to design a language that was simple to implement and to use. The general plan was to use the language itself to integrate data from different sources and present a mediator system with a unified view (e.g. a relational view) over the set of sources. As mentioned in the introduction, there are many languages for parsing a web page and returning the data that it contains in a structured fashion, as well as languages for querying other sources of data. Our aim was to create a language for *navigating* these sources, be them web page or whatnot.

Consider, as an example, a query in which one wants to know the price of Miguel de Unamuno's *Niebla.* A query such as this one can be answered using the web page of a bookseller; let us say, for the sake of example, that we want to use the web site of Barnes and Noble booksellers, with which most of us are quite familiar. Using such a web site, our query would result in the following plan:

i) go to the main page of the site and enter the words "Unamuno" in the text entry area provided;

ii) if the page returned is the page with the information about a single book, then

  ii.a) if the title of the book is not "Niebla", then report an empty result;

iii) otherwise, if the page returned contains a list of titles, then

  iii.a) look for the title "Niebla";

  iii.b) if the title doesn't exist, report an empty result;

  iii.c) if the title exist, then traverse the link associated to it;

iv) read the price from the page and return it as a result.

This plan is executed completely within the same web site but this may not be the case: the plan might include access to several web sites, and to data repositories that are not web sites. As an example, one of the problems that motivated this project was the creation of a data integration system to assign a *clone* number to a genetic sequence derived with an *access number.* In this case, given an access number, a web data base (namely the data base of the *national center for biotechnology information*, an organization of the American *National Institutes of Health*) was searched to obtain a genetic sequence and, once the genetic sequence was obtained, a search was made in a local data base to obtain the desired access number.

In all these cases we note that, in addition to the extraction of information from web pages or from local data bases, the search problem involves the procedural specification of the "navigation" of a web site, or the specification of a procedure for dealing with data in various repositories.

The requirements for our integration language and system were then the following:

i) the language should allow the integration of query systems that operate on specific repositories ("wrappers" for web pages, the "BLAST" system (Altschul et al., 1997) for querying data bases of genetic sequences, systems to query relational and XML data bases, etc.);

ii) the language should connect these query systems together, which implies that it should be as much as possible neutral with respect to the formats that these systems adopt;

iii) the language should be of enough complexity to implement the common procedural structures (alternativs, cycles), but not overly complex, since the detailed processing of the data repositories is done by the query subsystems.

We found a compromise between these requirement by implementing a *stack* language, reminiscent of FORTH (Rather, Colburn & Moore, 1993): a language without variables in which the only storage structure is a stack on which functions deposit values and from which they read them.

The individual data sources are searched through a general-purpose query predicate that acts as the language interface with specialized modules that deal with the different sources with which the system deal. These modules are connected to the navigation language through a standard interface, so that it is (relatively) easy to create new modules for integrating new data sources.

To facilitate the creation of the results, we keep the partial construction in a structure distinct from the stack: this will somewhat hurt the simplicity of the language, but it will avoid the programmer the task of shifting partial results and intermediate data around in the same structure. Moreover, the stack is a "first in, last out" structure, whereas the results are often created in the same order in which they will have to appear at the end of the execution. Therefore, we have introduced a list of output token and an instruction (print) that adds tokens to the list. Keeping with the spirit of the language, the list is used only for the creation of the final result, and it doesn't take part in the computation. In particular, there is no function to read the values of the list once they have been written there. In this sense, the list is very much analogous to the output stream of a programming language like C. The program is read from an input tape (the input is formalized as a tape because the interpreter has the possibility of jumping back and forth in it and therefore the input is not a stream) at a location maintained by the program counter "pc". The controller of the execution is composed of an interpreter which reads the next instruction and calls the appropriate function, and a scheduler, which determines which instruction must be read next. In addition to these elements, the controller has a list of auxiliary local storage, which are used to build the parameter lists for function calls.

The stack controller manages the stack, which is, as we mentioned before, the only storage used by the programs. The print controller implements the function "print" and writes on the output stream. The query controller implements the function "query" and connects to the appropriate source drivers.

3 SYNTAX OF THE LANGUAGE

The fundamental construct of the language is the *block*, which is simply a list of instructions such that, if the first instruction of the block is executed, then all the instructions of the block are (unless, of course, the program stops on an error). Some instructions may include other blocks, but blocks are not part of the same block to which their containing instruction belongs. Consider, for instance, the following sequence:

```
A
if B then {
    C
}
else {
    D
}
E
```

There are three blocks in this sequence: one is composed of the instructions A, B, and if...then...else, the second is composed of the instruction C, and the third is composed of the instruction D. Note that there is no guarantee that all instructions will be executed (as a matter of fact, we are here guaranteed that *not* all the instructions will be executed); nevertheless, if one instruction in any of the blocks above is executed all will be.

A block is composed of commands, of the following categories:

stack commands: push, pop, swap, empty, dup, rotr, rotl, depth;

execution control: stop, if, while, repeat;

navigation: traverse, query;

ouput: print, clear, stack.

arithmetic: +, -, *, /, mod;

string: sub, cat;

boolean: =, !=, >, <, >=, <=, and, or, not;

All operators work on the first n elements of the stack, where n is the arity of the operator. For instance, the operation + removes the first two elements from the stack and replaces them with their sum. As in the C programming language, boolean values are represented by numbers: zero means false, and every other number means true. So, the operator > removes the first two elements of the stack and replaces them with 1 if the top of the stack is greater than the second element, with 0 otherwise. This solution makes operators very fast and easy to implement. One possible drawback is that mathematical and logic expression are written in *reverse Polish notation,* and tend to be at first cryptic to people use to the common notation (unless they happen to be familiar with Hewlett & Packard pocket calculators). We consider this drawback a minor one, as, due to its application area, programs written in this language will seldom contain complex mathematical or logical expression. The syntax of the language is shown in table 1. Most of the syntax is self-explanatory; note however the difference between the operators *while* and *repeat:* the latter reads a number n from the top of the stack and executes its inner block n times, that is, the operator itself reads the stack only once. The *while* operator reads the top of the stack, then executes the block, then reads the stack again, and so on until a 0 (false) is read. An empty stack will result in an error. Writing a constant (number or string) is equivalent of doing a "push" of that constant. So, "2 3 - print" is a simple program that computes 2-3 and prints it.

<program>	::=	<block>
<block>	::=	<statement> | <statement> | <block>
<statement>	::=	<stack> | <control> | <output> | <aritmethic>
		| <stringop> | <navigation> | <boolean>
<stack>	::=	push <constant> | pop | top | swap | swap <number> |
		| rotr | rotl | depth | empty | dup | <constant>
<control>	::=	stop | if { <block> } | while { <block> }
		repeat { <block> }
<output>	::=	clear | print
<arithmetic>	::=	+ | - | * | / | mod
<stringop>	::=	cat | sub
<boolean>	::=	> | < | = | ! = | >= | <= | and | or | not
<constant>	::=	<string> | <number> | NIL
<navigation>	::=	traverse | query <auxlist>
<auxlist>	::=	<pair> | <pair>, <auxlist>
<pair>	::=	<string> : <string> | <string> : <number>

Table 1: The grammar of the laguage.

The <auxlist> in the query function is meant to give special directives to the query drivers: it consist of a list of "name: value" pairs, separated by commas (see the following sections for details).
Example:
Consider the *Niebla* query given in the introduction. We assume that there is a driver for the bookseller's main page that, given a page, puts 1 on the stack if the page is a list of titles, and puts 0 if the page is a book page. We'll call this driver "verify."

The second driver takes a page with a list of titles, and returns a list of the links of that page. This driver is called "titlelist". The third driver takes a book page and returns the title of the book; this driver is called "title". The final driver ("pagequery") takes a book page, and a name (such as "price") and returns the corresponding entry , or NIL if the entry was not found. Note that we have considered all these as separate drives for the sake of exemplification: in reality all these functions will be executed by sending different queries to the same driver. In this example, however, we want to avoid the details of the query process.

4 SEMANTICS

The stack organization of the language makes the definition of the semantics relatively easy, since the state is constituted just of the contents of the stack. Here we define the denotational semantics of the language. The stack is represented as a list, with the list operations "::" (insert an element at the head of a list), and @ (join of two lists). When convenient, we will represent a list using the notation $[u_1, \ldots, u_n]$ rather than the more cumbersome $u_1 :: u_2 :: \cdots :: u_n :: []$. Also, if u us a url, we will denote with $\rightarrow u$ the content of the page pointed by u. The semantics of an operator is, in this scheme, a funcion from stacks to stacks.

The semantics of the stacks operators is shown in table 2. The semantics of the control operations is shown

Operator	Semantics
push u	$[\![\text{push}u]\!](\mathfrak{s}) = u :: \mathfrak{s}$
pop	$[\![\text{pop}]\!](u :: \mathfrak{s}) = \mathfrak{s}$
empty	$[\![\text{empty}]\!](\mathfrak{s}) = \begin{cases} 1 :: [] & \text{if } \mathfrak{s} = [] \\ 0 :: \mathfrak{s} & \text{otherwise} \end{cases}$
swap	$[\![\text{swap}]\!]([n, u_1, u_2, \ldots, u_{n-1}, u_n, \ldots, u_k]) = \begin{cases} [u_n, u_2, \ldots, u_{n-1}, u_1, \ldots, u_k] & \text{if } k \geq n \\ \text{ERROR} & \text{otherwise} \end{cases}$
rotr	$[\![\text{rotr}]\!]([n, u_1, u_2, \ldots, u_n, \ldots, u_k]) = \begin{cases} [u_2, u_3, \ldots, u_n, u_1, \ldots, u_k] & \text{if } k \geq n \\ \text{ERROR} & \text{otherwise} \end{cases}$
rotl	$[\![\text{rotl}]\!]([n, u_1, u_2, \ldots, u_n, \ldots, u_k]) = \begin{cases} [u_n, u_1, \ldots, u_{n-1}, \ldots, u_k] & \text{if } k \geq n \\ \text{ERROR} & \text{otherwise} \end{cases}$
depth	$[\![\text{depth}]\!]([u_1, \ldots, u_n]) = [n, u_1, \ldots, u_n]$
dup	$[\![\text{dup}]\!](u :: \mathfrak{s}) = u :: u :: \mathfrak{s}$

Table 2: Denotational semantics of the stack operators.

in table 3 Note that, as it is usually the case, the denotational semantics of the while loop is recursive, and it

Operator	Semantics
sequence	$[\![AB]\!](\mathfrak{s}) = [\![B]\!]([\![A]\!](\mathfrak{s}))$
stop	$[\![\text{stop}]\!](\mathfrak{s}) = \text{ERROR}$
if	$[\![\text{if \{ A \} else \{ B \}}]\!](u :: \mathfrak{s}) = \begin{cases} [\![A]\!](\mathfrak{s}) & \text{if } u \neq 0 \\ [\![B]\!](\mathfrak{s}) & \text{otherwise} \end{cases}$
	$[\![\text{if \{ A \}}]\!](u :: \mathfrak{s}) = \begin{cases} [\![A]\!](\mathfrak{s}) & \text{if } u \neq 0 \\ \mathfrak{s} & \text{otherwise} \end{cases}$
while	$[\![\text{while \{ A \}}]\!](u :: \mathfrak{s}) = \begin{cases} [\![\text{while \{ A \}}]\!]([\![A]\!](\mathfrak{s})) & \text{if } u \neq 0 \\ \mathfrak{s} & \text{otherwise} \end{cases}$
repeat	$[\![\text{repeat \{ A \} }]\!](n :: \mathfrak{s}) = [\![A]\!]^n(\mathfrak{s})$

Table 3: Denotational semantics of the control operators.

must be defined as a suitable fixpoint of the semantics of the block A. The technique is well known (Tennent, 1991), and we will not go into the issue here.

The denotational semantics of the arithmetic, logic, boolean, and output operators is shown in figure 4. Finally, the semantics of the navigation operators is shown in figure 5

Note that the arithmetic operators have all the same semantics, with trivial variations, so we have reported it as the semantics of the generic operation $\oplus$, and similarly for the logic operatos and the operation ρ. In the semantics of the output operator, we have introduced the output stream as part of the state. Strinctly speaking,

Operator	Semantics
arithmetic	$[\![\oplus]\!](u :: v :: \mathfrak{s}) = (u \oplus v) :: \mathfrak{s}$
logic	$[\![\rho]\!](u :: v :: \mathfrak{s}) = \begin{cases} 1 :: \mathfrak{s} & \text{if } u\rho v \\ 0 :: \mathfrak{s} & \text{otherwise} \end{cases}$
boolean	$[\![\text{and}]\!](u :: v :: \mathfrak{s}) = \begin{cases} 1 :: \mathfrak{s} & \text{if } u \neq 0 \wedge v \neq 0 \\ 0 :: \mathfrak{s} & \text{otherwise} \end{cases}$
	$[\![\text{or}]\!](u :: v :: \mathfrak{s}) = \begin{cases} 1 :: \mathfrak{s} & \text{if } u \neq 0 \vee v \neq 0 \\ 0 :: \mathfrak{s} & \text{otherwise} \end{cases}$
	$[\![\text{not}]\!](u :: \mathfrak{s}) = \begin{cases} 1 :: \mathfrak{s} & \text{if } u = 0 \\ 0 :: \mathfrak{s} & \text{otherwise} \end{cases}$
string	$[\![\text{cat}]\!](''a_1a_2\cdots a_n'' :: ''b_1b_2\cdots b_n'' :: \mathfrak{s}) = ''a_1a_2\cdots a_nb_1b_2\cdots b_n'' :: \mathfrak{s}$
	$[\![\text{sub}]\!](k :: ''a_1a_2\cdots a_n'' :: \mathfrak{s}) = ''a_1a_2\cdots a_p'' :: \mathfrak{s} \quad (p = \min\{k, n\})$
print	$[\![\text{print}]\!](u :: \mathfrak{s}, o) = (\mathfrak{s}, o@[u]);$
clear	$[\![\text{print}]\!](\mathfrak{s}, o) = (\mathfrak{s}, []);$

Table 4: Denotational semantics of the arithmetic, logic, output, and string operators.

Operator	Semantics
traverse	$[\![\text{traverse}]\!](u :: \mathfrak{s}) = (\rightarrow u) :: \mathfrak{s}$
query	$[\![\text{query}]\!]([u_1, \ldots, u_n]@\mathfrak{s}) = r :: \mathfrak{s}$
	(stackdepth = n; r is the query result)

Table 5: Denotational semantics of the navigation operators.

we should have introduced the output stream as part of the state of all the operators, and have all the operators leave it untouched. This would have made the notation heavier without adding anything to the rigour of the presentation, so we have omitted the output stream from the state except when explicitly needed.

5 LANGUAGE EXECUTION

The language parser creates an intermediate structure that an interpreter reads in order to execute a program. Each block is represented as a list of instructions, and blocks nested into if, while, or repeat statements are represented as additional lists attached to the instruction from which they depend. Each instruction is contained in a list element whose skeleton is as follows:

```
typedef struct _statm {
     char *name;
     par  *parlist;              /* list of the parameters (if any) */
     struct _statm *trueblock;   /* nested block (for while, if, repeat) */
     struct _statm *falseblock;  /* nested block (for the else part of the if)  */
     struct _statm *next;        /* next instruction in the block */
} statement;
```

Figure 1 shows a sample program and its representation for execution. A function `exec(statement *s)` executes a block that begins with the given statement. The function is called recursively to execute the blocks contained in the while, repeat, and if commands.

5.1 Query Interfaces

An important component of the language is the `query` statement, which represents the interface between the navigation language and the systems that interrogate data sources (individual web pages, data bases, file

```
push 1
dup
-
if {
     push ”it doesn’t work!”
     print
} else {
     ”yes! 1-1 = 0”
     print
}
```

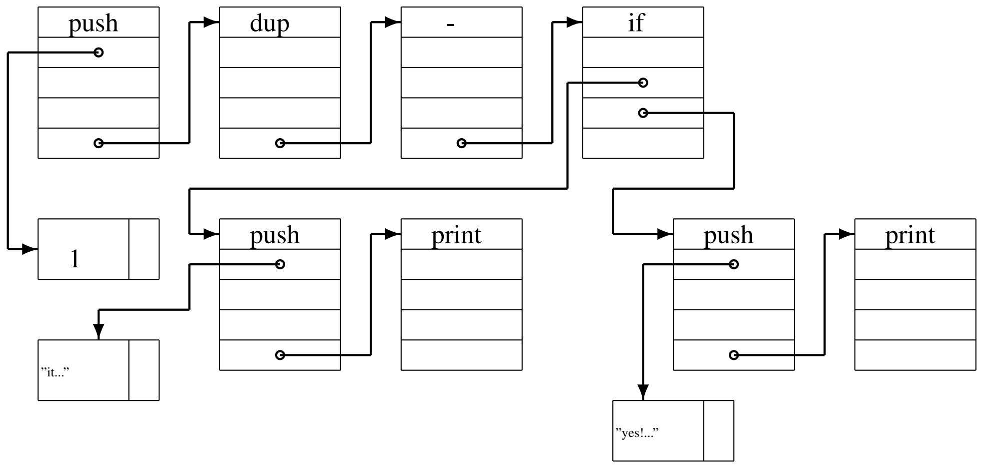

Figure 1: A simple program and the structure created for its execution.

```
push ”query text”
push ”http://page.to.query.edu”
query source:wquery
        stackdepth:2
        traverse:true
```

```
push ”query text”
push ”http://page.to.query.edu”
traverse
query source:wquery
        stackdepth:2
        traverse:false
```

Table 6: Example of use of the query instruction: two equivalent code fragments.

systems, etc.). The language is extensible in this component, in that there is a standard communication interface between the interpreter and the query subsystems that makes it easy to add new query capabilities.

The query instruction in the language is of the form

$$\text{query } \alpha_1 : \beta_1 \cdots \alpha_n : \beta_n$$

where the $\alpha_i : \beta_i$'s are pairs composed of the name of a parameter (α_i) and its value (β_i). The first such pair must always be of the form `source: <name>`, and is used by the interpreter to determine which source driver should be called. The second parameter may be of the form `stackdepth: <number>`; if this parameter is present, the interpreter will remove the first `<number>` elements from the stack to pass them to the driver as a query specification; the default number of elements to be removed from the stack is one.

The other pairs are passed to the drivers without interpretation, and are used to set specific parameters of each driver.

Upon execution of the query the interpreter will place on the stack a list of results and (at the top) the number of results written.

Example:

Suppose that a driver “wquery” will query a web page in two possible ways: if the parameter “traverse” is set to true, the driver will take a query, a link to a page, traverse the link an dexecute the query. If the parameter is false, the driver will take a query, the text of a page, and run that query on the page. Then the two code fragments of table 5.1 are equivalent.

6 CONCLUSIONS

In this paper we have presented a language to “thread together,” so to speak, systems that analyze web pages or access data bases, allowing the creation of complete web site and data base navigation systems to be used in data integration systems. Our requirements were to define a langauge that would be simple to interpret to avoid the addition of a heavy infrastructure to the web page analysis systems that, quite often, tend to be rather heavy themselves. At the same time the language was to be powerful enough to incorporate decisions based on the status of partial results and cycles. We found a good compromise with the implementation of a stack language.

Thanks to the simplicity of the language, the intepreter could easily be implemented in Java in a self-contained manner, without having to rely on any additional library and software beyond the basic Java libraries. This makes the installation of the system very easy modulo, of course, the possible complication of installing the individual query subsystems. We see this simplicity as an important feature of any software development, and a contribution to fighting the increased complexity and multiple dependencies of internet software, often a primary cause of teh poor quality of internet software.

The language is extensible in that additional query subsystems can be added with moderate development effort through a standard interface.

This paper presents the first version of the language, which will undoubtedly be extended in the future. The most important extension that we are planning is the possibility of defining functions.

References

Adelberg, B. (1998). Nodose: a tool for semi-automatically extracting structured and unstructured data from text documents. In *Proceedings of the SIGACT/SIGART/SIGMOD International conference on management of data*, (pp. 283–294). ACM Press.

Altschul, S., Madden, T., Schaffer, A., Zhang, J., Zhang, Z., Miller, W., & Lipman, D. (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, *25*(17), 3389–3402.

Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., & Widom, J. (1994). The TSIMMIS project: Integration of heterogeneous information sources. In *In Proceedings of the 100th IPSJ Anniversary Meeting, Tokyo, Japan*.

Crescenzi, V., Mecca, & G. Merialdo, P. (2001). Roadrunner: towards automatic data extraction from large web sites. In *Proceedings of the International Conference on very large data bases*, (pp. 109–118). ACM PRess.

da Silva, A. S., Evangelista Filha, I. M., Laender, A. H. F., & Embley, D. W. (2002). Representing and querying semistructured web data using nested tables with structural variants. In *Proceedings of ER 2002: the 21st international conference on conceptual modeling, Tampere, Finland*, (pp. 135–151).

Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., & Widom., J. (1995). Integrating and accessing heterogeneous information sources in TSIMMIS. In *Proceedings of the AAAI Symposium on Information Gathering, Stanford, California*.

Rather, E. D., Colburn, D. R., & Moore, C. H. (1993). The evolution of forth. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, (pp. 177–199)., New York, NY, USA. ACM Press.

Santini, S. & Gupta, A. (2004). Modeling functional data sources as relations. In et al., P. A. (Ed.), *Proceedings of ER 2004, the 23rd international conference on conceptual modeling*, volume 3288 of *Lecture Notes in Computer Sceince*, (pp. 55–68). Springer-Verlag.

Tennent, R. D. (1991). *Semantics of programming languages*. Prentice Hall.

Wiederhold, G. (1992). Mediators in the architecture of future information systems. In M. N. Huhns & M. P. Singh (Eds.), *Readings in Agents* (pp. 185–196). San Francisco, CA, USA: Morgan Kaufmann.

Zadorozhny, V., Raschid, L., Vidal, M.-E., Urhan, T., & Bright, L. (2002). Efficient evaluation of queries in a mediator for websources. In *Proceedings of ACM SIGMOD*. ACM Press.