A Common LISP Hypermedia Server

John C. Mallery (JCMA@AI.MIT.EDU)
Artificial Intelligence Laboratory
Massachusetts Institute of Technology

5 May, 1994

Abstract: A World−Wide Web (WWW) server was implemented in Common LISP in order to facilitate exploratory programming in the global hypermedia domain and to provide access to complex research programs, particularly artificial intelligence systems. The server was initially used to provide interfaces for document retrieval and for email servers. More advanced applications include interfaces to systems for inductive rule learning and natural−language question answering. Continuing research seeks to more fully generalize automatic form−processing techniques developed for email servers to operate seamlessly over the Web. The conclusions argue that presentation−based interfaces and more sophisticated form processing should be moved into the clients in order to reduce the load on servers and provide more advanced interaction models for users.

1. Overview

- **Motivations**: The Introduction discusses the motivations for developing a Common LISP HTTP server.
- **Server**: The design ideas embodied in the server are overviewed in the sections Server Features, Classes of HTTP URLs, Exporting URLs, and Computed URL Example.
- **Applications**: Screen snapshots show variety of applications, ranging from document retrieval and form−processing email servers to artificial intelligence applications, such as automatic rule induction and natural− language question answering.
- **Programming Language**: Several sections provide background on Common LISP, its object system, and the Common LISP Interface Manager.
- **Related Systems**: A system for distributing documents is briefly reviewed in the COMLINK section because it uses the Common LISP HTTP server as a WWW interface and because it embodies form−processing technology useful for the Web.
- **Recommendations**: The Conclusions recap an argument for extending HTTP/HTML+ form−processing with design concepts tested in the Common LISP Interface Manager and the COMLINK system.

2. Introduction

A World−Wide Web (WWW) server was implemented in Common LISP (see the FAQs, the Association of Lisp Users and the ANSI specification) in order to facilitate exploratory programming in the global hypermedia domain and to provide access to complex research systems over the Web. The general motivation for developing this server was to provide a computational tool that would strengthen the link between the artificial intelligence researchers and the distributed hypermedia community. As the amount of information available over the *World−Wide Web* grows, it will become increasingly necessary to deploy intelligent methods to store, retrieve, analyze, filter, and present information. At the same time, high−productivity programming tools employed by AI researchers will become increasingly relevant for testing new ideas in servers before incorporating them into standards−based clients. A Common LISP HTTP server provides a bridge that allows AI researchers to plug their systems into the WWW as it affords developers of distributed hypermedia standards a vehicle through which they can import effective and relevant technologies. In this spirit, the conclusion proposes that HTML+ borrow ideas from the CLIM Presentation System in order to modernize the user interfaces in clients, to upgrade the interaction paradigm, and to achieve a number of efficiency gains for servers.

More specific motivations for this work in the context of the Intelligent Information Infrastructure Project at M.I.T. were to develop an HTTP server meeting these criteria:

- **High−Productivity Programming:** Since all of our research systems are programmed in Common LISP and are incrementally extensible to meet research requirements, the server should be fully accessible from LISP and able to evolve at the same pace.
- **Multiple Transport Media:** We wanted to generalize our existing technology, and so, to develop a specification and control framework that works seamlessly over email and HTTP.
- **Automatic Form−Processing:** Since we already had an automatic form−processing substrate developed in an email server application, the server should be able to apply the same technology over the Web.
- **Dynamic HTML+ Generation:** Since our research emphasizes tailoring output to meet user requirements, we needed HTML+ operators that our research systems could use to generate HTML dynamically.

The current server is excellent for rapid−prototyping precisely because it builds on Common LISP and provides a fine−grained vocabulary of operators which are easily combined and modified according to evolving application requirements and draft protocol standards. The server itself is an example of rapid prototyping as its 2700 lines of LISP were written from scratch and debugged by one person in about two weeks. A high−profile demonstration at the end of the period was well−received.

More recently, an authoring tool for email−based forms was generalized to emit HTML+ (Renaud, 1994). This graphical tool is now used to author forms that can work over email and WWW. The authoring tool is important for the generalization of email forms across the Web because form processing in email servers performs automatic retries when user input is syntactically or semantically incorrect. To implement automatic retries over the WWW, it is necessary to dynamically generate retry forms to explain the problem, incorporate all correct answers from the original form, and allow the user to correct the erroneous answers. Computing form retries requires datastructures beyond those normally associated with scripting languages. [Other research (Houh, Lindblad, & Wetherall, 1994) arrives at similar conclusions concerning the limitations of scripting languages for more advanced, dynamic WWW applications.] In general, more advanced form−based applications will need the richer datastructures offered by full−featured programming languages like Common LISP.

3. Initial Applications

The Common LISP HTTP Server implementation was driven initially by the desire to provide WWW access to email servers and associated document or survey databases, which were built on the COMLINK System. Development of a native server offered WWW access not just for these specific applications, but also, for the COMLINK substrate systems in general. Thus, any application developed on top of COMLINK would automatically have the World−Wide Web as a user interface.

The initial applications included:

- **Document Search**: Retrieve government documents via a the taxonomy of categories used to distribute them over email (see search index and form interfaces).
- **Standing Search URLs:** Find the latest, topically−focused documents with convenient anchors that prespecify criteria for dynamic searches (see publications for example).
- **Public Access Email:** Send correspondence to elected government officials through a forms interface to an email server.
- **Electronic Publications:** Subscribe to electronic publications through a forms interface to an email server.

All these applications handle invalid or incoherent user input by returning dynamically generated

HTML+ that explains the problem and guides the user's efforts to resubmit.

4. Advanced Applications

Two artificial intelligence applications were fielded by the end of the development period:

- **Rule Induction:** Learn *if−then* rules over the Web (Renaud, 1994) using an advanced datahandling environment (Mallery, 1994) and a sophisticated rule induction module (Unseld & Mallery, 1991). See snapshots of the experiment and several rules learned.

- **Natural−Language Question Answering:** Ask English questions about the M.I.T. Artificial Intelligence Laboratory to a natural language system (Katz, 1990; Katz, 1994). Ask an English question to see how this works. See snapshots of asking a question and viewing the answer.

Integration with both of these large LISP systems was possible within several of hours of work each.

5. Server Features

The main implemented features of the server include:

- **All Major Content Types Served:** The server not only handles text and HTML but also serves images (xbitmap, gif, jpeg), audio, video (mpeg), and application (postscript).

- **Major Methods Handled:** The server fully supports the GET and HEAD operations, and handles the POST method to receive user values in form processing.

- **Object−Oriented Implementation:** The Common LISP Object System (CLOS) is used to implement key server components. This allows modular programming based on generic operations defined on URL and server object.

- **Typed Condition Handling:** The Common LISP condition system is used to signal various conditions, or errors, that may occur. Condition classes are defined for all HTTP response codes and are arranged hierarchically. When most conditions are signalled, execution transfers to a handler at the top−level of the server. This handler applies a generic response function to the condition object that transmits the appropriate response code to the client. Conditions can specialize their method for the response function in order to *customize behavior*.

- **HTML+ Authoring Functions:** A series of functions and macros provide a vocabulary of HTML+ operations, at least as complete as support of HTML+ by reference clients. (See the example code.) The authoring functions serve a number of purposes:

  - **Defined Interface:** Provide a functional interface for generating HTML+ under program control, whether on the fly to respond to a user or batch to write static files.

  - **More Compact Vocabulary:** Quite often authoring abstractions condense a range of HTML+ tags into a single parameterized function. For example, an enumeration macro and a text style macro span the various available tags. (See the example code.)

  - **Input Types:** Tags for HTML+ input fields were consolidated in an abstraction analogous to CLIM presentation types. Implemented as CLOS classes, *input types* support generic methods for *accepting* input from the user by writing the appropriate HTML+ tags during form creation and interpreting returns during form submission. A single set of arguments for the generic function standardizes parameters across all input types (audio, date, float, image, integer−range, multi−line−text, multiple−choice, radio−button, reset−button, select−choices, string, submit−button, url).

  - **Query Abstraction**: The query abstraction from COMLINK is imposed on HTML+/HTTP form processing to facilitate asking the user atomic questions and responding to user answers. Although it simplifies accepting single−valued input types, the query concept has an even bigger impact for input types returning multiple values: Here, the input type system takes care to *reintegrate* what HTML+ treats as *separate* queries into a *single* sequence−valued return. This conserves code abstraction by preventing every response function for a form from having to reinvent the same program cliche.

- **Smooth Code Evolution:** An abstraction layer separates code that generates HTML+ from the raw tags. This allows the authoring functions to be updated when the underlying specification changes, and thereby obviating the need to update *any* calling functions.

- **Network Security:** The server implements subnet security based on client IP address.

6. Classes of HTTP URLs

Uniform Resource Locators are implemented as CLOS objects, to which URL strings are mapped upon receipt. Although the URL implementation handles all URL schemes defined by the URL specification, only the base classes for HTTP scheme will be described here:

- **Path URLs** are analogous to a directory in a file system because they contain no specific URL name and extension type. These can map to a physical file if desired, or just to a directory hierarchy.
- **Object URLs** are specific documents with a name and usually an extension that indicates the kind of data available in the URL.
- **Search URLs** specify searches for particular databases and may have an associated HTML document that describes the search. Parameterized export types are defined for kinds of database interfaced.
- **Form URLs** are just like an object URL with extension HTML, except that they take a response function to be called whenever a client submits (POSTs) form values.
- **Computed URLs** are much like form URLs except that they fully compute their response to a GET operation. (See the example code.)

7. Exporting URLs

URLs become accessible via the server once they have been exported with the EXPORT function. The export function always takes an external URL name and a *export type*, which defines the computation used to serve the data denoted by the URL. In most cases, some additional arguments are provided, depending on export type. For example, when data resides in a file, the physical pathname where the data is stored is passed as an argument to export. Similarly, when a database is searched, the database is passed as an export argument along with the URL. The following export types are defined:

- **Text Types**: Text−File, HTML−File, Text−Directory (computes a display of links to substructure), LISP−Directory, HTML−Form, HTML−Computed.
- **Other Types**: X−Bitmap−Image, GIF−Image, JPEG−Image, Basic−Audio, MPEG−Video, Postcript−File.
- **Redirect**: When a URL has moved to another host, the server signals an HTTP redirect that tells the client where to find the document.

8. Computed URL Example

In this example, the client receives an HTML+ page that the server computes by listing a remote directory and repackaging the file names in a friendlier format. Below, we see the Common LISP definition for the response function that is invoked when a client sends the server a GET for the computed URL.

```lisp
;; Export the computed URL and provide its response function.
(export-url "http://clinton.ai.mit.edu/radio-addresses.html"
            :html-computed
            #'compute-clinton-audio-page)

;; Writes HTML+ directly to the client over STREAM, which is a live
;; HTTP connection to the client.
(defmethod compute-clinton-audio-page ((url url:http-computed-url) stream)
  (flet ;; Abstract the internal function that writes an entry.
    ((write-pathname (pathname stream)
       (let ((url (url:pathname-ftp-url-string pathname))
             (friendly-date (uu-radio-address-anchor-date pathname)))
         (when friendly-date
           ;; Encapsulate text within an enumeration entry.
           (html+:enumerating-item (stream)
             ;; Change the text style to bold.
             (html+:with-rendition (:bold :stream stream)
               ;; Write an HTML+ anchor spec using the friendly name.
               (html+:note-anchor
                 friendly-date :reference url :stream stream)))))))
    ;; Main body of method.
    (let ((directory-list (www-utils:ftp-directory-info
                            *saturday-radio-addresses-audio-directory*
                            "anonymous"
                            (www-utils:user-mail-address))))
      (if directory-list
          ;; Inform the client that we're winning and returning HTML+.
          (with-successful-response (stream :html)
            ;; Provide the header info that goes in the HTML+.
            (html+:with-document-preamble (:stream stream)
              (html+:declare-base-reference url :stream stream)
              (html+:declare-title "Saturday Radio Addresses" :stream stream))
            ;; Use the body environment.
            (html+:with-document-body (:stream stream)
              (html+:with-section-heading
                ((html+:image
                   "http://clinton.ai.mit.edu/icons/sound-icon.xbm"
                   "The President's Saturday Radio Addresses to the Nation"
                   :stream nil))
                (html+:new-paragraph :stream stream)
                ;; Provide the instructions for the WWW page.
                (write-string "By clicking on a date, you can listen to
                               the audio for a Saturday radio address
                               by President Clinton." stream)
                (html+:horizontal-line :stream stream)
                (html+:new-paragraph :stream stream)
                ;; Enter an environment for enumerating items.
                (html+:with-enumeration (stream :itemize)
                  ;; Map over files and write them as bullets.
                  (loop for (pathname) in directory-list
                        do (write-pathname pathname stream))))))
          ;; Signal that the server cannot connect to the remote host.
          (error "Unable to connect to UU.NET to get file list.")))))
```

9. Common LISP

The HTTP server is written in Common LISP (see Winston and Horn (1989), Steele (1990), the FAQs, the Association of Lisp Users and the Draft ANSI specification) to provide a user−extensible environment for interactive multimedia over WWW. Because the goal is to develop a fine−grained vocabulary of operators that programs and software developers share, Common LISP is one of the best choices available today. Once operators are abstracted, programs and users call them, instead of (re−)writing them again. This fine−grained abstraction yields high productivity because:

- **No Code Duplication:** No effort is wasted writing again code that already exists because programs are shared at a very fine level of abstraction (*e.g.*, individual functions, variables).

- **Code Evolution:** Well abstracted code is easily evolved as protocols or requirements change.
- **Dynamic Linking**: Incremental compilation speeds development of large programs and facilitates evolutionary programming.
- **Powerful Abstractions:** Function−compositional languages, such as Common LISP, allow, and even encourage, programmers to build up a vocabulary of high−level operators that are recycled to solve similar domain problems. Indeed, writing an application becomes a process of developing a vocabulary of operators to specify the desired functional process.[See Abelson and Sussman (1985) for a detailed introduction to abstraction in a functional language.]

A high productivity environment reduces the work required to develop new packages and makes most efficient use of the scarce time of scientists. Generalizing the individual case to a community of researchers, sharing this kind of environment and each other's extensions maximizes the rate at which the community searches mechanism space for problem solutions. Moreover, well−abstracted code joins with self−documenting capabilities in the HTTP server to make it an effective pedagogical environment; everything is readily available for inspection, modification, and experimentation.

10. The Common LISP Object System

Common LISP embeds a native object−oriented programming language, the Common LISP Object System (CLOS). All significant datastructures in the HTTP server are implemented as CLOS objects (Bobrow *et al.*, 1988; Keene, 1989). CLOS distinguishes classes and instance objects. The behavior of instance objects is defined by associated classes, which can inherit functionality and state variables from multiple superior classes. Instances can have local state, held by instance variables, and they can have methods, or operations, which they support. Generic operations constitute a protocol that instances of different classes all support with their own, possibly different methods. CLOS supports *multimethods* that can dispatch based on the types of multiple arguments. A pervasive object−oriented implementation helps enforce abstraction and modularity as it enhances flexibility and code sharing.

11. The Common LISP Interface Manager

Modern Common LISP provides a high−level window system tool, the Common LISP Interface Manager (CLIM) (see the specification, (Rao, *et al*, 1991). Written in Common LISP, CLIM runs on a variety of workstations, implementing its machine−independent abstractions with native window systems and mimicking their look and feel. By introducing high−level abstractions for specifying window interfaces, CLIM makes it easier and quicker to define window interfaces. Instead of requiring many months, sophisticated user interfaces can be written in several days.

Beyond general window system abstractions, the automatic form processing in COMLINK forms relies heavily on an important CLIM abstraction: *the presentation system* controls how the user specifies or perceives arbitrary LISP objects. Each *presentation type* has a method to *present* (display) an object on the computer screen, and a method to *accept* (receive input) an object from the user, whether via the keyboard, the mouse, or other input device.

In general, presentation types specialize built−in or constructed LISP types. *Presentation translators* can be defined to convert from the LISP object associated with one presentation type to another. The combination of type subsumption and presentation translation allows CLIM to resolve equivalences, and thus, accept input more intelligently.

The present and accept methods are like little generators and parsers specialized to a presentation type. They can perform a simple operation, such as accepting an integer between 1 and 10, or they can execute a very complex computation. For example, in one research system (Mallery, 1991), nodes in semantic network representing natural language text can be presented through a sentence generator so that the user sees them as generated sentences or sentence fragments. Similarly, sentences or paragraphs can be accepted via a presentation type that calls a full natural language system to parse and represent the text. Thus, present and accept methods execute LISP code that can perform an arbitrary translation between what the user sees or what the user inputs and the internal datastructures manipulated by programs.

Once defined, these presentation types mediate all data entry and display, and thus, users perceive only the external, user–friendly representation, and never with the internal representation. Such dynamic, bidirectional translation allows users to *directly* manipulate program data (viewing it in translated form) and actually simplifies the program model presented to the user.

12. The COMLINK System

Over the past year and a half, the author developed a system that performs sophisticated operations over email, routes documents, and automatically surveys users. The COMLINK system uses a transaction–controlled, persistent–object database to represent users, hosts, mailing lists, document categories, documents, and more. For the purposes of this paper, three aspects of this system are relevant.

Document Routing and Retrieval

The COMLINK system supports *document universes* with associated taxonomies of categories. As documents are distributed through document universes, they are archived and indexed by some categories. Users can subscribe to a publications stream or retrieve documents by means of boolean combinations of categories. When retrieving documents, users may also provide temporal and quantity constraints to further circumscribe the documents found. The document retrieval and publication code that stands behind the initial WWW applications uses this technology as a back–end.

Automatic Form Processing

The COMLINK system interacts with users via textual forms exchanged in email, as well as conventional ‘‘subject line’’ commands, much like those found in standard listservers. Email servers are associated with a *command interface* that provides access to all the forms and subject line commands.

Forms are composed of a series of queries. In addition to a question or instructions, a *query* has an associated CLIM presentation type that presents any default value and parses any new value supplied by the user. Forms are written to a stream by calling the WRITE–FORM function with a set of value bindings for each query of the form. As WRITE–FORM calls the generic operation to present each query, the queries present themselves by calling the PRESENT method for their presentation type.

Forms are parsed by finding queries and converting the textual input associated with each query into its internal representation. The basic procedure for parsing a query is:

- **Scan**: Locate the query between special delimiters.
- **Intern**: Convert the query name into the query object associated with the form.
- **Accept**: Parse the textual representation that follows the delimited query name by calling the interned query’s accept method, which in turn calls the accept method for the associated presentation type.
- **Handle Errors**: If the query value fails to conform to requirements of the presentation type, signal the condition so that the user can be asked to respecify failing queries. Typed error objects carry the information needed to provide the user with user–friendly explanations about what was wrong and how to correct it.

When query values are successfully parsed, the form’s response function is applied to the parsed query values to perform the computation associated with the form. If there are query parsing errors, the system returns to the user a form with all the correct values defaulted and an explanation about how to correct the failing queries for successful resubmission.

Graphical Form Authoring Tool

A graphical form authoring tool was written by Renaud (1994) for the COMLINK system. Coded in CLIM, the interface defines meta–level abstractions for forms, queries, and presentation types that allow users to define automatic surveys and forms without having to write LISP code. At the same time, the data structures are abstracted in a way that forms can be defined dynamically under program control. For the set of presentation types previously defined for COMLINK, Renaud defined new presentation and accept multimethods that dispatch on the HTML+ *presentation view*. This means that the same presentation type, which already worked for the *email view*, could now operate for the *HTML+ view*, displaying itself in HTML+ and accepting its input with the HTML+ form−processing facilities.

Unlike HTML+ form input types, CLIM has a rich basic set of presentation types, which are *routinely* extended by application programs. After pairing up the presentation types that match between CLIM and HTML+ form input types, Renaud defined the appropriate cliches to accept input from the user for CLIM presentation types via HTML+ cliches. Once this small correspondence set was exhausted, the rest of the task reduced to accepting a text string from the client and applying the CLIM accept method to parse the input from the string. Unfortunately, all the computation required to parse input strings must remain on the server because there is *currently* no defined way to tell clients how to accept the input or check its validity.

13. Conclusions

An immediate goal for the Common LISP server is to develop seamless form processing over both email and the Web. At first, CLIM presentation types will be evaluated by the server as it checks the values returned in HTML+ forms for validity and parses them into the appropriate internal representations. After a period of experimentation, during which a good set of presentation types will be identified, it should be possible to propose a set of presentation types that clients can handle without undue difficulty. If servers could transfer definitions to clients, the presentation types available to remote applications could be extended or refreshed dynamically. This would require extensions to the HTTP protocol and agreement on a safe language(s) for transferring presentation parsers and generators to clients. By relocating the main responsibility for validating user input, considerable computational load and unnecessary connections can be offloaded from servers and distributed among clients.

The Common LISP HTTP server makes it possible to interface complex LISP systems often found in artificial intelligence applications to the World−Wide Web. Although the immediate goal of the server was to serve as a research tool for a project on intelligent information routing at M.I.T., the server can be useful for many other members of the world−wide LISP community and allow them to participate in the explosion of interesting WWW applications. At the same time, the rapid prototyping features of Common LISP can now become available to the WWW research community so that they can more quickly mock up and test out new ideas.

14. Availability

The full source code for the Common LISP HTTP server is available for anonymous FTP (View the source code). The server currently runs on Symbolics LISP Machines under Genera 8.3. Ports to Common LISP running on other machines and operating systems should be available during the summer of 1994.

15. Acknowledgments

This paper and the server was improved by comments from Mark Nahabedian, Benjamin Renaud, Howard Shrobe, and Robert Thau. Benjamin Renaud implemented the WWW interface to the rule learning system. Boris Katz adapted his natural language system to allow WWW access. This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the M.I.T. Artificial Intelligence Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under contract number MDA972−93−1−003N7.

16. References

Abelson, H. and G. J. Sussman, *The Structure and Interpretation of Computer Programs*, Cambridge: M.I.T. Press, 1985.

Bobrow, D., *et al.*, ''A Common LISP Object System Specification: X3J13 Document 88−002R,'' *SIGPLAN Notices*, 23, September, 1988.

Houh, H., Lindblad, C. & Wetherall, D., ''Active Pages: Intelligent Nodes on the World Wide Web,'' *Proceedings of the First International Conference on the World−Wide Web*, Geneva: CERN, 1994.

Katz, B., ''Using English for Indexing and Retrieving,'' in *Artificial Intelligence at M.I.T.: Expanding Frontiers*, edited by P. H. Winston with S. A. Shellard, Cambridge: M.I.T. Press, vol. 1, ch. 6, 1990.

Katz, B., ''Using Natural Language Annotations in the Voyager Information System,'' *International Aerospace Congress IAC'94,* Moscow, Russia, 1994.

Mallery, J. C., ''Semantic Content Analysis: A New Methodology for The RELATUS Natural Language Environment,'' in *Artificial Intelligence and International Politics,* V. Hudson, ed., Boulder: Westview Press, 1991.

Mallery, J. C., ''Beyond Correlation: Bringing Artificial Intelligence to Event Data,'' *International Interactions*, forthcoming, 1994.

Keene, S. E., *Object−Oriented Programming in Common LISP: A Programmer's Guide to CLOS,* Reading: Addison−Wesley, 1989.

Renaud, B., *Global Data Sharing and Analysis: A Networked Hypermedia Approach*, Cambridge: Political Science Department, Massachusetts Institute of Technology, S.B. Thesis, May 1994.

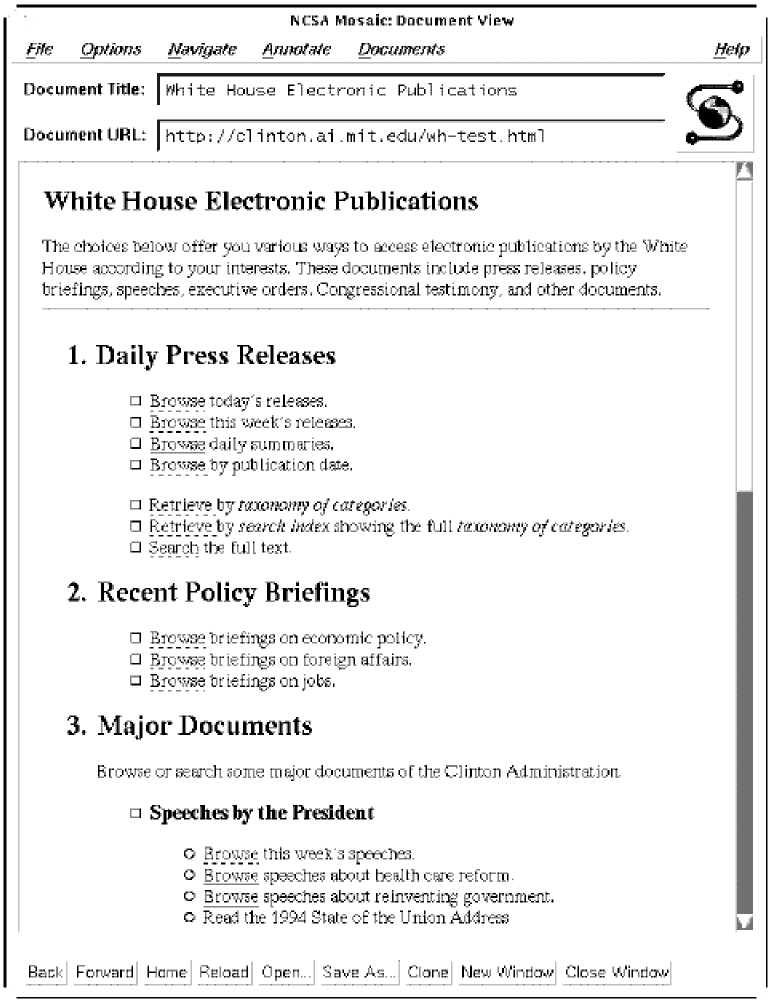
Rao, R., York, W. M., and Doughty, D., ''A Guided Tour of the Common LISP Interface Manager,'' *LISP Pointers,* 4(1991).

Steele, G. L., *Common LISP: The Language*, Bedford: Digital Press, 1990.

Winston, P. H., and Horn, B. K. P. H., *LISP*, Reading: Addison−Wesley, 1989. This is a good introduction to Common LISP with special reference to artificial intelligence.

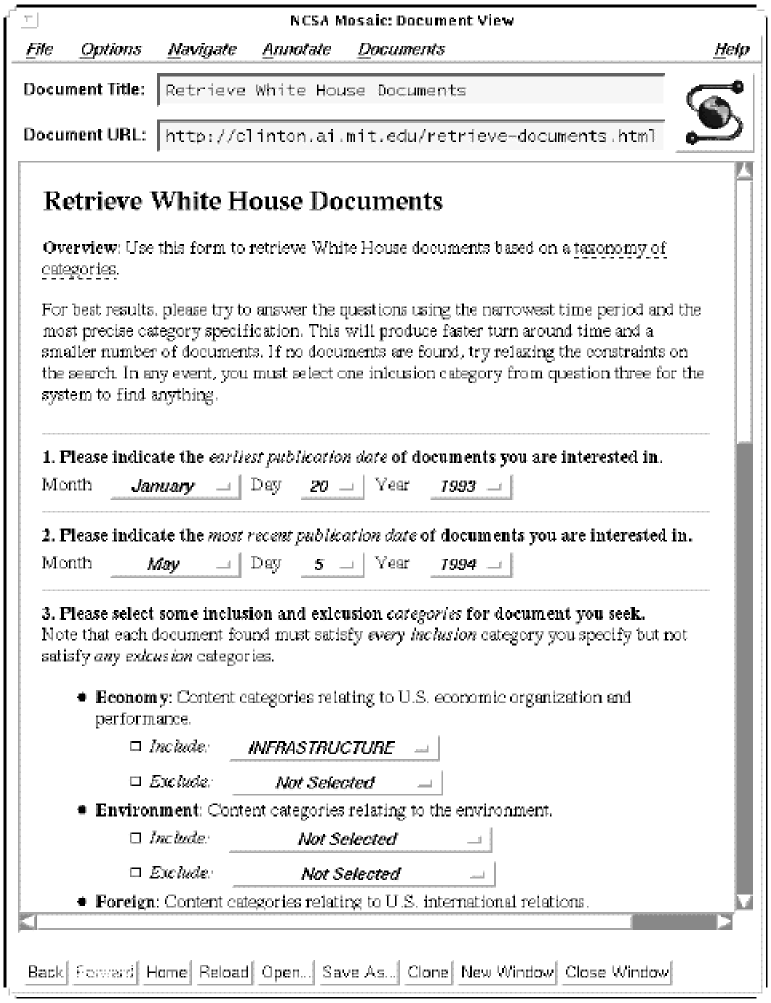
Unseld, S., and Mallery, J. C., ''Interaction Detection in Complex Datamodels,'' Cambridge: M.I.T. A.I. Laboratory Memo 1298, November, 1991.

17. Screen Snapshots



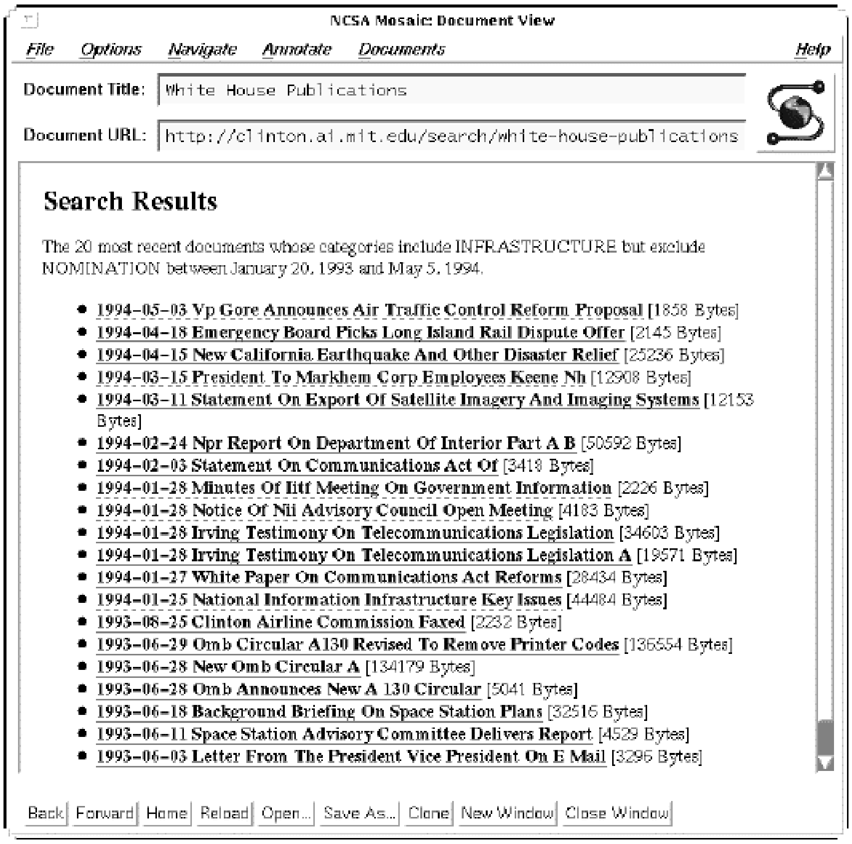
Various Ways to Search and Retrieve Documents

The page for retrieving documents offers a variety of way to access White House Electronic Publications. It includes a number of standing search URLs that automatically run a search when the user clicks on them.



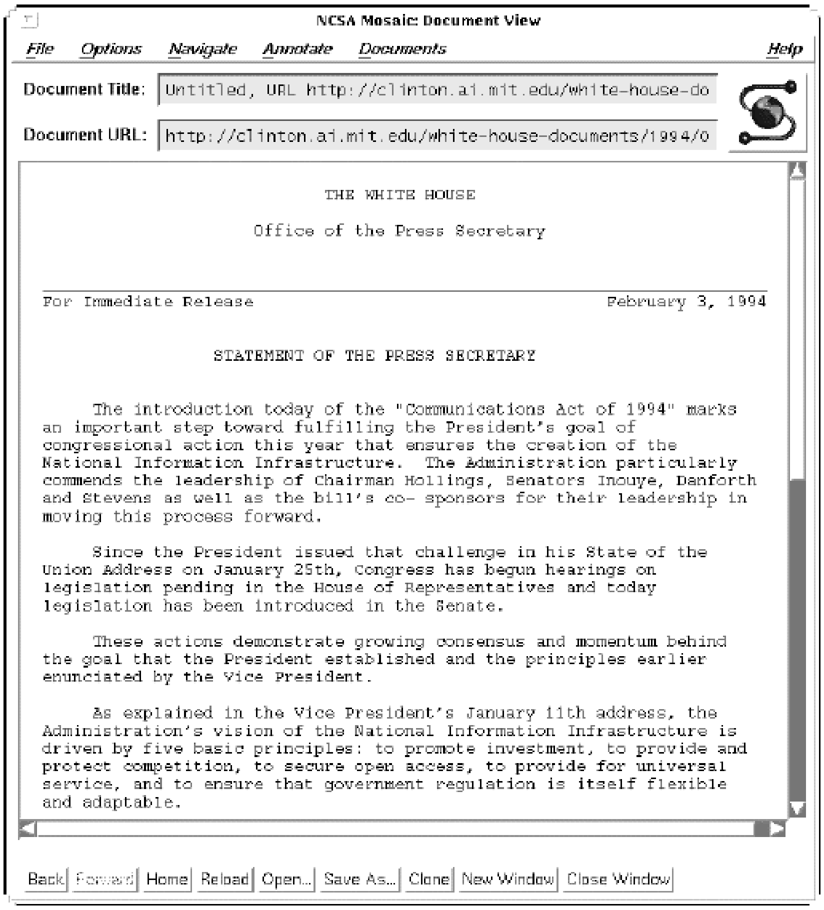
Forms−Based Search Interface

A forms−based interface to boolean keyword search is not only easier to use, but also makes it impossible for users to mispecify searches syntactically, if not semantically.



Search Results

After running a successful search, the user sees this page of results, which are generated on the fly from document records in a persistent–object database accessible to the server.



THE WHITE HOUSE

Office of the Press Secretary

For Immediate Release February 3, 1994

STATEMENT OF THE PRESS SECRETARY

The introduction today of the "Communications Act of 1994" marks an important step toward fulfilling the President's goal of congressional action this year that ensures the creation of the National Information Infrastructure. The Administration particularly commends the leadership of Chairman Hollings, Senators Inouye, Danforth and Stevens as well as the bill's co- sponsors for their leadership in moving this process forward.

Since the President issued that challenge in his State of the Union Address on January 25th, Congress has begun hearings on legislation pending in the House of Representatives and today legislation has been introduced in the Senate.

These actions demonstrate growing consensus and momentum behind the goal that the President established and the principles earlier enunciated by the Vice President.

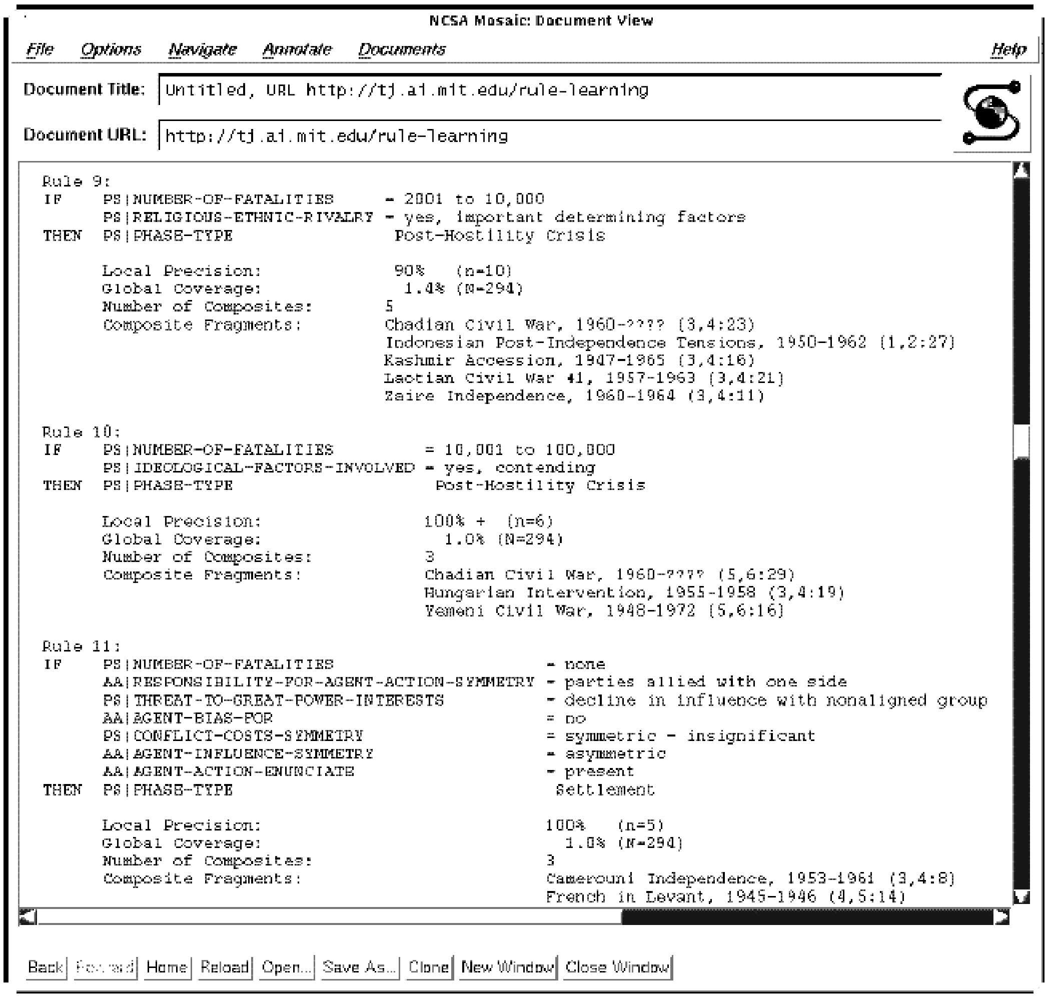
As explained in the Vice President's January 11th address, the Administration's vision of the National Information Infrastructure is driven by five basic principles: to promote investment, to provide and protect competition, to secure open access, to provide for universal service, and to ensure that government regulation is itself flexible and adaptable.

Viewing a Document

Clicking on a title of brings up the document.

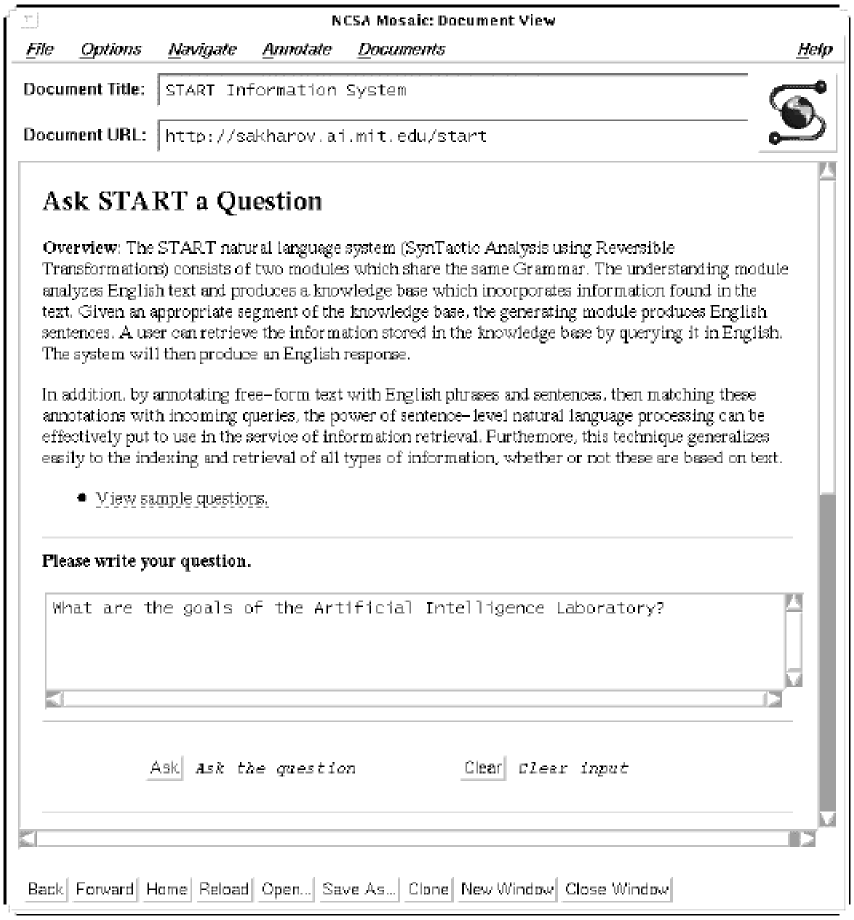
Specifying a Rule Learning Experiment

This page allows the user learn if–then rules about international conflicts by selecting a collection of cases and specifying a control matrix to guide the rule learning experiment. The control matrix tells the system which independent variables to examine as it looks for rules that predict dependent variable values.



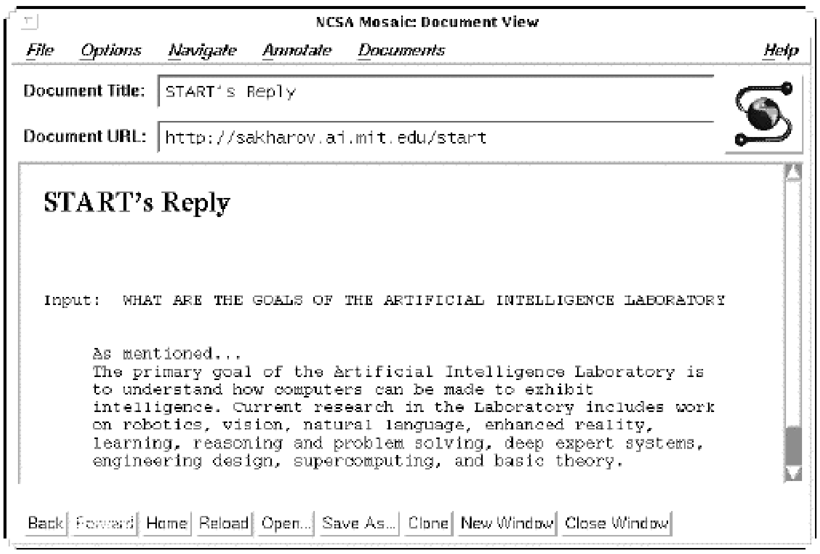
Several Rules Learned over the Web

The rule learner first builds a decision tree and then converts the branches of the tree into a series of rules. This page shows several rules found during the experiment specified in the previous screen.



Asking a Question to a Natural Language System

The user asks a question about the M.I.T. Artificial Intelligence Laboratory.



Viewing the Answer to a Question

After parsing and representing the question, the START system finds the answer and returns it on a Web page.