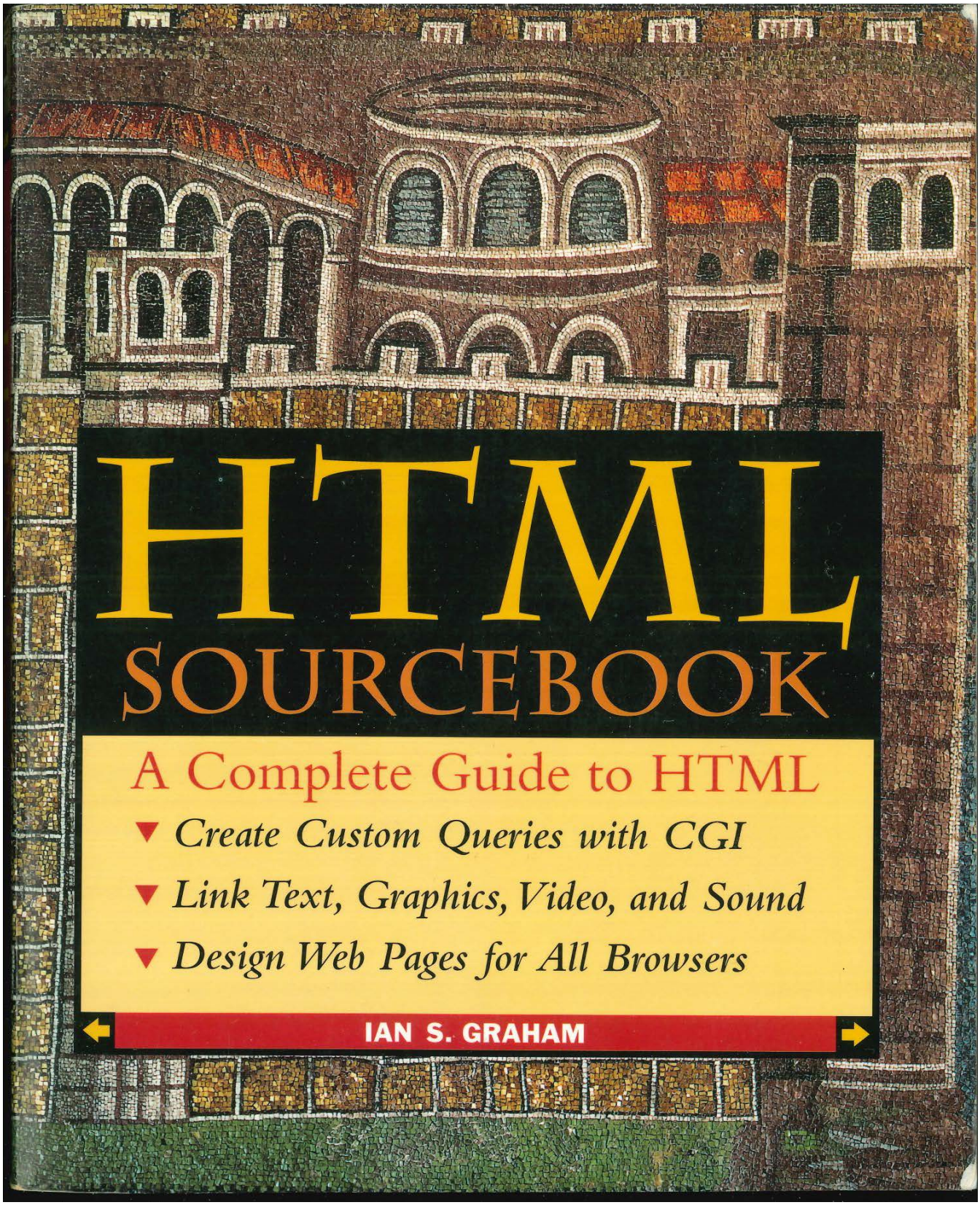


FreeWheel, Ex. 1222, Part 1 of 2

THE HTML SOURCEBOOK

Ian S. Graham



John Wiley & Sons, Inc.

New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Katherine Schowalter
Editor: Paul Farrell
Assistant Editor: Allison Roarty
Managing Editor: Frank Grazioli
Interior Design & Composition: Benchmark Productions, Inc.

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc. is aware of a claim, the product names appear in Initial Capital or all CAPITAL letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This text is printed on acid-free paper.

Copyright © 1995 Ian S. Graham

Published by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional service. If legal advice or other expert assistance is required, the services of a competent professional person should be sought.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without that permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permission Department, John Wiley & Sons, Inc.

Library of Congress Cataloging-in-Publication Data:
ISBN 0 471-11849-4

Printed in the United States of America

10 9 8 7 6 5 4

CONTENTS

Introduction ix
Uniform Resource Locators x
The Hypertext Transfer Protocol x
The Hypertext Markup Language xi
Overview of the Book xii
Acknowledgments xvi
Dedication xvi

Chapter 1 **Introduction to the Hypertext Markup Language** 1
Basic Outline of the HyperText Markup Language 4
Example 1: A Simple HTML Document 5
HTML Element 7
Head and Title Elements 8
Body Element 9
Highlighting Elements 10
Paragraphs 11
Unordered Lists 13
Horizontal Rule Element 14
Lessons from Example 1 14
Example 2: Images and Hypertext Links 15
The Example Document 17
Example Document Rendered 17
Anchors 21
Partial URLs 22
Creating Links 26
Hypertext Links: The Good, the Bad, and the Ugly 27
Lessons from Example 2 31
Example 3: Home Pages 32
Title and Heading 37
Text Portion 39
Organization 39
Icons 40
Uniform Resource Locators 41
Lessons from Example 3 43
Example 4: Collections of Hypertext Documents 44
Pre Element 48
Organization 51
Archiving 52
Lessons from Example 4 55
Example 5: Images, Movies, and Sound Files 56
Linking Large Images 56
Lessons from Example 5 61
Example 6: Fill-In Forms 61
Form Element 62
Form Restrictions 66
Lessons from Example 6 67
References 68

Chapter 2 **HTML in Detail** 71
Introduction to HTML 71
Allowed Characters in HTML documents 73
Special Characters 74
Comments in HTML Documents 75
HTML as a MIME Type 76
HTML Elements and Markup Tags 76
Case-Sensitivity 78
Empty Elements 79
Element Nesting 80
Unknown Elements or Attributes 80
Overall Document Structure 80
Hypertext Markup Language Specification: Element by Element 81
Key to This Section 82
Head Elements 86
Body Elements 94
List Elements 113
Character-Related Elements 134
Character Highlighting Elements 138
Logical Highlighting 138
Physical Highlighting 145
HTML+ Elements 148
References 159

Chapter 3 **Uniform Resource Locators (URLs)** 161
Allowed Characters in URLs 162
Disallowed Characters 163
Special Characters 164
Example of a Uniform Resource Locator 165
1. Protocol 165
2. Address and Port Number 165
3. Resource Location 166
Partial URLs 167
URL Specifications 168
Ftp URLs 169
Gopher URLs 171
HTTP URLs 174
Mailto URLs 177
News URLs 177
Telnet/tn3270/rlogin URLs 178
WAIS URLs 178
File URLs 179
References 179

Chapter 4 **The HTTP Protocol and the Common Gateway Interface** 181
The HTTP Protocol 182
Example HTTP Client-Server Sessions 184
User Authentication, Data Encryption, and Access Control 201

HTTP Methods and Headers Reference 204
HTTP Methods 205
HTTP Request Headers 205
HTTP Response Headers 207
The Common Gateway Interface 208
Sending Data from the Client to the Server 208
Sending Data to the Gateway Program from the Server 209
Returning Data from the Gateway Program to the Server 210
The POST Method and Standard Input 225
Security Considerations 229
References 230

Chapter 5 HTML and CGI Tools 231
Images in HTML Documents 232
X-Bitmap Images 232
X-Pixelmaps 232
GIF Image Files 233
Reducing Image File Size: The Color Map 233
Reducing Image Size: Rescaling Images 235
Transparent GIF Images 235
Active Images 238
Creating the Image Database 242
Icon Archive Sites 247
Client-Side Executable Programs 247
Sending the Script to the Client 248
Configuring the Client 250
Security Issues 250
Server-Side Document Includes 253
Include Command Format 253
Example of Server-Side Includes 258
HTML Utility Programs 261
Dtd2html 262
HTML Table Converter 262
Hypermail 263
MHonArc: Mail to HTML Archive 263
Table of Contents Generator 264
TreeLink 264
CGI Utility Functions 265
CGI Email Handler 265
CGI Feedback Form 265
Determining Client Software 266
Convert UNIX Man Pages to HTML 266
Processing Queries and FORM Packages 267
Database CGI Gateway Programs 269
WAIS Gateways 269
WWWWAIS 272
Gateways to Structured Query Language Databases 272
Macintosh Search Tools: TR-WWW 276

CGI Archive Sites 276
Database Gateway References 277

Chapter 6 **HTML Editors and Document Translators** 279
HTML Editors 280
Simple Text Editors (Macintosh, PC, UNIX) 281
Alpha (Macintosh) 282
BBEdit HTML Extensions (Macintosh) 282
CU_HTML.DOT (PC-Word for Windows) 283
Emacs (UNIX, PC) 284
GT_HTML.DOT (PC-Word for Windows) 285
HoTMetaL (PC-Windows, UNIX) 285
HTML Assistant (PC-Windows) 286
HTMLed (PC-Windows) 287
HTML.edit (Macintosh) 288
HTML Editor (Macintosh) 288
HTML for Word 2.0 (PC-Word for Windows) 288
Htmltext (UNIX) 289
NextStep HTML-Editor (NeXT) 289
S.H.E. (Macintosh) 290
TkHTML (UNIX) 290
TkWWW (UNIX) 291
Document Translators and Converters 291
Asc2html 293
Charconv 293
Cyberleaf 294
FasTag 294
Frame2html 295
HLPDK 295
Hyperlatex 296
Latex2html 296
Mif2html 297
Miftran 297
Mm2html 297
Ms2html.pl 298
Ps2html 298
RosettaMan 299
Rtftohtml 299
RTFTOHTM 300
Scribe2html 300
Striphtml 300
TagWrite 301
Tex2rtf 301
Texi2html 301
WPTOHTML 302
Wp2x 302
WebMaker 302
HTML Verifiers 303

Sgmls ... 303
HTML Validation Site ... 306
Link Verifiers ... 306
Linkcheck ... 306
Verify_Links ... 307

Chapter 7 **Web Browsers and Helper Applications** ... 309
Accessing Software ... 310
PC Platform Browsers ... 310
MSDOS Browser: DosLynx ... 311
Windows and OS/2 Browsers ... 312
Helper Applications ... 315
Macintosh Platform Browsers ... 317
MacWeb ... 318
Macintosh Helper Applications ... 319
UNIX and VAX/VMS Platform Browsers ... 321
Batch mode browser (UNIX) ... 321
Chimera (UNIX) ... 321
Emacs-w3 mode (UNIX, VMS, others) ... 322
LineMode Browser (UNIX, VMS) ... 323
Lynx (UNIX, VMS) ... 323
MidasWWW (UNIX, VMS) ... 324
Mosaic for X-Windows (UNIX) ... 325
Mosaic (TueV) 2.4.2 ... 326
Quadralay GWHIS Browser (UNIX) ... 326
Rashty VMS Client (VMS) ... 326
Tkwww (UNIX) ... 327
ViolaWWW (UNIX) ... 327
UNIX Helper Packages ... 328
NeXT Platform Browsers ... 329
The CERN NeXT Browser ... 330
Omniweb ... 330
Amiga Browser: AMosaic ... 331
Coming Attractions ... 331
IBM OS/2 Browser: Web Explorer ... 331
MicroMind SlipKnot ... 331
Netscape Communications Corp. ... 331

Chapter 8 **HTTP Servers and Server Utilities** ... 333
Basic Server Issues ... 333
UNIX Servers ... 334
VMS Servers ... 335
Windows NT or OS/2 ... 335
Windows or Macintosh ... 336
Behind a Firewall? ... 336
List of Server Software ... 336
CERN HTTP Server (UNIX, VMS) ... 337
CL-HTTP (Symbolics LISP Machines) ... 337

CMS HTTPD (VM/CMS) ... 338
DECthread HTTP Server (VAX/VMS) ... 339
GN Gopher/HTTP Server (UNIX) ... 339
GWHIS HTTP Server (UNIX) ... 340
HTTPS (Windows NT) ... 340
Jungle (UNIX) ... 341
MacHTTP (Macintosh) ... 341
NCSA HTTPD (UNIX) ... 342
OS2HTTPD (OS/2) ... 343
Plexus (UNIX) ... 343
SerWeb (Windows 3.1) ... 344
Web4Ham (Windows 3.1) ... 344
WinHTTPD: NCSA HTTPD for Windows (Windows 3.1) ... 344
Coming Attractions ... 345
BASIS WEBserver ... 345
MDMA: Multithreaded Daemon for Multimedia Access ... 345
Netscape Netsite ... 346
Server Support Programs ... 346
Getstats ... 346
Gwstat ... 347
Webstat ... 347
Wusage ... 348
Wwwstat ... 348

Chapter 9 Real-World Examples ... 349
Electronic E-Print Servers ... 349
OncoLink ... 354
Introduction ... 355
The OncoLink Implementation ... 356
Use of OncoLink ... 357
Views of the Solar System ... 364
Background ... 364
HTML Issues ... 367
Summary ... 369
NetBoy—Choice of an Online Generation ... 369
San Francisco Reservations' World Wide Web Page ... 373
Introduction ... 373
Why on the World Wide Web? ... 374
Designing the WWW Page ... 375
Final Notes on SFR ... 381

Appendix A The ISO Latin-1 Character Set ... 383
Appendix B Multipurpose Internet Mail Extensions (MIME) ... 389
Appendix C Finding Software Using Archie ... 395
Appendix D Listening at a TCP/IP Port ... 401
Glossary ... 403
Index ... 409

INTRODUCTION

It is fair to say that the World Wide Web project has taken the Internet by storm, confounding Internet skeptics and supporters alike. In hindsight, however, the reasons are obvious. The World Wide Web (WWW) model makes accessing the Internet easy, both to consumers of Internet-based information and to information providers. It's downright easy to distribute information via the Web, and just plain fun to go out and look for it. It is no surprise that World Wide Web utilities have grown, in less than three years, to be the most popular tools on the Internet.

A tool may be easy to use but often requires skill and training to be used well. This is certainly true of the tools involved in preparing and distributing information via hypertext documents and Internet hypertext servers. Preparing well-designed, useful, and reliable resources requires an in-depth understanding of how the tools that deliver these resources work and how to use them *well*. The intention of this book is to provide this understanding. Assuming that you are familiar with traditional Internet resources, such as FTP, telnet, electronic mail, and Gopher, there are essentially three new components to consider:

1. *Uniform Resource Locators*, or *URLs*, which are the scheme by which Internet resources are addressed in the WWW.
2. The *HyperText Transfer Protocol (HTTP)* and HTTP *client-server* interactions. HTTP servers are designed specifically to distribute hypertext documents, and you must know how they work to take advantage of their powerful features.
3. The *HyperText Markup Language*, or *HTML*. This is the markup language with which World Wide Web hypertext documents are written, and is what allows you to create hypertext links, fill-in forms, and clickable images. Writing good HTML documents involves both technical issues (proper construction of the HTML document) and design issues (ensuring that the information content is clearly presented to the user).

The goal of this book is to explain all three of these issues and to give you the tools to develop your own high-quality World Wide Web products. The remainder of this introduction looks briefly at these three components and explains their basic features. This is followed by an outline of the book and some suggestions as to how to best approach the text and examples.

UNIFORM RESOURCE LOCATORS

Uniform Resource Locators, or URLs, are a naming scheme for specifying how and where to find any Internet server resource, such as from Gopher, FTP or WAIS servers. For example, the URL that references the file ***macweb.zip*** in the directory ***/pub/web/browsers*** on the anonymous FTP server ***ftp.bozo.net*** is simply:

```
ftp://ftp.bozo.net/pub/web/browsers/macweb.zip
```

WWW hypertext documents use URLs to reference other hypertext resources.

THE HYPERTEXT TRANSFER PROTOCOL

The *HyperText Transfer Protocol*, or HTTP, is a new Internet protocol designed expressly for the rapid distribution of hypertext documents. Like other Internet tools, such as FTP, WAIS, or Gopher, HTTP is a *client-server* protocol. In the client-server model, a *client* program running on the user's machine sends a message requesting service to a *server* program running on another machine on the Internet. The server responds to the request by sending a message back to the client. In exchanging these messages, the client and server use a well-understood *protocol*. FTP, WAIS, and Gopher are other examples of Internet client-server protocols, all of which are accessible to a World Wide Web browser. However, the HTTP protocol is designed expressly for hypertext document delivery, so most of your communication will be with HTTP servers.

At the simplest level, HTTP servers act much like anonymous FTP servers, delivering files when clients request them. However, HTTP servers support additional important features:

- The ability to return to the client not just files, but also information generated by programs running on the server
- The ability to take data sent from the client and pass this information on to other programs on the server for further processing

These special server-side programs are called *gateway* programs, because they usually act as a gateway between the HTTP server and other local resources, such as databases. Just as an FTP server can access many files, an HTTP server can access many different gateway programs; in both cases, you can specify which resource (file or program) you want through a URL.

The interaction between the server and these gateway programs is governed by the *Common Gateway Interface (CGI)* specifications. Using the CGI specifications, a programmer can easily write simple programs or scripts to process user queries, interrogate databases, make images that respond to mouse clicks, and so on.

THE HYPERTEXT MARKUP LANGUAGE

The *HyperText Markup Language*, or *HTML*, is the language used to prepare hypertext documents. These are the documents you distribute on the World Wide Web and are what your human clients actually see. HTML contains commands, called *tags*, to mark text as headings, paragraphs, lists, quotations, emphasized, and so on. It also has tags for including images within the documents, for including fill-in forms that accept user input, and, most importantly, for including hypertext links connecting the document being read to other documents or Internet resources, such as WAIS databases and anonymous FTP sites. It is this last feature that allows you to click on a string of highlighted text and access a new document, an image, or a movie file from a computer thousands of miles away. And how does the HTML document specify where this document is? Through a URL, which is included in the HTML markup instructions and which is used by your browser to find the designated resource.

What resources can URLs point to? They can be other HTML documents, pictures, sound files, movie files, or even database search engines. They can be on your computer or anywhere on the Internet. They can be accessed from HTTP servers or from FTP, Gopher, WAIS, or other servers. The URL is an immensely flexible scheme and, in combination with HTML, yields an incredibly powerful package for preparing a web of hypertext documents linked to each other and to Internet resources around the world. This image of interlinked resources is, in fact, the vision that gave rise to the name World Wide Web.

OVERVIEW OF THE BOOK

This book is an introduction to HTML, URLs, HTTP, the CGI interface, and the design and preparation of resources for delivery via the World Wide Web. It begins with the HTML language. Almost every resource that you prepare will be presented through an HTML document so that your HTML presentation is your *face* to the world. It is crucial that you know how to write proper HTML, and that you understand the design issues involved in creating good documents, if you are to make a lasting impression on your audience and present your information clearly and concisely. It won't matter if the Internet resources you make available are the best in the world if your presentation of them is badly designed, frustratingly slow, or difficult to follow.

HTML is also an obvious place to start. You can write simple HTML documents and view them with a WWW browser, such as **Mosaic**, **MacWeb**, **lynx**, **Cello**, or **Netscape** without having to worry about CGI programs, HTTP servers, and other advanced features. You can also easily add to your documents URLs pointing to server resources around the world, and get used to how the system works: Browsers understand HTML *hypertext anchors* and the URLs they contain, and they have built-in software to talk to Internet servers using the proper protocols. You can accomplish a lot just by creating a few pages of HTML.

Chapter 1 is an introduction to HTML and to the design issues involved in preparing HTML documents. This nontechnical chapter combines a brief overview of HTML with important aspects of the document design process. The details of the HTML language and more sophisticated client-server issues are left to Chapters 2 and 3.

Design issues are very important in developing good World Wide Web presentations. HTML documents are not like text documents or traditional hypertext presentations, since they are limited by the varied capabilities of browsers and by the speed with which documents can be transported across the Internet. Chapter 1 discusses what this means in practice and gives guidelines for avoiding major HTML authoring mistakes. In most cases, this is done using examples with the important issues being presented in point form so that you can easily extract the main points on first reading.

One point that is emphasized in Chapter 1 and everywhere in the book is the importance of using correct HTML markup constructions when you create your HTML documents. Although HTML is a relatively straightforward language, there are many important rules specifying where tags can be placed. Ensuring that your documents obey these rules is the only way you can guarantee that they will be properly displayed on the many different browsers your clients may use. All too often, writers prepare documents that look wonderful on one browser, but end up looking horrible or even unviewable on others.

Although some general rules for constructing valid HTML are included in Chapter 1, Chapter 2 and the references therein should be used as detailed guides to correct HTML. In particular, Chapter 2 presents a detailed exposition of the HTML language and of the allowed nesting of the different HTML markup instructions. It also explains the syntax and rules for constructing URLs.

Chapter 3 explains the structure and syntax of Uniform Resource Locators (URLs). This is the addressing scheme used in HTML documents to indicate the target of hypertext links.

Of course, HTML is only a beginning. To truly take advantage of the system, you must understand the interaction between WWW client browsers and HTTP servers, and be able to write server-side gateway programs that take advantage of this interaction. Chapter 4 delves into the details of the interaction between WWW clients and HTTP servers and explains the Common Gateway Interface (CGI) specification for writing server-side programs that interface with the HTTP server. Chapter 4 includes simple examples to demonstrate the HTTP protocol and the CGI interface, as well as useful references to sites on the Internet that contain instructional interactive documents. Chapters 2, 3 and 4 are the technical core of this book and should be of use as reference material when writing HTML documents or server CGI programs.

Needless to say, there already are many useful CGI programs available on the Internet, ranging from the **imagemap** program for handling clickable images to sophisticated front-end packages for databases. These and other useful CGI tools are discussed in Chapter 5, which also looks at auxiliary tools useful in developing and organizing HTML documents. For example, there are tools for converting collections of e-mail letters into hypertext archives, or for creating a hypertext "Table of Contents" for large collections of related HTML files. The second half of Chapter 5 discusses these tools and indicates sites where they can be obtained. Almost all of them are available over the Internet, either from anonymous FTP sites or from HTTP servers. URLs are used to indicate the locations of these programs and of additional documentation when available.

Preparing HTML documents can be tedious, since HTML markup tags are complicated text strings that must be included in your text document. HTML codes are not only time consuming to type, but also a common source of error: It is easy to make a mistake typing all those tags! You will not be the first to notice this fact, and many individuals, groups, and com-

panies are actively developing HTML editors to help in the document creation process. Chapter 6 summarizes and briefly describes the various HTML editors available on PC, Macintosh, and UNIX platforms, and explains how to obtain them.

In some cases, you may not want an editor, but would rather be able to convert documents from another format, such as Microsoft's Rich Text Format, FrameMaker's MIF format, LaTeX, and so on into HTML. Chapter 6 includes a summary of several of these packages, including instructions on how they can be obtained. Finally, there are a number or useful tools for *validating* the HTML document for conformity with the language specification and for checking the validity of hypertext links within a document. These tools are listed at the end of Chapter 6.

Chapter 7 is a brief review of the different browsers available for exploring the WWW and for viewing HTML documents. You *should* be interested in how the documents you design will look on browsers other than the one you regularly use. Designing for a single browser is dangerous, since there is a tendency to tailor the HTML for the peculiarities of that particular program. This can lead to HTML documents that look fine on your browser, but horrible on others. It is wise to pick up another browser or two, just to avoid these problems.

Chapter 8 discusses the issues involved in setting up an HTTP server and briefly reviews the commonly available server packages. Again there are many URL references to additional documentation and to locations where server software can be obtained.

Chapter 9, the final chapter in the book, gives some examples of WWW sites containing interesting and well-designed presentations. These sites range from business and entertainment to those devoted to scientific research and education. I asked the creators of these sites to describe the creation process so that you can get a feel for what was involved. I encourage you to visit these sites and see how they work. I can guarantee you will be impressed.

Finally, you should just go out and browse! Writing a book and spouting off one's own ideas of good and bad design is all well and good, but you, as a writer of HTML documents, will appreciate how things look and feel only by going out there and looking and feeling. This book is merely a framework for appreciating what tens of thousands of creative individuals are already doing. So, go and see for yourself!

ACKNOWLEDGMENTS

I would first like to thank my coworkers in the Instructional and Research Computing Group at the University of Toronto: Without their support and encouragement this book would not have been possible. In particular I want to thank John Bradley and Anna Pezacki for giving me the time off to write, probably against their better judgement! I also must especially thank Allen Forsyth and Norman Wilson for critically reading several sections of the manuscript and for removing the "lumpiness" from my early drafts (thank you, Norman, for that elegant image). I also greatly benefitted from the technical expertise of Rudy Ziegler and Terry Jones, who helped me through several problems with image file conversion and computer networking.

Most importantly, I would like to thank my wife, Ann Dean, both for her editing skills and for her unwavering support and patience during the past few months. Without her, this book would not have been possible.

DEDICATION

To Ann

CHAPTER

1

INTRODUCTION TO THE HYPERTEXT MARKUP LANGUAGE

The *HyperText Markup Language*, or HTML, is designed to specify the logical organization of a text document, with important extensions for hypertext links and user interaction. HTML was *not* designed to be the language of a What You See Is What You Get (WYSIWYG) word processor, such as Word or WordPerfect. Instead, HTML requires that you construct documents with sections of text marked as *logical* units, such as titles, paragraphs, or lists, and leave the interpretation of these marked elements up to the browser displaying the document. This model builds enormous flexibility into the system and allows browsers of different abilities to view the same HTML documents. In fact, there are browsers for everything from fancy UNIX graphics computers to plain-text terminals, such as VT-100s or old 8086-based DOS computers. As an example, in viewing the same document, a graphical UNIX browser may present major headings with a large,

perhaps slanted and bold-faced font (since elegant typesetting is possible with graphics displays), while a VT-100 browser may just center the title, using the single available font. Both presentations will look different, but both will reproduce the logical organization that you built in with the HTML tags.

So, what does an HTML document look like? A simple example is shown in Figure 1.1. As you can see, this looks just like a plain text document. In fact, that is exactly what it is. Thus, an HTML document can be prepared with a simple text editor, such as the **NotePad** editor on a Windows PC, **TeachText** on the Macintosh, or **vi** on a UNIX workstation. You don't need a special word processor or fancy HTML editor to create HTML documents.

The difference between an HTML document and a simple text document lies in the HTML markup *tags*. These are the portions of text surrounded by the less than and greater than signs (<. . .>) and are the *instructions* that tell the browser what each part of the document means. For example, the tag `<H1>` indicates the *start* of a heading of level 1, while the `</H1>` tag marks the *end* of a heading of level 1. Thus, the text string

```
<h1> This is the Heading </H1>
```

marks the string "This is the Heading" as a level one heading (there are six possible headings levels, from **H1** to **H6**). Note how a forward slash inside the tag indicates an end tag. The names inside the tags are case-insensitive so that `<h1>` is equivalent to `<H1>`, and `<STRONG>` is equivalent to `<StROng>`. Capitalization is recommended to make the tags stand out.

An HTML document is described as being composed of *elements*. The string

```
<h1> This is the Heading </H1>
```

is then an **H1** element, consisting of an **H1** start tag, the enclosed text, and an **H1** end tag. You will also often see an **H1** element referred to as the *container* of a heading.

Some elements are called *empty* elements, which simply means they do not affect a block of text and do not require an end tag. An example is the **BR** element in Figure 1.1. The tag `<BR>` forces a line break at the location of the tag. This does not affect any enclosed text, since a line break does not *contain* anything.

Sometimes, elements take *attributes*, which are much like variables and are usually assigned *values* that define special characteristics of the element. An example is the **IMG** element, which is used to include an image within an HTML document. An **IMG** element might appear via the tag:

```
<IMG SRC="filename.gif">
```

Here, **SRC** is an attribute of the **IMG** element and is used to specify the name of the image file (actually, a Uniform Resource Locator (URL) *pointing* to the image file) to be included in the document. The attribute name, like the element name itself, is case-insensitive. Thus, the above line could equally well be written as `<iMg src="filename.gif">` or `<iMg SrC="filename.gif">`. However, the value *assigned* to the attribute **SRC** is *case-sensitive*; case-sensitivity is preserved by enclosing the string in quotation marks. As you may have noticed, the **IMG** element, like the **BR** element, is empty, since it merely inserts an image and does not affect a block of text.

Finally, we emphasize that HTML is a *structured* language, which means that there are rules for where element tags can and cannot go. These rules are there to enforce an overall *logical structure* upon the document. Thus, a heading element, such as `<H1>. . . . . </H1>`, can contain text, text marked for emphasis, and a few other text-like elements—but it cannot contain elements that mark other headings or lists. This means that such constructions as

```
<H1><H2>...text ... </H2></H1>
```

are invalid. Obviously, it does not make sense for a heading to *contain* a list or another heading, and the HTML language rules reflect this. In addition, elements cannot overlap—this means that such things as

```
<EM> <H2> ..... text .... </EM> </H2>
```

are illegal and lead to unpredictable behavior when the document is rendered for display. There are many such structural rules; they are given in detail in Chapter 2. This chapter illustrates the most obvious cases.

BASIC OUTLINE OF THE HYPERTEXT MARKUP LANGUAGE

1. HTML documents are divided into *elements*, which are marked by *tags* of the form `<NAME>..... some text ...</NAME>`, where the enclosed text is the content of the element. Some elements do not affect a block of text and are hence called *empty* elements, which do not require end tags.
2. Element names and attributes are case-insensitive. Thus, `<NAME ATTRIBUTE="string">`, `<NamE ATtRiButE="string">`, and `<name attribute="string">` are equivalent. However, the attribute value (here the string "`string`"), when enclosed in quotation marks, is case-sensitive.
3. The placement of elements in a document must obey the HTML rules of nesting, which simply means that there are rules about where elements can and cannot appear. For example, a Heading element, such as H1, cannot contain a list or another heading, but can contain a hypertext anchor. In addition, elements cannot *overlap*. Details of the nesting rules for each HTML element are given in Chapter 2.

EXAMPLE 1: A SIMPLE HTML DOCUMENT

We could go on explaining the language, but it is easier to get a feel for it by looking at some examples; the details of the HTML language are found in Chapter 2 if you are already burning to find out. Figure 1.1 shows a simple but complete HTML document, illustrating the overall structure of a document and some of the simpler markup elements. This document was created using the UNIX **vi** editor and was saved in a file named ***ex1.html***. The ***.html*** filename extension is important, as WWW browsers and HTTP servers understand files with this suffix to be HTML documents as opposed to a *plain* text document, such as e-mail letters or program listings. On PCs, this extension is ***.htm***, since four-letter extensions are not possible. More will be said later about extension names and what they mean.

The rendering of Figure 1.1 by two different browsers is shown in Figures 1.2 and 1.3. All browsers allow you to load and view locally created HTML documents simply by giving the browser the name of the local file. To view the example using the **lynx** browser you type

```
lynx ex1.html
```

at the DOS or UNIX command prompt. With the Macintosh **MacWeb** browser, you just start the program and select the Open... menu from the File pull-down menu at the top of the window, then select the file you wish to open. Similar procedures are possible with all of the graphical browsers.

```
<HTML>
<HEAD>
<TITLE> This is the Title of the Document </TITLE>
</HEAD>
<BODY>
<H1> This is a Heading</H1>

      Hello.  This is not a very exciting document.
I
    bet you were expecting <EM> poetry</EM>, or

some kind of <STRONG> exciting <BR> fact</STRONG> about the Internet and
the World Wide Web.

<P> Sorry.  No such luck.        This document
does
contain examples of HTML markup, for example, here is an "unordered list":
<UL>
   <LI> One item of the list,
   <LI> A second list item  <LI> A third list item that goes on and on and
   on to indicate that the lists can wrap right around the page and still
   be nicely formatted by the browsers.
   <LI> The final item.
</UL>
<p> Lists are exciting. You can also have ordered lists (the items are numbered)
and description lists.
<HR>
<p> And you can draw horizontal lines, which are useful for dividing
sections.
</BODY>
</HTML>
```

Figure 1.1 Contents of the example HTML document ex1.html. The rendering of this document by different browsers is shown in Figures 1.2 and 1.3.

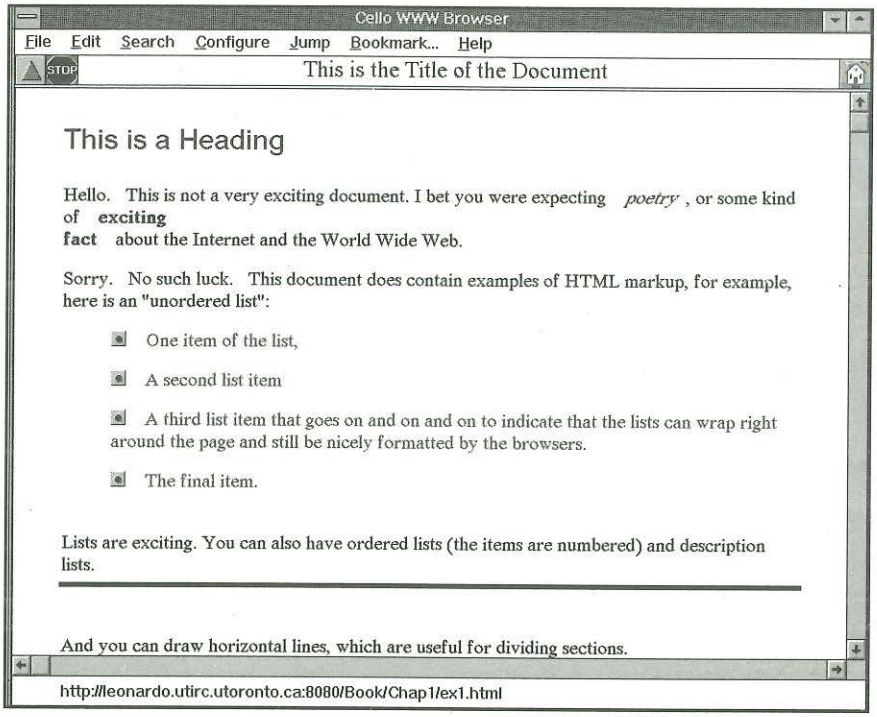


Figure 1.2 Cello rendering of the HTML document ***ex1.html*** (the HTML document is listed in Figure 1.1).

Figures 1.2 and 1.3 show this document as rendered by two different WWW browsers. Figure 1.2 shows what the document looks like, using the graphical **Cello** (for Windows 3.1) browser, while Figure 1.3 shows what you get from the character-based browser **lynx**.

HTML ELEMENT

What do these markup tags and elements mean? Since HTML is a hierarchical language, this is best analyzed by starting from the outside and working in. The outermost element, which encompasses the entire document, is named **HTML**. This element indicates that the enclosed text is an HTML document. This element allows the browser to distinguish between differing versions of the HTML language.

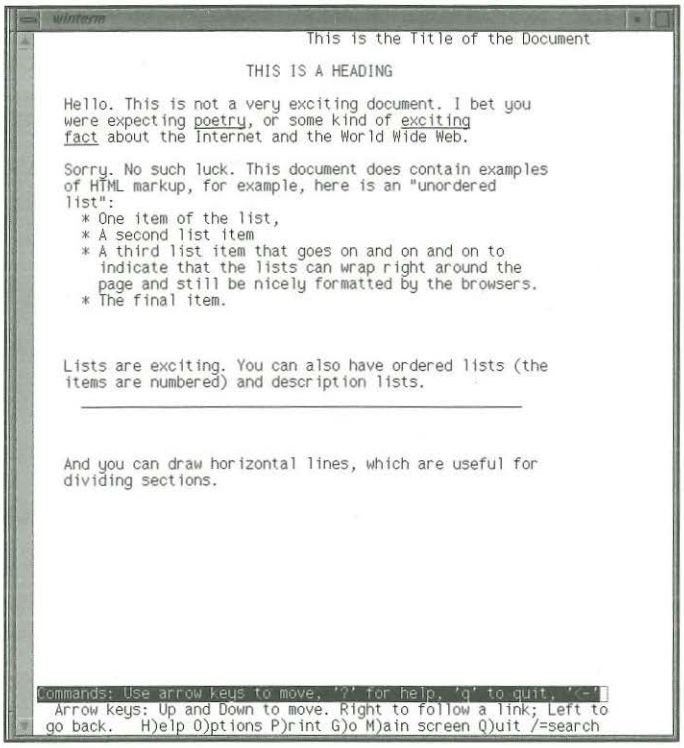


Figure 1.3 Lynx rendering of the HTML document ***ex1.html*** (the HTML document is listed in Figure 1.1).

HEAD AND TITLE ELEMENTS

The next element below the HTML element is named HEAD. The HEAD element is a container for information *about* the document, such as the TITLE element. This information is not displayed as part of the document. Looking at Figures 1.2 and 1.3, you will see that the contents of the TITLE element are displayed apart from the text. With **Cello**, the title is displayed in the frame of the window, while with **lynx**, the name is displayed at the top of the screen and to the right.

Because of the way it is used, you want the TITLE to be both descriptive and short. The TITLE is displayed separately from the text and usually in

a restricted space, such as a window title bar, a small fixed-size text box, or as a single line at the top of a text screen. If a **TITLE** is too long, it simply will not fit. The **TITLE** should also be descriptive of the document, as it is often used as a reference to visited sites—you should be able to guess the content of a document from the **TITLE** alone.

BODY ELEMENT

Parallel with the **HEAD**, lies the **BODY**. This element contains *all* the text and other material that is to be displayed. Notice in Figure 1.1 that all the displayed material lies between the `<BODY>` and `</BODY>` tags. Everything that does not go in the **HEAD** goes here.

Why bother with this **HEAD/BODY** separation? Recall that HTML is designed to organize your document in a logical way. It then makes sense to separate the document itself (the **BODY**) from information *about* the document (the **HEAD**). Currently, there is not much to put in the head, other than the **TITLE**. However, there are several other **HEAD** elements (discussed in Chapter 2) that describe the relationships between a document and other documents. These will be extremely useful for indexing, cataloging, and organizational purposes once they are more commonly used. Indexing programs will be able to quickly read the document **HEAD** (which is always much smaller than the **BODY**) and obtain useful indexing material without having to read the document **BODY**.

The first element in the **BODY** is an **H1** element. **H1** stands for a level-one *heading* element; in HTML, headings come in six levels—**H1** through **H6**—of decreasing rank. The **H1** element means that the enclosed text is a heading and should be rendered to reflect this fact. The **Cello** browser (Figure 1.2) shows the heading

```
<H1> This is a Heading</H1>
```

as a large, bold-faced string of characters, left justified and separated by a wide space from the following text. **Lynx**, on the other hand (Figure 1.3),

shows this as a capitalized text string centered on the page. This example should remind you of the point made at the beginning of this section: that different browsers may render the same elements in very different ways. HTML markup instructions are designed to specify the logical structure of the document far more than the physical layout. Thus, the browser is free to find the best way to display items, such as headings, consistent with its own limitations.

Referring back to Figure 1.1, review the next few lines of text:

```
        Hello.  This is not a very exciting document.
I
     bet you were expecting <EM> poetry</EM>, or

some kind of <STRONG> exciting <BR> fact</STRONG> about the Internet and
the World Wide Web.
```

You will see in Figures 1.2 and 1.3 that these lines have been rendered as a continuous paragraph of text, ignoring the blank lines, extra spaces, and tabs that are actually present. The rendering of an HTML document *ignores* extra spaces, tabs, and blank lines, and effectively treats any combination of them as a single space character. This means that you can use spaces, line breaks, and indentations to organize the logical layout of an HTML document. This will be familiar if you write computer programs or have used typesetting languages, such as TeX, and is equivalent to using spaces and tabs to make a program easier to read.

HIGHLIGHTING ELEMENTS

This sentence contains two new elements: EM for *emphasis* and STRONG for *strong emphasis*. These are *logical* descriptions of the enclosed text. The recommendation in the HTML specification is that text marked with EM be italicized if possible, while text marked with STRONG should be rendered as bold. This is exactly what is done by **Cello**, as shown in Figure 1.2. **Lynx**, on the other hand, renders both EM and

STRONG as underlined text (Figure 1.3). Character-based programs, such as **lynx**, can really do only four things to text: Underline it, bold-face it, force it to capital letters, or display it in reverse video. Given these limitations, **lynx** cannot render as distinct all the different elements in HTML. It therefore renders EM and STRONG in the same way.

Highlighting elements, such as EM and STRONG, can be placed almost anywhere you find regular text. The only exception is the TITLE element in the HEAD. The text inside a TITLE element can be text only; there can be no HTML elements inside a TITLE. Recall that the text inside a TITLE is not part of the document, but simply a text string providing information *about* the document.

HTML has several other logical highlighting elements, such as CODE for computer code, KBD for keyboard input, VAR for a variable, and so on. HTML also has physical highlighting elements, such as B for bold-faced, I for italics, and TT for a typewriter font. It is recommended that you specify logical meaning for text strings rather than specific physical styles, since this assigns more meaning to the associated text and also gives the browser more flexibility in determining the best presentation.

PARAGRAPHS

Look at the next line, beginning with the string `<P> Sorry....`:

```
<P> Sorry.  No such luck.       This document
does
contain examples of HTML markup, for example, here is an "unordered
list":
<UL>
```

The `<P>` tag marks the beginning of a paragraph and can best be thought of as marking the start of a paragraph *container*. Most browsers interpret a `<P>` paragraph mark as a skipped line, as shown in Figures 1.2 and 1.3.

Note, also, that a paragraph mark can be anywhere in a line and need not be the first piece of text on a line. For example, the three lines:

```
the World Wide Web.

<P> Sorry.  No such luck.        This document
```

could be written:

```
the World Wide Web.  <P> Sorry.  No such luck.  This document
```

and both would be rendered the same. Recall that the rendering of an HTML document depends only on where the markup tags are located relative to the text they are marking. Of course, putting <P> at the beginning of a line makes it easier to read the *raw* HTML and is strongly recommended.

You will notice that there is no ending tag </P> in Figure 1.1 to mark the end of the paragraphs. In HTML, the ending paragraph tag is optional. The rule is that a paragraph is *ended* by the next <P> tag starting another paragraph, or by any other tag that starts another *block* of text, such as a heading tag (<Hn>), a quotation tag (<BLOCKQUOTE>) , list tags, and so on. Thus, the paragraph ending with the words "`unordered list`" is actually ended by the following <UL> tag that marks the beginning of an *unordered list.*

You may have noticed that the first paragraph in Figure 1.1 had no paragraph tags. This is because an initial break is caused by the first heading element. A correct HTML document should have a <P> tag at the beginning of this first paragraph. This does not change the spacing between this paragraph and the previous heading.

The HTML definition states that if two adjacent elements describing the logical structure of the document require some special vertical spacing, then only one of them (the largest spacing value) should be used and the other should be ignored. This implies that silly and invalid constructs, such as:

```
<p><p><p><p>  This is a paragraph
```

should yield, at most, a single paragraph break and nothing more. You should avoid doing such things, since, by definition, paragraphs should not be empty.

UNORDERED LISTS

Having beaten paragraphs into the ground, we now move on to the next part of this example, namely the unordered list. HTML supports various types of lists; the example given here is an unordered list. An unordered list begins with the tag <UL>, such as:

```
<UL>
   <LI> One item of the list,
   <LI> A second list item  <LI> A third list item that goes on and on and
   on to indicate that the lists can wrap right around the page and still
   be nicely formatted by the browsers.
   <LI> The final item.
</UL>
<p> Lists are exciting. You can also have ordered lists (the items are numbered)
```

Lists are definitely not empty, and every list must be terminated with an end tag, here </UL>, to define the end of a list. Other similar list elements are: the *ordered* list element **OL**, the *menu* list element **MENU**, and the *directory* list element **DIR**.

A **UL** , **MENU**, **OL**, or **DIR** list can contain only **LI** list elements. A *list element* has the start tag <LI> and cannot be empty, as every list item must consist of some text. Nevertheless, an ending </LI> tag is not required, as the ending of a list element is implied by the next <LI> or by the </UL> tag that finally terminates the list.

In general, unordered lists are just that—unordered lists of items, with each element marked by an indentation of some type and a star or bullet. It is up to the browser to format them nicely, and, as you can see, both **lynx** and **Cello** do very similar things. However, you will also note that **lynx** and **Cello** do different things to the spacing that surrounds the list; such browser to browser variations are common.

As mentioned, the only thing that can go inside a UL element is an LI element. Thus, you cannot write such things as:

```
<UL>
    here is some non-list text inside a list
    <LI>  Here is list item 1.
</UL>
```

A list item (LI) element, however, can contain lots of things. For example, an LI element can contain text, the IMG element (for inline images), text emphasis (such as the STRONG element), another list, and even a fill-in HTML form. However, it cannot contain a heading element. Heading elements can be only directly inside the BODY, or inside a FORM (for fill-in forms) or a BLOCKQUOTE (for quoted text) element.

HORIZONTAL RULE ELEMENT

The final element in Figure 1.1 is the HR, or horizontal rule, element. This element simply draws a horizontal dividing line across the page, which is useful for dividing sections. This is also an empty element, since it does not act on a body of text.

LESSONS FROM EXAMPLE 1

1. Titles should be short and descriptive of the document content.
2. HTML is a hierarchical set of markup instructions. The outer layer of this organization, showing the basic document outline, is:

```
<HTML>
        <HEAD>
        .. document head ..
        </HEAD>
        <BODY>
        .. document body ..
        </BODY>
<HTML>
```

The TITLE goes inside the HEAD, while the text to be displayed goes inside the BODY.

3. White spaces, tabs, and blank lines are irrelevant in the formatting of a document; the only thing that matters is the placement of the HTML markup *tags*.
4. Heading elements (H1 through H6) can go only inside the BODY, FORM, or BLOCKQUOTE elements.
5. UL, OL, MENU, and DIR lists can contain only LI list item elements. The LI elements can contain text, images, and other lists, but cannot contain headings.

EXAMPLE 2: IMAGES AND HYPERTEXT LINKS

Figure 1.1 served to illustrate some of the basic features of HTML. This second example consists of two distinct documents, ***ex2a.html*** and ***ex2b.html***, with a hypertext link from one to the other. The documents are shown in Figures 1.4 and 1.5. *Ex2a.html* also includes inline images to illustrate a few things to think about when using images in your documents. Notice how space characters are used to make the HTML documents easier to follow.

```
<HTML>
<HEAD>
<TITLE> Example 2A, Showing IMG and Hypertext Links </TITLE>
</HEAD>
<BODY>
```

```
<H1> Example 2A: Image Inclusion and Hypertext Links </H1>

<P> Greetings from the exciting world of HTML Example documents. OK, so text
    is not so exciting.  But how about some pictures!

<P> There are many ways to fit in the image.  For example,
    you could fit it in this way:
    <IMG SRC="home.gif" ALIGN="top">, this way
    <IMG SRC="home.gif" ALIGN="middle"> or this way
    <IMG SRC="home.gif" ALIGN="bottom">.

<P> Another important thing: you can make
    <a href="ex2b.html"> hypertext links </a> to other files.
    You can even make hypertext links using images,  for example
    <a href="ex2b.html"> <IMG SRC="sright.xbm" ALIGN="middle"> </a>.

<P> Lastly, here is a row of images:
    <IMG src="home.gif" alt="[Home Icon]">
    <IMG src="home.gif" alt="[Home Icon]">
    <IMG src="home.gif" alt="[Home Icon]">
    <IMG src="home.gif" alt="[Home Icon]">
    <IMG src="home.gif" alt="[Home Icon]">
    <IMG src="home.gif" alt="[Home Icon]">
    <IMG src="home.gif" alt="[Home Icon]">
    <IMG src="home.gif" alt="[Home Icon]">

</body>
</html>
```

Figure 1.4 Contents of the example HTML document ***ex2a.html***. The rendering of this document by the **Mosaic for X-Windows**, **MacWeb**, and **lynx** browsers is shown in Figures 1.6 through 1.8.

```
<HTML>
<HEAD>
<TITLE> Example 2B: Target of example Hypertext Link</TITLE>
</HEAD>
<BODY>

<h2> Target of Hypertext Link </h2>

<p> OK, so now that you are here, how do you get back?  This document
    doesn't have any hypertext links, so you have to use a "back" button (or the
    'u' key if using lynx) to move back to the previously viewed document.

</BODY>
</HTML>
```

Figure 1.5 Contents of the example HTML document ***ex2b.html***. This document is the target of a hypertext link from the file ***ex2a.html*** shown in Figure 1.4.

THE EXAMPLE DOCUMENT

Figure 1.4 and the contents of the file *ex2a.html* will be examined first. The first paragraph is a simple text paragraph. The second paragraph is similar, except that it contains three images included using the IMG element: `<IMG SRC="home.gif">`. *Home.gif* is a GIF format image file; I know this by the *.gif* filename extension (and, of course, because I created it). GIF files are one of the three image formats that can be included within HTML documents.

EXAMPLE DOCUMENT RENDERED

The rendering of the document *ex2a.html* (listed in Figure 1.4) is shown in Figures 1.6 through 1.8. Figure 1.6 shows the document as presented by the **Mosaic for X-Windows** browser (Version 2.4). Figure 1.7 shows the same document viewed using the Macintosh **MacWeb** browser, and Figure 1.8 shows the document as seen by the **lynx** browser. Many of the differences amongst them are simply due to the different window sizes and fonts. Still, there are differences that warrant mention.

Note the appearance of the images in Figures 1.6 and 1.7. Images are included as if they were large letters or words appearing in line with the surrounding text and deforming the line spacing to ensure that no text overlaps the image. By default, the included image is inserted to align the bottom of the image with the bottom of the text. Note that there is no special wrapping or flowing of the text around the image and, since you have no way of guaranteeing how the document will be presented, no guarantee that an image embedded in the middle of a sentence will appear in a particular place on the screen. The only way you can guarantee image placement is to put it as the first item following a paragraph (or other) break; then, it will always be the first item on a line.

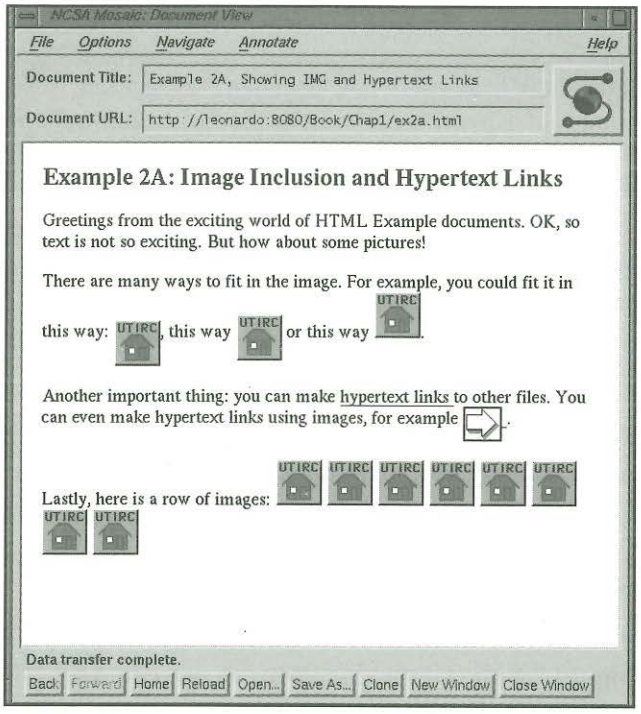


Figure 1.6 Mosaic For X-Windows rendering of the HTML document ***ex2a.html*** (the HTML document source is shown in Figure 1.4).

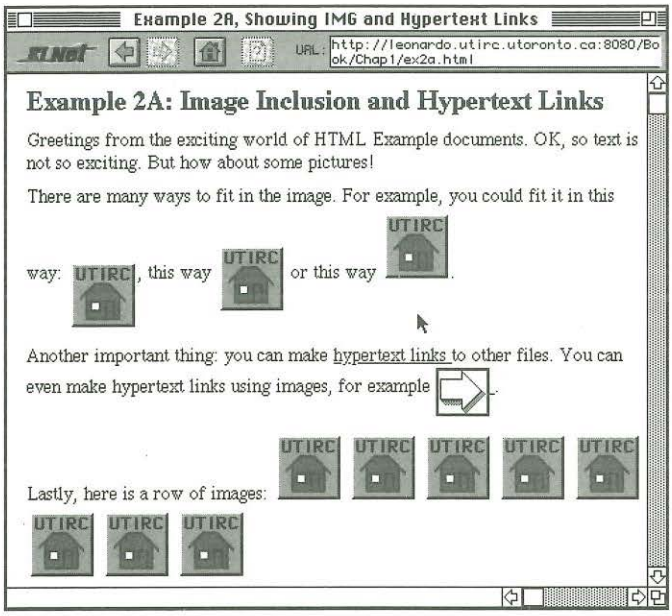


Figure 1.7 **MacWeb** browser rendering of the HTML document ***ex2a.html*** (the HTML document source is listed in Figure 1.4).

Alignment Options

There are three alignment options for the image. These are specified by the **ALIGN** attribute, which can take on the three possible values: `ALIGN=top`, `ALIGN=bottom`, and `ALIGN=middle` (the argument is case-insensitive), with `align=bottom` being the default. The three options are illustrated with the three Home Icons (I am afraid I am not a great graphic artist). The first, `ALIGN=top`, aligns the top of the image with the text, while the `ALIGN=middle` value aligns the middle of the image with the bottom of the text and `ALIGN=bottom` aligns the bottom of the image with the bottom of the text.

Lynx Interpretation

Figure 1.8 shows the **lynx** interpretation: **lynx** is a text-only browser and cannot display images. **Lynx** simply replaces each occurrence of an image

with the text string `[IMAGE]`. This is not very descriptive, but it tells you at least what you are missing. If the string `[IMAGE]` is insufficient, you can replace it with something more useful by using the `ALT="string"` attribute to the **IMG** element. You use the **ALT** attribute to specify a text alternative for an image, which is particularly useful when a browser is unable to display images. The usage is simply:

```
<IMG SRC="image.file" ALT="[A text alternative to the Image]">
```

The string specified for **ALT** is case-sensitive, and it is common (but not necessary) to surround the string with square brackets. An example of this use is shown at the bottom of ***ex2a.html***, where the attribute `alt="[Home Icon]"` is used with all the inline images. The resulting text rendering is shown in Figure 1.8.

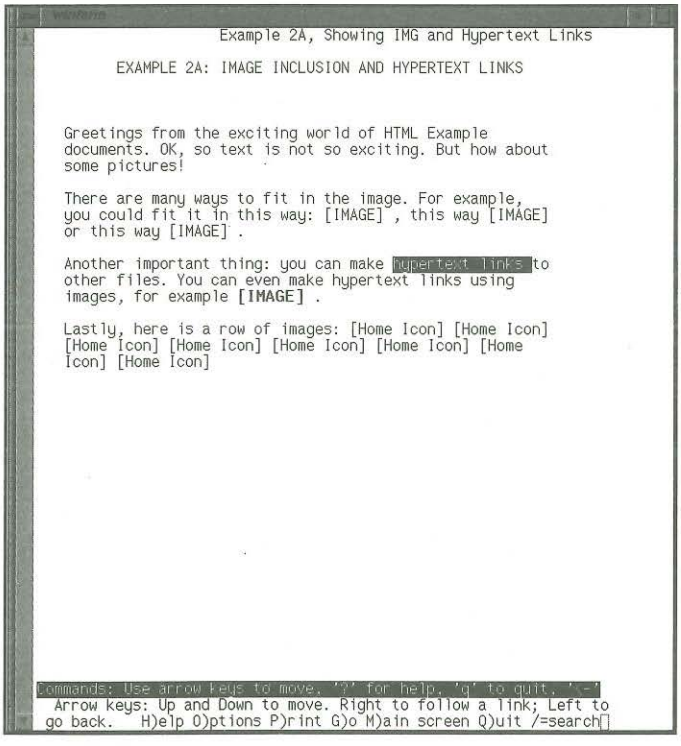


Figure 1.8 **Lynx** browser rendering of the HTML document ***ex2a.html*** (the HTML document source is listed in Figure 1.4).

How does the browser actually obtain the images and complete the document? The browser first obtains the HTML document and then looks for IMG elements. If it finds IMG elements, the browser makes additional connections to the server to obtain the required images. Thus, a single document containing ten images requires 11 distinct connections to complete the document. Needless to say, this can be slow, particularly if you have a slow network connection. Most graphical browsers, such as **Mosaic** or **MacWeb**, have a Delay Image Loading button or pull-down menu selection that disables the automatic loading of inline images. This can save time when viewing a document, but is useful only if the author has written the document to be understandable without the pictures.

ANCHORS

The third paragraph in ***ex2a.html*** shows a hypertext link. The form is straightforward:

```
<A HREF="ex2a.html"> hypertext links </A>
```

The element marking a hypertext link is called an **A**, or *anchor*, element, and the marked text is referred to as a *hypertext anchor*. The area between the beginning `<A>` and ending `</A>` tags becomes a *hot* part of the text. With **Mosaic** or **MacWeb**, this section of text is displayed with an underline and usually in a different color (often blue), while with **lynx**, this region of text is displayed in bold characters. Placing the mouse over this region and clicking the mouse button, or, with **lynx**, using the tab key to move the reverse-video region to lie over the hot part and pressing return, causes the client to access the indicated document or other Internet resource.

You can also use images as hypertext anchors. At the end of the second paragraph in ***ex2a.html*** (Figure 1.4), I have marked the image ***sright.xbm*** as the anchor (the ***.xbm*** means this is an X-Bitmap image). The relevant piece of HTML is:

```
<a href="ex2a.html"> <IMG SRC="sright.xbm" ALIGN="middle"> </a>
```

The graphical browsers indicate this by *boxing* the image with a colored or highlighted box, while **lynx** simply bolds the `[Image]` text string it puts in place of the image. With this mechanism, you can use small images as *button* icons, as is commonly done in multimedia applications. This doesn't do much good with **lynx**, of course, so, if you do employ *navigation icons* such as these, you had better add an **ALT** attribute to let your **lynx** guests know what is going on.

This second image is in the X-Bitmap format, as indicated by the extension ***.xbm***. Only three image formats can be included inline within HTML documents. These are:

.gif	GIF image files (color, grayscale, or black and white)
.xbm	X-Bitmaps (black and white)
.xpm	X-Pixelmaps (color)

Thus, if you want to display images within your document, you must ensure they are in one of these three format options.

PARTIAL URLs

The *target* of the hypertext link is indicated by the anchor attribute **HREF**, which takes as its value the URL of the target document or resource. As mentioned in the Introduction, a URL is a text string that indicates the server protocol (HTTP, FTP, WAIS, etc.) to use in accessing the resource, the Internet Domain Name of the server, and the location and name of the resource on that particular server. Obviously, the **HREF** attributes in Figure 1.4 do not contain all this information! These URLs are examples of *partial* URLs, which are a shorthand way of referring to files or other resources *relative* to the URL of the document being currently viewed. For

Figure 1.4, this means: Use the same protocol, Internet Domain Name, and directory path of the present document (*ex2a.html*), and retrieve the indicated file *ex2b.html* from the same directory.

If you click the mouse button over the hypertext anchor, the browser downloads and displays the linked document, as shown in Figure 1.9. To return to the previously viewed document, press the Back button on the browser control panel (with **lynx**, you press the letter "u"), which takes you back to the previously displayed document, namely *ex2a.html*. Figure 1.10 shows what this document looks like the second time around, using **Mosaic for X-Windows**. The document is now subtly different: The portion of text that served as the launching point for the hypertext link, previously underlined by a solid line, is now underlined by a dashed line. On a color display, you also see the anchored text rendered with a changed text color. Programs such as **Mosaic** and **MacWeb** use this change in highlighting to let you know that you have already visited this link. This helps to keep you oriented by letting you know where you have already been. Unfortunately, this is not possible with **lynx**, as there are an insufficient number of highlighting modes to allow for this level of subtlety. With **lynx**, you have to pay a bit more attention to what you have been doing.

As mentioned, the anchor

```
<A HREF="ex2b.html"> hypertext links </A>
```

uses a partial URL, which refers to locations relative to the URL of the document itself. This partial URL idea is great news, because it means that you need not specify entire URLs for simple relative links between files on the same computer. Instead, you need only specify their position on the file system relative to each other, as was done in Figure 1.4.

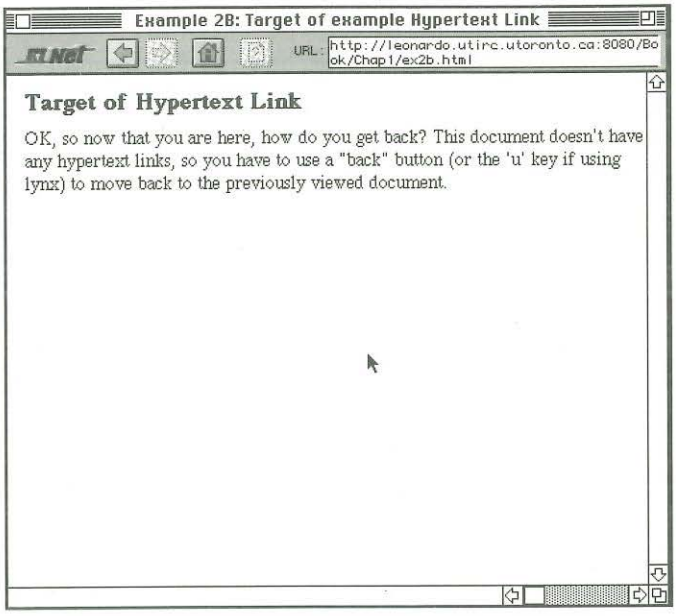


Figure 1.9 MacWeb browser rendering of the HTML document ***ex2b.html*** (the HTML document source is listed in Figure 1.5).

Partial URLs can also point to directories other than the one containing the current document. Specification of these relative directories is done using a UNIX-like path structure. Suppose that our example documents *ex2a.html* and *ex2b.html* lay in the directory structure shown in Figure 1.11. That is, our files *ex2a.html* and *ex2b.html* are in the directory *Project1/Examples*, while the file *e2c.html* is in *Project1/Examples/SubDir* and *ex2d.html* is located in *Project1/Other*.

How do you reference the files *ex2c.html* and *ex2d.html* from the file *ex2a.html*? To reference *ex2c.html*, you simply create a hypertext link that accesses the partial URL *SubDir/ex2c.html*:

```
<A HREF="SubDir/ex2c.html"> hypertext links </A>
```

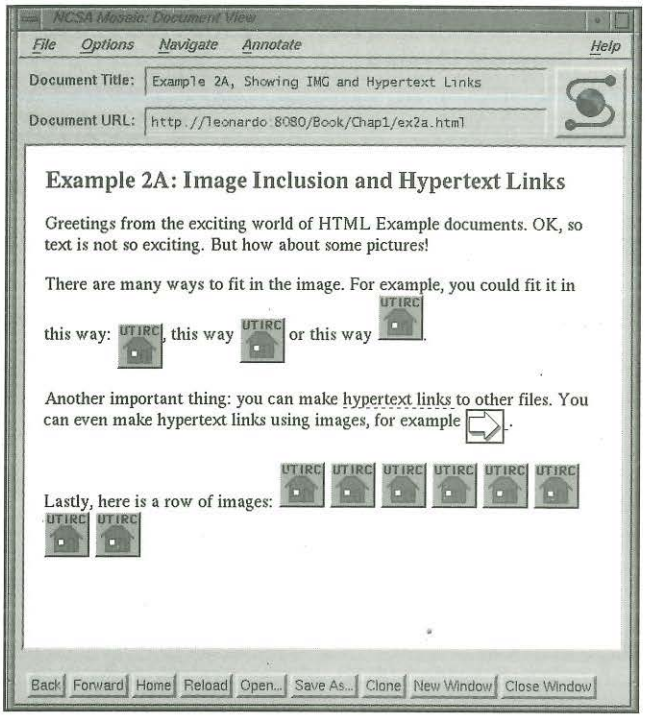


Figure 1.10 Mosaic for X-Windows browser rendering of the HTML document ***ex2a.html***, after returning from a hypertext jump to the document ***ex2b.html***. Note how the line under the anchor is now dashed instead of solid.

Notice the UNIX-like directory pathnames in which the forward slash character indicates a new directory. The specification of the URL syntax uses the forward slash to denote directories or any other hierarchical relationship (formally, URLs can reference not just files, but also programs and other resources). Note that you *cannot* use backslashes as you do with DOS and Windows, or colons as you do on Macintoshes.

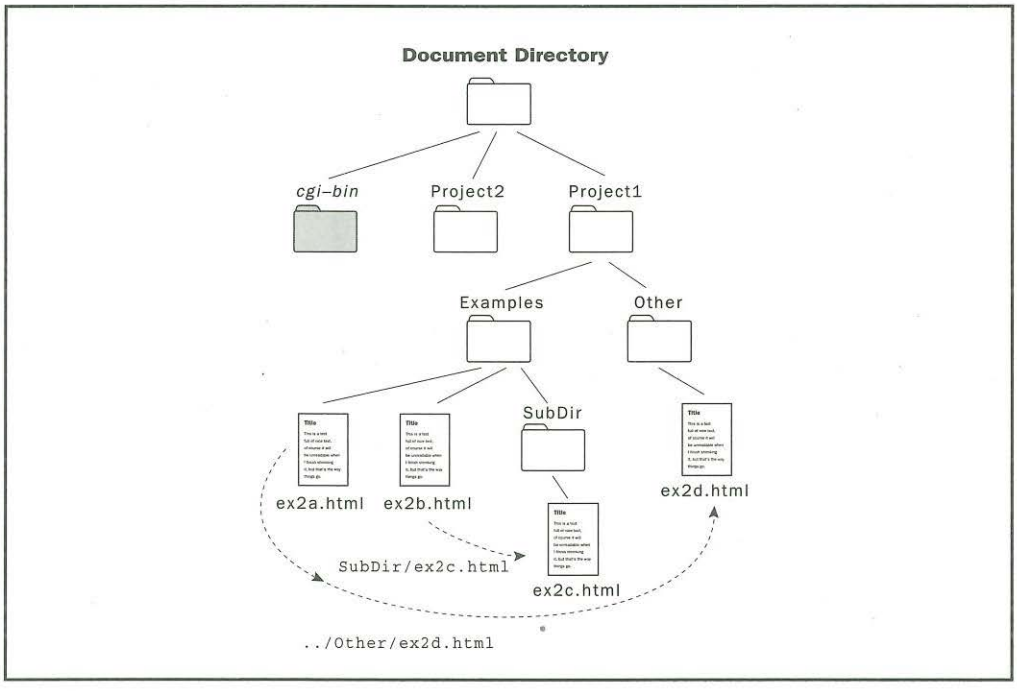


Figure 1.11 Accessing neighboring files using partial URLs. The Folders and files show the organization of files lying in the *Document Directory*—the directory that contains resources available to clients accessing an HTTP server. The dotted lines and the corresponding text strings show partial URLs relating the file ***ex2a.html*** to the files ***ex2c.html*** and ***ex2d.html***. The grayed directory, ***cgi-bin***, is a special directory used to store programs that can be executed by the HTTP server. This directory is kept physically distinct from the true document directories, since programs present a security risk to the server and must be guarded more carefully than documents (see Chapter 4 for more details).

CREATING LINKS

If you want to create a link from *ex2a.html* to *ex2d.html* in *Project/Other,* write the URL as:

```
<A HREF="../Other/ex2d.html"> hypertext links </A>
```

since the file is one directory level up (the symbol ".." meaning one directory up) and one level down into the directory *Other*. Also, a single dot (.)

implies the current directory, so `HREF="./Subdir/ex2c.html"` and `HREF="Subdir/ex2c.html"` are equivalent.

Of course, this is going to be a problem if you actually have the slash character in a filename, since the URL convention will try to interpret it as a directory change. You therefore should avoid directory or filenames containing this character if at all possible. The URL specification does have a way of allowing this and other *special* characters to be in a URL without having them interpreted specially. This *encoding* mechanism is discussed in Chapter 2.

There are a lot of other partial URL forms, and of course, the format for full URLs has yet to be discussed. More examples of URLs appear later in this chapter, while a detailed description of the URL syntax is given in Chapter 2.

Finally, we refer to the last line of ***ex2a.html*** (Figure 1.4), which is simply a row of inline images. Note that these wrap to best fit the width of the screen, just as if they were a sequence of words (Figures 1.5 and 1.6). Consequently, if the user changes the size of the browser window, these icons will be repositioned. This once again points out the wide variation possible between different renderings of the same document.

HYPERTEXT LINKS: THE GOOD, THE BAD, AND THE UGLY

As you can see, hypertext links are easy and can be very useful. However, you should not get carried away, as they can quickly become a source of irritation. For example, it is usually best that hypertext links flow naturally out of the text. Thus, it is better to write:

```
<p> The issue of hormone-controlled ostrich-feather growth has recently been a
topic of intense <A HREF="ostrich-paper.html"> interest </A>
```

than:

```
<p> For information about current hormone-controlled ostrich-feather growth press
<A HREF="ostrich-paper.html"> here </A>
```

```
<HTML>
<HEAD>
<TITLE> Examples of Bad Hypertext Links </TITLE>
</HEAD>
<BODY>
<H1> Examples of Good/Bad Hypertext Link Design</H1>

<p> <B> 1) Don't distort the Written Text  </B>
<P> <B> Good: </B>
<BR>The issue of hormone-controlled ostrich-feather growth has recently
    been a topic of intense <A HREF="ostrich-paper.html"> interest</A>.

<P> <B> Not So Good: </B>
<BR> For information about current hormone-controlled ostrich-feather
     growth press  <A HREF="ostrich-paper.html"> here </A>.
<p> <B> 2) Keep the linked text section short. </B>
<p> <B> Good: </B>
<br> The life cycle of the  <A HREF="animal.html"> atlantic polar-bear
     ocelot </A> is a complex and .......

<p> <B> Not So Good: </B>
<br> <A HREF="animal.html"> The life cycle of the atlantic polar-bear
     ocelot is a complex </A> and .......

<p> <B> 3) Link Icon and Text Together </B>
<p> <B> Good: </B>
<br> <A HREF="file.html"> <IMG SRC="home.gif"> The latest </A> home
     security systems breakdown show......
<p> <B> Not so Good: </B>
<br> <A HREF="file.html"> <IMG SRC="home.gif"></A> The latest home
     security systems breakdown show......

</BODY>
</HTML>
```

Figure 1.12 Listing for the document ***badlinks.html***. This document illustrates good and bad examples of hypertext links. A **Mosaic for X-Windows** rendering is shown in Figure 1.13.

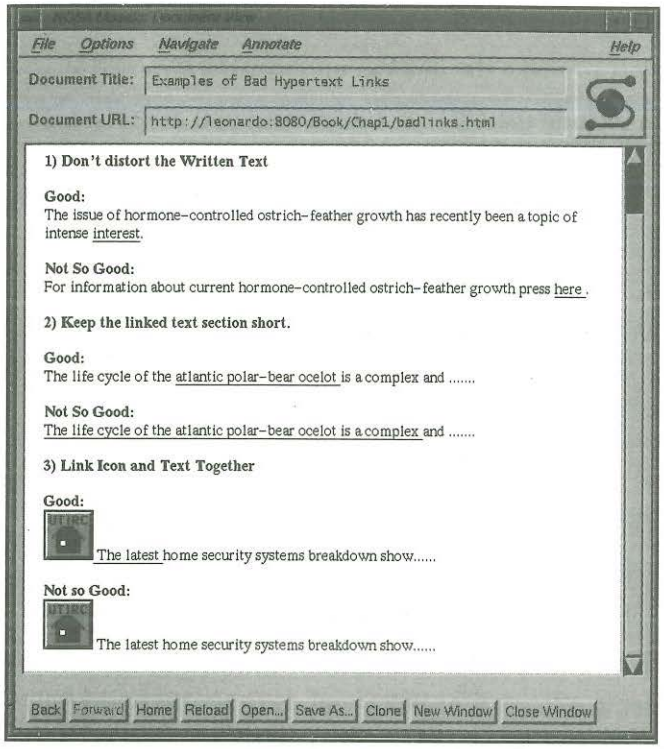


Figure 1.13 Mosaic for X-Windows rendering of the file ***badlinks.html*** (shown in Figure 1.12).

You also don't want the link to be gratuitously long. It is much more effective to link to a single word or selection of words than to a whole sentence. Thus such links as:

```
<p> <A HREF="animal.html"> The life cycle of the atlantic polar-bear ocelot </A>
is a complex and interesting example of .....
```

are probably better when written:

```
<p> The life cycle of the  <A HREF="animal.html"> atlantic polar-bear ocelot </A>
is a complex and interesting example of .....
```

These (and other) examples of bad hypertext links are included in the HTML document shown in Figure 1.12 and displayed in Figure 1.13. There is no hard and fast rule for creating bad and good link presentations.

In general, if you are adding links within a text document that is intended to be easy to read, it is best to make the links as unobtrusive as possible.

Sometimes, it is useful to include the URL of a document you are referencing as part of the text—you don't always have to hide the URL in the HTML markup. This is particularly useful if your document is likely to be printed by a browser, since the HTML markup tags are lost on paper. For example, the WWW FAQ document, a wonderful repository of useful information about the World Wide Web, contains such lines as:

```
<p> ....Read the Wusage home page, which is found at
<A HREF="http://siva.cshl.org/wusage.html">
    http://siva.cshl.org/wusage.html
</A>
```

If you are reading the HTML document containing this markup, just click on the anchored text to view the referenced document. If you are reading the printed version, you still have all the information you need to access the site when you next have access to the Internet.

If you are putting together a collection of links, you want to think carefully about their organization. Should they be in a paragraph? Probably not. More likely, they should go in a list or as a menu bar. You will also want to organize them in some logical manner, particularly if you have a lot of hypertext links to present—a page full of random links is exceedingly frustrating to use. Link-mania pages like this can occur quite innocently, often when you have been slowly assembling lists of interesting URLs. This may be fine for you, but can be confusing to someone else.

Figure 1.15 shows a *home* page constructed using some of the above thoughts on "bad" anchor design (the source for this document is in Figure 1.14), while Figure 1.17 shows an improved and better organized home page using the suggestions given in the preceding paragraph (the source is in Figure 1.16). We will talk more about home pages in the next example, but, by simple comparison, you will see that Figure 1.17 is much easier to understand than Figure 1.15, even though Figure 1.17 contains less textual information.

Finally, and most important of all: Make sure all your hypertext links work and go to the right place! There is nothing worse than clicking on a hypertext link only to get "ERROR. Requested document not available" in response. Actually, that's not quite true. It is even worse to click on an anchor that indicates a particular destination and find that you have actually accessed something completely different and obviously incorrect. These types of errors tell the user that the document developer did not bother with even the simplest checks of his or her work and immediately brings into question the contents of the documents themselves. It is easy to check your links: Just do it.

LESSONS FROM EXAMPLE 2

1. Images are included via the IMG element, such as:

```
<IMG src="prism-small.gif" ALT="[Text stuff]">
```

   SRC specifies the URL of the image file to be included. This image file can be in one of three formats (the format is specified by the filename extension):

.gif	GIF format image file
.xbm	X-Bitmap (black and white)
.xpm	X-Pixelmap (color)

   The **ALT** attribute gives a text string to be displayed by browsers that cannot display images.

   Images are treated as if they were big text letters; the surrounding text does not flow around the image.

2. Hypertext links to another document are included using the A (anchor) element, such as:

```
<a href="SubDir/example1.2B.html"> hypertext links </a>
```

   where HREF is used to specify the URL of the target of the link. The examples here are of partial URLs: Partial URLs assume the same

Internet site and protocol as for the document currently being viewed, and look for the file (or resource) relative to it. In this regard, the slash (/) and double dot (..) characters are special, representing relative positions in the directory (or other) hierarchy.

Images can also be hypertext anchors via constructs, such as:

```
<a href="someplace.html"> <img src="image.gif"> </a>
```

3. Don't use hypertext links gratuitously. If they are embedded in the text, try to make them flow with the text. If a paragraph has many links, try thinking of another way to present the material: Maybe it should be a list or a menu, or perhaps the hypertext anchors could be combined into a less intrusive form. Above all, make sure the links work; links going nowhere or to the wrong place are cardinal sins of HTML authoring.

EXAMPLE 3: HOME PAGES

This time, we will explore an example that is a bit more meaningful. This is an example of a *home* page and is based on a page constructed for the Instructional and Research Computing Group at the University of Toronto. A home page is designed to be the first page seen by explorers from the outside world and is your introduction to new Web visitors to your site. It is used to direct people to other interesting resources that you have and perhaps to other resources in your Internet neighborhood. This example comes from an academic institution, but the organizational model will be largely the same for business, entertainment, or other applications, although the details will certainly be different. Chapter 9 gives some examples from these other environments.

There are several issues to think about when designing a home page. A home page should be:

1. Small—A home page should be a small HTML document with a minimum of extraneous graphics or text. Large images and the details of the resources at your site should be elsewhere, with hypertext links from your home page to these locations.
2. Concise—A home page is like an introductory map of your site, explaining what the site is and where to find the local resources. Thus, it should briefly outline the content of your server and provide hypertext pointers to those resources. It should also briefly outline the organization of your site, such as the meanings of icons or menus, so that visitors will know how to navigate their way around.
3. Not Dependent on Graphics—You can include graphics in this page, but they should be small. Many people have slow Internet connections and don't want to spend many minutes waiting for thousands of kilobytes of gratuitous imagery. Icons or images on a home page should be small image files, totaling less than ten kilobytes if possible. Also, you must ensure that a user can navigate from your home page without a graphical browser, since many users will be accessing your site with text-only browsers. Thus, if you use icons for navigation buttons you should also provide a text-only option.

In addition your home page should provide:

4. Contact Information—Someone will want to contact you at some point, perhaps to point out a problem or even compliment you on your work. The home page should include contact information for the administrator of the resources. A common, generic e-mail address for the site administrator is *webmaster@www.domain.name*. This information should be included somewhere on the home page, usually at the bottom.

```
<HTML> <HEAD>
<TITLE> Instructional and Research Computing </TITLE>
</HEAD> <BODY>
<hr>

<h1> Instructional and Research Computing </H1>

This is the home page of the Instructional and Research Computing
Group <STRONG>(IRC)</STRONG>, one of seven departments of the Division
of Computing and Communications.
The IRC group provides support for
<A HREF="MulVis/intro.html"> multimedia and visualization techniques</A>,
access to and support for
<A HREF="HPC/intro.html"> high performance computing</A>,
and support for <A HREF="AdTech/intro.html"> adaptive technology
</A>.  (aids for the physically challenged)  We also have some interesting
links to <A HREF="Lists/Lists.html"> WWW Starting Points </A>,
a big list of <A HREF="Lists/Lists.html"> WWW Search Tools</A>,
another list of hypertext pointers to
<A HREF="Lists/Libraries.html"> Libraries </A> resources, and a
link to the  <A HREF="http://www.utoronto.ca/uoft.html"> Main University
Home Page </A>.

<p>
If you become lost in our documents use the navigation icons.
The <EM> home </EM> icon brings you back here, while the <EM> up </EM>
icon takes up up one level in the document hierarchy.  <EM> Info </EM>
and <EM> help </EM> are also useful, while the <EM> letter</EM> icon let
you send us a message, and the <EM> search </EM> icon allows you to do
a textual search of our pages.
<hr>
  <A HREF="home.html"><IMG SRC="home.gif"     ALIGN=TOP ALT="[Home]"  ></A>
  <A HREF="help.html"><IMG SRC="ic_help.gif" ALIGN=TOP ALT="[Help]"  ></A>
  <A HREF="info.html"><IMG SRC="ic_info.gif" ALIGN=TOP ALT="[Info]"  ></A>
  <A HREF="/cgi-bin/mail.pl"> <IMG SRC="ic_mail.gif" ALIGN=TOP ALT="[Mail]"  ></A>
  <A HREF="home.html"><IMG SRC="ic_up.gif"    ALIGN=TOP ALT="[Up]"     ></A>..
  <A HREF="cgi-bin/doc-search.pl"> <IMG SRC="ic_find.gif" ALIGN=TOP ALT="[Search]"></A>
<hr>
<ADDRESS>
<A HREF="Staff/web_admin.html"> webmaster@site.address.edu </A>
</ADDRESS>

</BODY>
</HTML>
```

Figure 1.14 HTML document listing for the file ***home_bad.html***, which contains a poorly designed home page. Figure 1.15 shows the rendering of this document on a PC browser (**Air Mosaic**).

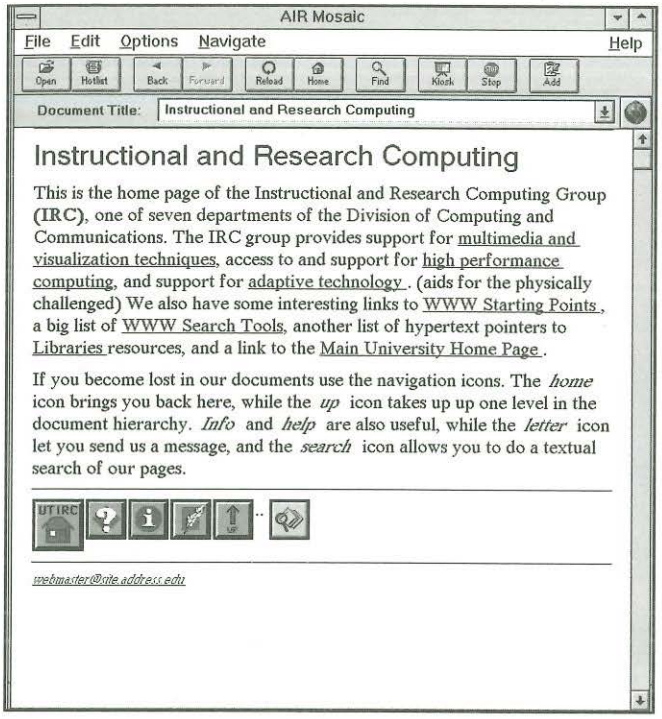


Figure 1.15 Air Mosaic rendering of the HTML document ***home_bad.html.***

Figure 1.16 shows a typical HTML home page document, while Figures 1.17 and 1.18 show how this page looks using the Windows **Air Mosaic** and UNIX **lynx** browsers. Let's first look at the document structure shown in Figure 1.16 and review how the home page was constructed.

```
<HTML> <HEAD>
<TITLE> Instructional and Research Computing </TITLE>
</HEAD> <BODY>
<hr>

<h1> Instructional and Research Computing </H1>
```

```
<HTML> <HEAD>
<TITLE> Instructional and Research Computing </TITLE>
</HEAD> <BODY>
[<A HREF="home.html">              Home      </A>]
[<A HREF="help.html">              Help      </A>]
[<A HREF="info.html">              Info      </A>]
[<A HREF="/cgi-bin/mail.pl">       Mail      </A>]
[<A HREF="home.html">              Up        </A>]
[<A HREF="cgi-bin/doc-search.pl"> Search    </A>]
<hr>

<h1> Instructional and Research Computing </H1>

This is the home page of the Instructional and Research Computing
Group <STRONG>(IRC)</STRONG>, one of seven departments of the Division
of Computing and Communications. We provide:

<UL>
<LI> support for <A HREF="MulVis/intro.html"> multimedia and visualization
     techniques</A>
<LI> access to and support for <A HREF="HPC/intro.html"> high performance
     computing</A>
<LI> support for <A HREF="AdTech/intro.html"> adaptive technology </A>
     (aids for the physically challenged).
</UL>
Some other useful University resources are:
<p>
  <A HREF="Lists/Lists.html"> WWW Starting Points </A> |
  <A HREF="Lists/Lists.html"> WWW Search Tools</A> |
  <A HREF="Lists/Libraries.html"> Libraries </A> |
  <A HREF="http://www.utoronto.ca/uoft.html"> Main University Home Page </A>
<p>
If you become lost in our documents use the navigation icons.
The <EM> home </EM> icon brings you back here, while the <EM> up </EM>
icon takes up up one level in the document hierarchy.  <EM> Info </EM>
and <EM> help </EM> are also useful, while the <EM> letter</EM> icon let
you send us a message, and the <EM> search </EM> icon allows you to do
a textual search of our pages.
<hr>
  <A HREF="home.html"><IMG SRC="home.gif"       ALIGN=TOP ALT="[Home]"  ></A>
  <A HREF="help.html"><IMG SRC="ic_help.gif"    ALIGN=TOP ALT="[Help]"  ></A>
  <A HREF="info.html"><IMG SRC="ic_info.gif"    ALIGN=TOP ALT="[Info]"  ></A>
  <A HREF="/cgi-bin/mail.pl">
     <IMG SRC="ic_mail.gif"  ALIGN=TOP ALT="[Mail]"  ></A>
  <A HREF="home.html">
     <IMG SRC="ic_up.gif"    ALIGN=TOP ALT="[Up]"    ></A>..
  <A HREF="cgi-bin/doc-search.pl">
     <IMG SRC="ic_find.gif"  ALIGN=TOP ALT="[Search]">
  </A>
<hr>
<ADDRESS>
<A HREF="Staff/web_admin.html"> webmaster@site.address.edu </A>
</ADDRESS>

</BODY>
</HTML>
```

Figure 1.16 Listing of the example home page document ***home.html***.

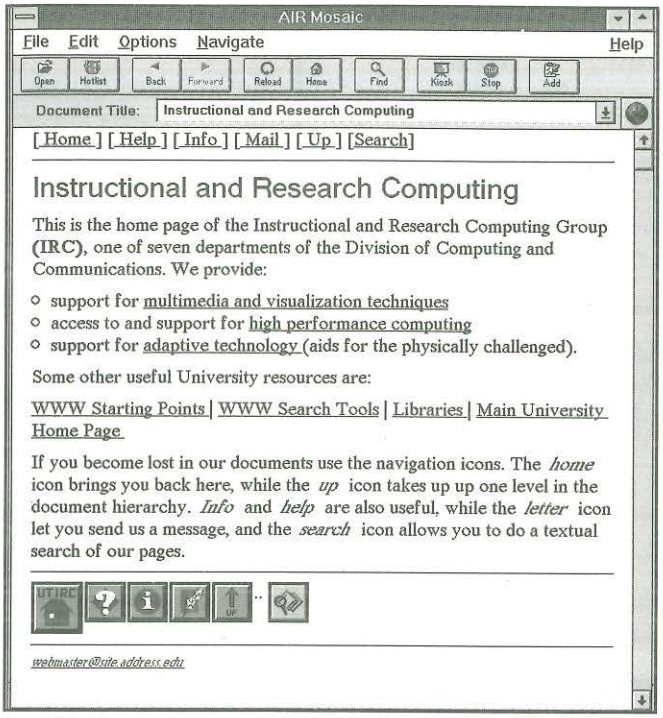


Figure 1.17 Air Mosaic rendering of the HTML document ***home.html*** (the HTML document is shown in Figure 1.16).

TITLE AND HEADING

Figure 1.16 begins with a **TITLE** and a **H1** heading. These are both clearly descriptive of the contents and origin of this document. As mentioned earlier, headings can range from **H1** down to **H6**, in decreasing order of importance. Documents should use heading elements that retain this sense of relative importance, as it helps to build organizational structure within and between documents. There are, for example, useful programs that can build a Table of Contents for large collections of HTML documents based on the contents of the heading elements. This, of course, will work only if the heading elements were used correctly.

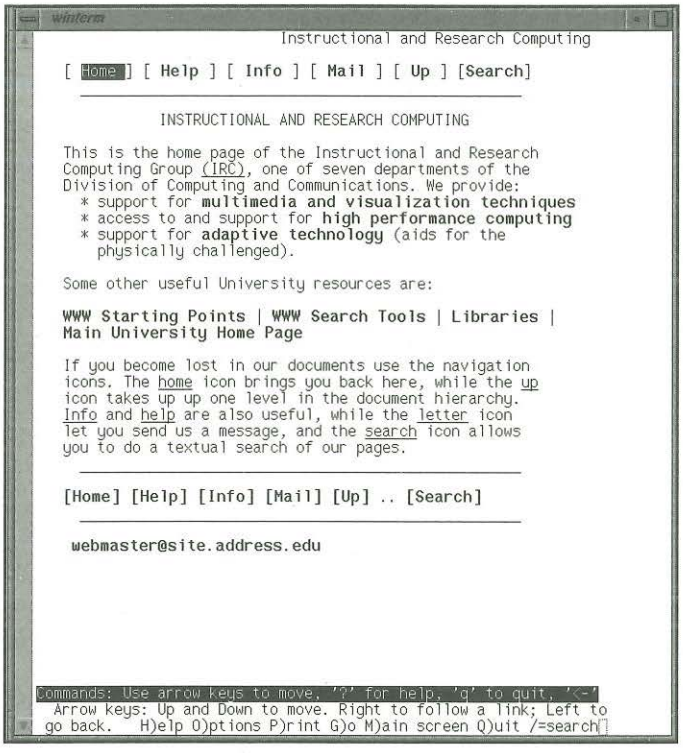


Figure 1.18 Lynx rendering of the HTML document ***home.html*** (the HTML document is shown in Figure 1.16).

Some sites place a GIF image containing the company or organizational logo at the top of the document. This can look very nice and add much to the character of the page. You must be careful, however, that the image not be too big or contain too many colors. Most computer graphics boards are capable of displaying only 256 colors simultaneously and many World Wide Web browsers limit each image within a document to less than 50 colors so that they can display more than one image at the same time (with 50 colors per image, the browser can display up to five images before *running out* of colors). Thus, attempting to display an image containing 256 colors can present problems—in fact, the image can look downright ridiculous. Most graphics editing programs allow you to process images and

reduce the number of colors they contain, often with very little loss of image quality. This is very useful for the reason just mentioned and has the added bonus of making the image files smaller.

TEXT PORTION

The text portion of the home page document (Figure 1.16) should provide clear and concise descriptions of the site and the material it contains, with a minimum of extraneous detail. This should match the type of introduction you prefer: polite and well mannered, or eclectic and off the wall. In the example in Figure 1.16, the first paragraph explains what the site is and provides links to the major areas of interest. Each of these areas may, in turn, have their own home page, specific to the subject at hand. This hierarchical structure lets people quickly find what they are looking for and makes it easy to organize your documents. For example, your HTTP server can have independent subdirectories for each distinct project, with your server's main home page having hypertext links to each project's home page. Other hypertext links will provide alternative relationships between documents, but within this overall hierarchical arrangement.

ORGANIZATION

This home page (Figure 1.16) also contains a second, less detailed list that provides a brief selection of alternative services. These items link to documents or services that are perhaps peripheral to the main purpose of the server, but that may be useful in directing the visitor to his or her destination. In our case, this is a collection of hypertext pointers to other resources that our group commonly uses.

Referring to Figures 1.17 and 1.18, you can see how clean the organization appears. Since the document is written in correct HTML, both browsers display it clearly, subject to their own limitations. Comparing this with the less thought-out version (Figure 1.15) shows the importance of good organizational design: A little thought about how the hypertext links should be

organized makes an enormous difference in the clarity and readability of the presentation.

It is useful to think how this home page model might apply to other applications; for example, to an electronic magazine, or *e-zine*. E-zine home pages often have a magazine title bar (a graphic), followed by the magazine title (an **H1** heading) and a brief introduction to the e-zine. This is usually followed by a collection of pointers to "this month's articles," followed by a hypertext link to a reference document that, in turn, contains a list of links to the home pages of previous e-zine issues. This is very similar to the model just presented. Some examples of different successful home page designs are shown in Chapter 9.

ICONS

Let us return to our example and to the bottom of the HTML document shown in Figure 1.16 and displayed in Figures 1.17 and 1.18. The noticeable feature is a collection of small *icons*, preceded by a short explanation of their meaning. These icons appear on every HTML document at our site and provide universal cues to navigating through the local document collection. For example, the *home* icon always links back to this home page, while the *up* icon links up to the top of whatever set of documents you are looking at. Thus if you had selected to visit the High Performance Computing section, then the *up* icon would bring you back to the High Performance Computing home page. At our site, the *Info* icon refers to a page giving a brief description of IRC and its mandate, while the *Help* icon connects to a page that briefly describes the meanings of all the icons. Finally, the *Mail* icon links to a program on the HTTP server that allows the user to send mail to the server administrator, while the *Search* icon links to a server program that allows the user to do keyword searches on the collection of HTML documents. Note that all these icons are equipped with an `ALT="string"` text alternative. If you are going to navigate with

icons, be sure that people using a text-only browser, such as **lynx**, know what the icons mean!

Having navigation icons is extremely important, particularly when you have a large number of related hypertext documents. It is very easy to get lost when you are browsing through such large collections. Hypertext is not like a *linear* book, where you can always tell where you are by the page number or the thickness of the remaining pages. Navigation icons replace these tactile methods of navigation with symbols that link you to reference points within the collection. In addition, icons can direct the user to general services that may be useful wherever they are in a collection, such as a search tool for searching a database or a mail tool for sending an electronic message to the site administrator.

The very top of this home page shows a text-only variant of the navigation icons (see Figures 1.16 to 1.18). This was added for contrast with the iconic approach. Text or icons both add the same functionality, and choosing between one or the other is largely a matter of taste. Text-based navigation aids can take up less space on a page and do not require that you access the server several times to obtain icon images. The latter problem is mitigated by using the same icons in all your pages. Most clients keep local copies of images once they have been accessed, and don't bother to retrieve them from the server when they are required on subsequent occasions.

Image downloading problems can also be alleviated by using small icons and a reduced number of colors per image. The icons included in the document shown in Figure 1.16 and displayed in Figure 1.17 are only 36 pixels square and contain only 16 colors each (4 bits/pixel). As a result, each icon takes only around 280 bytes so that downloading them to the browser is fast and does not place large demands on the network.

UNIFORM RESOURCE LOCATORS

Looking at Figure 1.16, you will see that most of the URL references are partial references. There is one, however, that is not. This is the URL

pointing to the *main* University Home Page. This is a complete URL that specifies the complete information needed to access the main University HTTP server. This URL is:

```
http://www.utoronto.ca/uoft.html
```

The URL has three main parts:

1. `http:`—The protocol specifier. The string `http:` means that the HTTP protocol is to be used. Other possible protocols are `ftp:` (FTP protocol), `gopher:` (Gopher protocol), `wais:` (WAIS protocol), and `news:` (NNTP protocol). These and other schemes are discussed in Chapter 3.

2. `//www.utoronto.ca`—The Internet domain name of the server. This gives the Internet name that the client should contact. Sometimes, you will see this with a number after the name; for example:

   ```
   www.somewhere.edu:8080
   ```

   This number is a *port number* and specifies the port number that the server is actually talking on. Most HTTP servers "talk" at Port 80. You can leave out the port number if you are contacting the server at the default port.

3. `/uoft.html`—The path and filename to the desired file (or other resource). The part after the last slash specifies the path to a file (or another resource, such as a program), and the file or program name itself. Here, the URL is to the file ***uoft.html*** that lies right at the top of the server's *document directory*: This is the directory under which the HTTP server keeps the documents. This directory can contain subdirectories so that such paths as ***/Project1/Examples/ex2a.html*** are possible (as in Figure 1.11).

   Programs are often placed in a special virtual subdirectory of the document directory, often named ***cgi-bin*** or ***htbin***. Thus, if you see a URL that looks like:

```
http://some.where.edu/cgi-bin/stuff
```

or

```
http://some.where.edu/htbin/stuff
```

then stuff is a program, not a regular file.

The URL scheme allows for all sorts of protocols: `ftp` for the FTP protocol; `gopher` for the Gopher protocol; `wais` for the WAIS protocol, and so on. Thus, using URLs, you can create hypertext links to anonymous FTP servers, Gopher sites, WAIS databases, and many other Internet resources. Clients, such as **Mosaic** and **MacWeb**, are designed to understand these protocols. When they encounter a hypertext reference in an HTML element, such as `HREF="url"`, where `"url"` points to an anonymous FTP or Gopher site, they are able to contact the site, using the appropriate protocol, and interpret the returned results.

The final important point regarding Figure 1.16 concerns the **ADDRESS** element and the *signature* e-mail address located at the bottom of the document. It is always a good idea to sign your HTML documents, particularly the home page or other major pages. The HTML **ADDRESS** element is specifically designed for this purpose. This document places the e-mail address relevant to this page inside the **ADDRESS** element, and also has a hypertext link from the e-mail address to another page containing additional information. This page, here named ***web_admin.html***, often contains additional information about the server and server administrator. For more personal projects, this might be an HTML document containing a brief biography of the document author.

LESSONS FROM EXAMPLE 3

1. A home page should be small and should not contain many large images.
2. Home pages should clearly and concisely describe the contents of a World Wide Web site and should contain hypertext links to these resources.

3. A home page can explain and introduce navigation icons if they are used.
4. Home pages should contain contact information for the administrator of the documents managed at the site.

EXAMPLE 4: COLLECTIONS OF HYPERTEXT DOCUMENTS

This example looks at a hypertext collection of text-based documents. One common use of HTML is to prepare online documentation or online collections of reference materials. These can be large collections of documents, often with some overall hierarchical structure (such as sections and subsections), but also with many hypertext links crosslinking these documents and linking them to other resources, such as programs that can search the documents for keywords, or to other Internet sites containing related information.

There are several design issues to consider when developing large document collections:

1. Each document should be small, usually no more than two or three screens full of data. The advantage of the hypertext model lies in the linking of various components of the documentation package. This advantage is largely lost if you are viewing a single, huge document containing hundreds or thousands of lines of displayed text. Although you can build hypertext links within a document to other points inside the same document, this is generally more difficult to navigate than a collection of smaller files.
2. Each document should have navigation tools in the form of hypertext links that connect the document to other documents in the hierarchy and to general navigation points within the collection. Thus, each page

should have links to *next* and *previous* documents (if there is an obvious order to the pages) and to a Table of Contents or the section heading. If the document is big, say more than two or three screens full of text, then it might be a good idea to place the navigation icons both at the top and bottom of the document. This makes them easier to find.

3. Documents should show a consistent presentation style. Each document should be consistently designed, with the same heading structure, the same navigation icons, and similar contents outlines. This makes it easy to get the *feel* for the collection, and also makes it possible to index or catalog the collection using programs that take advantage of this structure.

Artistic license is, of course, allowed! But these are general guidelines, based on experience, that help to make your work more pleasing and easier to use.

```
<html><head>
<title> HR element in HTML </title>
</head><body>
[<a href="htmlindex.html">Index</a>]
[<a href="body.html">Up</a>]
[<a href="lists_reg.html">Back</a>]
[<a href="entities.html">Next</a>]

<H1> 4.7 Horizontal Ruled Line </H1>

The HR element is used to draw a horizontal dividing line
completely across the screen. This can be to logicallyseparate
blocks of text, or to separateicon lists from the body of the text.

<p> The HR element is empty (you don't need a
<code>&lt;/HR></code>).
```

```
<h2> Example </h2>
The following shows an example of the use of &lt;HR> and the
resulting rendering (on your browser).
<blockquote>
<pre>
The following document is scanned from the back of
a cereal box. To see the scanned image, press the
icon at the bottom of the text ....
&lt;HR>
&lt;H1> MIGHTY CHOKEE-OS! &lt;/H1>
The cereal of chocolate deprived kiddies everywhere!
&lt;p> Aren't you lucky your parents love you enough
to buy you CHOCKEE-OS!
&lt;p> Remember to ask Mom and Dad for NEW SUPER
CHOCKEE-OS, now with Nicotine!!
</pre>
</blockquote>
<p> <b> This is rendered as:</b>
The following document is scanned from the back of a
cereal box. To see the scanned image, press the icon at
the bottom of the text ....
<HR>
<H1> MIGHTY CHOKEE-OS! </H1>
The cereal of chocolate deprived kiddies everywhere!
<p> Aren't you lucky your parents love you enough
to buy you CHOCKEE-OS!
<p> Remember to ask Mom and Dad for NEW SUPER CHOCKEE-OS,
now with Nicotine!!
<hr>
<p> [<a href="htmlindex.html">Index</a>]
[<a href="body.html">Up</a>]
[<a href="lists_reg.html">Back</a>]
[<a href="entities.html">Next</a>]
</body> </html>
```

Figure 1.19 HTML listing for the document ***hrule.html***, a typical text-only HTML document.

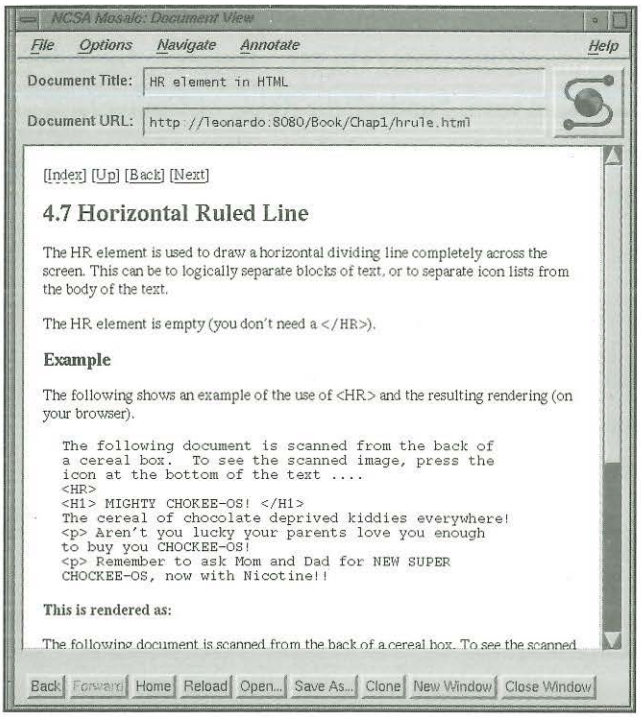


Figure 1.20 Mosaic For X-Windows display of typical text-only document (***hrule.html***).

Figures 1.19 and 1.20 show an example HTML document from a large collection of related files. This particular example is one of approximately 70 documents that discuss various aspects of the HTML language (although, in significantly less detail than in this book!). This collection of documents can be accessed at the URL:

```
http://www.utirc.utoronto.ca/HTMLdocs/NewHTML/htmlindex.html
```

This is a *flat* collection of documents, in that all the files lie in the same directory and are linked together in a serial fashion. However, there are many hypertext links relating the documents together (for example, one document discussing the IMG element has a sentence mentioning URLs, which, in turn, contains a hypertext link to a document giving a more

detailed discussion of URLs). The documents are also ordered hierarchically. Thus, this example document discussing the **HR** element is "under" the **BODY** document which is, in turn, "under" the Table of Contents. The Table of Contents page contains hypertext links to all the documents in this collection and is an easy tool for quickly finding and accessing a particular section.

Note the navigation text icons at the top of the page. There are four navigation buttons: *Index*, *Up*, *Back*, and *Next*. The *Index* button takes you directly to a Table of Contents page, while the Up button takes you one level up in the hierarchy; in this case, to the BODY page. The *Back* button takes you backwards to the preceding document in the hierarchy, while the *Next* button takes you forwards to the next document. The *Back* and *Next* buttons are the ones to use if you want to read the document straight through.

PRE ELEMENT

As an HTML aside, you should note the HTML **PRE** element in Figure 1.19. The **PRE** element is used to enclose preformatted text for presentation as is, preserving the space characters and carriage returns typed into the HTML document and displaying the characters using a fixed-width typewriter font. Thus, you should use the **PRE** element to display computer codes, text examples, or verbatim text sequences. This is also the only way that you can currently create tables for display in an HTML document, since this is the only element that preserves the horizontal spacing needed to align columns.

The usefulness of the **PRE** element is well illustrated in Figure 2.18 (Chapter 2), where it is used to display both program code and a small table. You can include character emphasis elements within a PRE element so that you can use the **STRONG** and **EM** elements to emphasize certain text strings. You can also include hypertext anchors. However, most other

elements, such as headings, list elements, or the ADDRESS element, are prohibited.

```
html>
<head>
<title> HTML Documentation Table of Contents</title>
</head>

<body>

<h1> HTML Documentation Table of Contents </h1>

<dl>
<dt> <a href="htmlindex.html"> Table of Contents (this page)</a>
<dt> <a href="about_the_author.html"> About the Author</a>
</dl>
<ol>
<li> <a href="intro.html"> Introduction to this Document</a>
<li> <a href="html_intro.html"> Introduction to HTML </a>
<ol>
    <li> <a href="elements.html"> HTML Elements </a>
    <li> <a href="doc_struct.html"> HTML Document Structure</a>
    <li> <a href="naming.html"> HTML Document Naming Scheme </a>
</ol>
<li> <a href="head.html">HEAD</a> of an HTML Document
     <ol>
     <li> <a href="title.html"> TITLE</a>
     <li> <a href="isindex.html"> ISINDEX</a>
     <li> <a href="nextid.html"> NEXTID</a>
     <li> <a href="link.html"> LINK</a>
     <li> <a href="base.html"> BASE</a>
```

```
    </ol>
<li> <a href="body.html"> BODY</a> of an HTML Document
    <ol>
    <li>  <a href="headings.html"> Headings</a> (Hn)
    <li>  <a href="paragraph.html"> Paragraphs</a> (P)
    <li>  <a href="line_break.html"> Line Breaks</a> (BR)
    <li>  <a href="image.html"> Inlined Images</a> (IMG)
    <ol>
    <li>   <a href="image-examples.html"> Examples </a> of Images
    </ol>
    <li>  <a href="anchors.html"> Hypertext Anchors</a> (A)
    <ol>
        <li>  <a href="A_href.html">Link to</A> an object (HREF)
        <li>  <a href="A_name.html">Link from </A>an object (NAME)
        <li>  <a href="A_rel.html">Relationship</A> between objects (REL)
        <li>  <a href="A_rev.html"> Relationship</A> between objects (REV)
        <li>  <a href="A_urn.html"> URN </A>
        <li>  <a href="A_title.html">TITLE </A>
        <li>  <a href="A_methods.html">How</A> to link  (METHODS)
    </ol>
      .
      .
      .
    </ol>
<li> Stepping up to <a
href="http://info.cern.ch/hypertext/WWW/MarkUp/HTMLPlus/htmlplus_1.html">HTML+</a>
<li> <a href="bibliography.html">Bibliography</a>
</ol>
</body>
</html>
```

Figure 1.21 HTML source document for the Table of Contents page ***htmlindex.html***. Some of this document has been omitted to save space. The rendering of the document is shown in Figure 1.22.

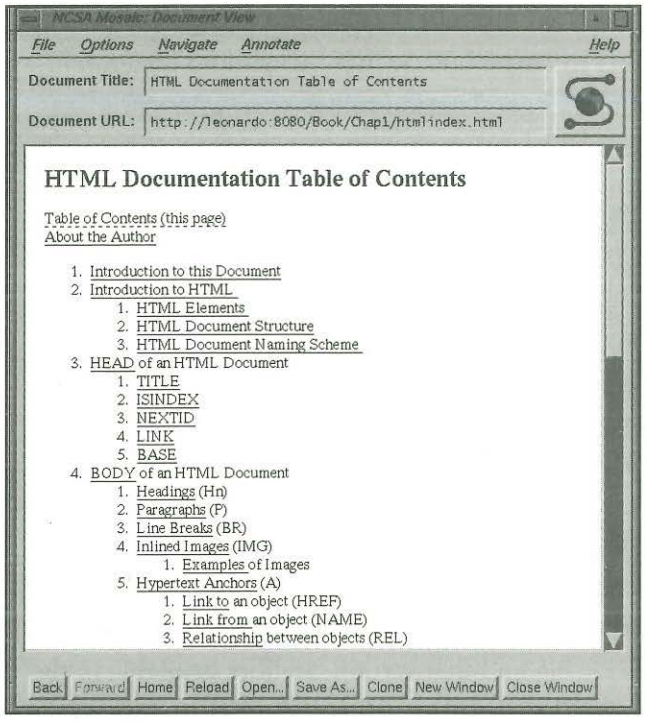


Figure 1.22 Table of Contents page for the HTML document collection that contains the file ***hrule.html*** (shown in Figure 1.19). This collection of documents can be accessed at:

```
http://www.utirc.utoronto.ca/HTMLdocs/NewHTML/htmlindex.html
```

These documents are not up to date.

ORGANIZATION

Figure 1.22 shows the HTML Table of Contents document for this collection (the listing is in Figure 1.21), although only part of this appears within the displayed window. Notice how this gives a complete overview of the document tree, including the relative placement of the sections in the hierarchy and the hypertext links to each section. This Table of Contents was constructed by hand—a tedious process, to say the least. Fortunately, there are now programs that can automatically generate a hypertext Table of

Contents directly from the HTML document collection, using the headings embedded in the documents to create both section names and a hierarchical organization. This is another good reason to use appropriate heading elements. Information about these indexing tools is provided in Chapter 5.

There are several other organizational features that you may also want to include. Often, a printed version of a document collection is useful. If this is desired, you can combine the documents together and present them as a single, large file that clients can download and read as HTML or print (from their browsers) as a single text file. However, you should let them know what to expect. For example, you can add lines in the Table of Contents or some other page that say something like:

```
<p> This entire archive of documents is also available as a <A
HREF="alldocs.html"> concatenated HTML documents </A> (198 Kbytes), suitable for
printing.  Note, however, that the hypertext links in this document have been
removed.
```

This guides the user to a document that can be both viewed and printed, but also warns that the file is big and that certain facilities available in the discrete files are not present. They can then choose whether to click on the phrase "concatenated HTML documents" and access this resource.

ARCHIVING

In some cases, you might want to make the entire document collection available as an archive. Then, if someone is making extensive use of the documents, they can copy down the entire collection and install it on their own machine, reducing the load on your server and increasing the speed with which they can access the material. If you are using a PC, you most likely will make such an archive using the **PKZIP** package. This allows you to archive files and directories into a single compressed file, usually with the filename extension *.zip*. Thus, you could **pkzip** all the files in the document collection into a file called *alldocs.zip*. On a Macintosh, the canonical program for archiving is **StuffIt**: The resulting archive would be named

alldocs.sit. UNIX users will use a program called **tar** (for tape archiver), which would result in the archive file *alldocs.tar*. UNIX also has two programs for compressing programs, **compress**, which places a *.Z* at the end of the compressed filename, and **gzip** (for GNU zip, which is distinctly different from **PKZIP**), which places a *.z* at the end of the compressed filename. This would yield the compressed archive files *alldocs.tar.Z* (using **compress**) or *alldocs.tar.z*. (using **gzip**). If you were generous in preparing archives for multiple platforms, you might then have, in your HTML documents, a section pointing to these files. An example of such a document is shown in Figures 1.23 and 1.24.

Many HTTP server programs allow you to control access to certain files or directories on the server and restrict access to files to authorized users. These facilities allow you to control access to archives such as this, should there be copyright problems associated with the archives or other reasons to control access.

As a final HTML aside, note that Figure 1.24 illustrates how lists can be nested. Here, an unordered list element (**UL**) is nested inside an *ordered list* element (**OL**):

```
<UL>
   <LI> <A HREF="alldocs.zip">.........
   .
   .
   .
   <UL>
     <LI> (This is a concatenation of the .....
   </UL>
</UL>
```

Recall, however, that you cannot put lists inside headings, or headings inside lists.

```
<HTML>
<HEAD>
<TITLE> Archives of this Documentation </TITLE>
</HEAD>
<BODY>

<H2> Document Archives </H2>

<p> Archives of the document collection are available in the following
formats:

<UL>
<LI> <A HREF="alldocs.zip">  alldocs.zip</A>   (138 Kbytes) -- <EM> DOS PKZIP  </EM>
<LI> <A HREF="alldocs.sit">  alldocs.sit</A>   (532 Kbytes) -- <EM> Macintosh
Stuffit</EM>
<LI> <A HREF="alldocs.tar">  alldocs.tar</A>   (527 Kbytes) -- <EM> UNIX tar </EM>
<LI> <A HREF="alldocs.tar.Z">alldocs.tar.Z</A> (133 Kbytes) -- <EM> UNIX tar
(compressed)</EM>
<LI> <A HREF="alldocs.tar.z">alldocs.tar.z</A> (104 Kbytes) -- <EM> UNIX tar
(gnuzipped) </EM>
<LI> <A HREF="alldocs.html"> alldocs.html </A> (523 Kbytes) -- <EM>
Concatenated HTML documents </EM>
<UL>
<LI> (This is a concatenation of the HTML documents, suitable for
printing from a browser. The Hypertext links have been removed).
</UL>
</UL>
</BODY>
</HTML>
```

Figure 1.23 Example HTML document ***src_link.html*** that contains links to alternative formats of a document collection. Clicking on the items retrieves the archives to the client's machine.

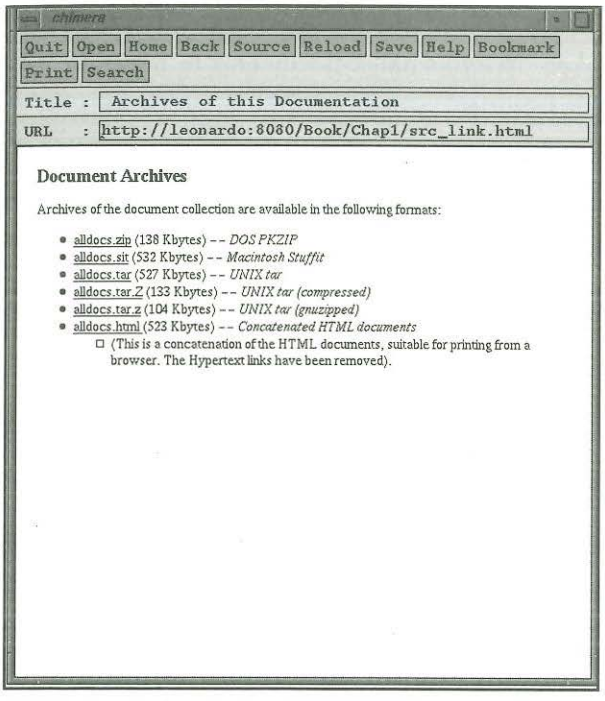


Figure 1.24 Rendering of the HTML document ***src_link.html*** (shown in Figure 1.23) by the UNIX **chimera** browser.

LESSONS FROM EXAMPLE 4

1. Collections of documents should be consistently designed to make them easy to navigate. You should also try to create a hypertext Table of Contents. There are programs available that can help you do this, some of which are discussed in Chapter 5.

2. You should use navigation icons (or text) as hypertext links to help the user navigate through the document collection. Make sure that you provide a text-only navigation option for users who are using a nongraphical browser.

3. Single HTML documents should be small and self-contained. Larger documents should be broken up into smaller documents to best take advantage of the hypertext facilities.

4. Sometimes a *flat* printable text document is also desirable. You can concatenate your HTML files together to make such a document, and then create a hypertext link from your collection to this document, but be sure to include information about the size of this file (if it is large) next to this link so that the user knows what to expect.

EXAMPLE 5: IMAGES, MOVIES, AND SOUND FILES

As mentioned earlier, WWW browsers can display only GIF or X-Bitmap images and often restrict GIF images to less than 50 or so displayed colors per image. The discussion also stated that small images are advantageous, since large images can take a long time to download and are often irritating for that reason alone.

LINKING LARGE IMAGES

However, sometimes these restrictions are unreasonable. Suppose you need to display a page containing a large image that is an important part of your presentation. For example, it may be an image of a campus map that can be *clicked* on to access information about various campus buildings, in which case, a tiny image is not appropriate. Alternatively, you may have truly high-quality images containing 256 colors, or perhaps non-GIF format images, and want to make them available for viewing. Perhaps you even have movie or sound files. How can these be included in a document and presented to your clients?

Large GIF images can always be added to an HTML document. The key to good document design is simply to warn the user, at the hypertext anchors pointing to a file, that it contains a large image. This is a useful way of directing users to large active images (see creating active images in Chapter 5) and gives the user the option of disabling image loading when accessing the document, or to not access the document altogether. You are still, however, limited to GIF or X-Bitmap images, and by browsers that may limit the numbers of distinct colors that a single image can use.

A *thumbnail sketch* is a particularly useful way of linking to a large image. A thumbnail is simply a reduced size *icon* of the actual image or of some characteristic portion of the image. These are easy to make with almost any commercial or public-domain image editing program. You can then include the thumbnail in your document, and make a hypertext link from the thumbnail to the document containing the large image, or to the image itself. This is what was done in Figures 1.25 and 1.26, where the small images are links to larger images and, in this case, also to movie files. In this example, the thumbnail of the larger GIF image is only 1,500 bytes long, one-tenth the size of the original file. Note that sizes of the linked documents are also given (these are big files!). It is always good practice to give this information when files are large.

You should also indicate the data type of the file, as in Figure 1.26, where the text indicates that the linked image file is in GIF format and that the linked movie is in MPEG format. There are many different movie and image formats, and most clients are capable of displaying only a subset of them. It is always a good idea to give the format of your large data files so that users can avoid downloading files they cannot use.

```
<HTML>
<HEAD>
<TITLE>Simulated Vortex Dynamics in a Porous-Body Wake</TITLE>
</HEAD>

<BODY>
<H1>Simulated Vortex Dynamics in a Porous-Body Wake</H1>

This video presents the result of a numerical simulation on the wake
generated by a porous body.  The wake flow is simulated by inserting
small-scale discrete vortices into a uniform stream,  and  the colors
in the video represent the magnitude of vorticity.  The initial flow
field is subjected to a small perturbation based on experimental data.
The evolution of the wake flow is  manifested  by the merging  and
interactions of the small-scale vortices.<P>

The objective of this investigation is to study the merging and
inter-action processes of vortices and the formation of large eddies in
the flow.  Such an investigation is of importance to many flow-related
industrial and environmental problems, such as mixing, cooling,
combustion and dispersion of air-borne or water-borne contaminants.<P>
<HR>
<B> <A HREF="legend.gif"><IMG SRC="legicon.gif"
ALIGN=Bottom> Initial flow</A> and color legend for vorticity.</B>
(14.5 KB gif image)<p>
<HR>
<B> <A HREF="flow.mpeg"><IMG SRC="vortex.gif" ALT="[movie icon]"
ALIGN=Bottom> Visualization</A> of the evolution of the wake flow.</B>
(0.38 MB mpeg-1 movie)<p>
</BODY>
</HTML>
```

Figure 1.25 The HTML document ***vortex.html***, showing links from image icons to full-size images and video sequences. Figure 1.26 shows the rendering of this document by the **Mosaic for X-Windows** browser.

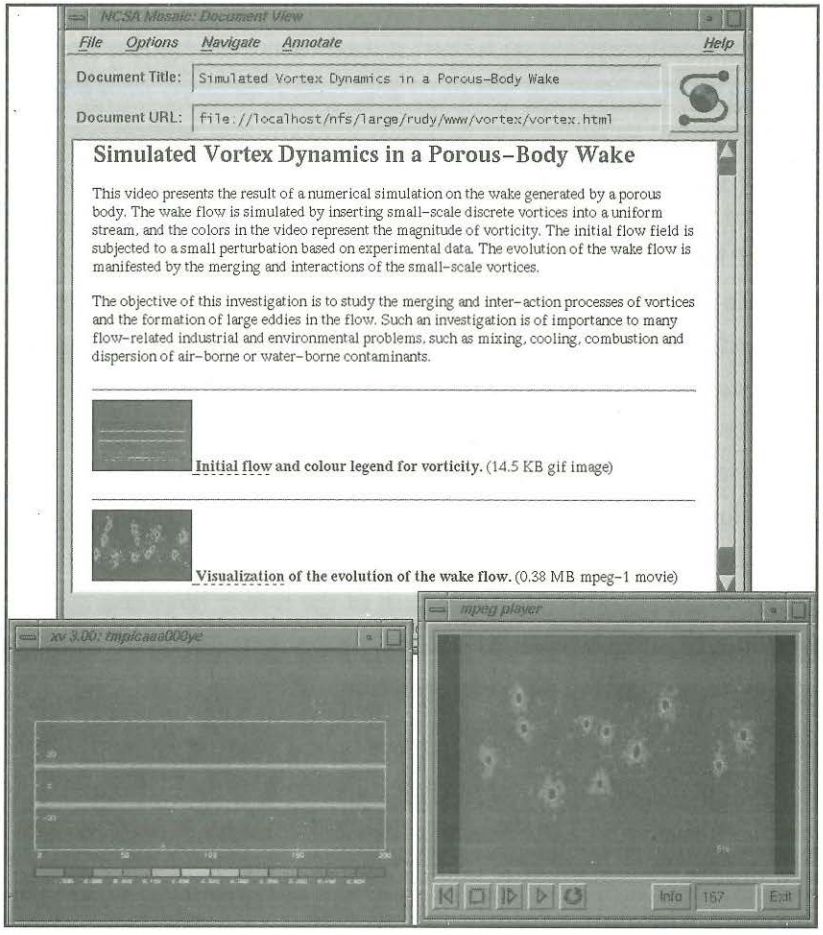


Figure 1.26 The **Mosaic for X-Windows** rendering of the document listed in Figure 1.25, showing thumbnail image icons linked to full-sized image files, movie files, and sounds. The picture overlaid on the browser resulted from clicking on the image icon at the top of the screen, while the movie-playing window was launched by pressing on the icon at the bottom of the screen. This page and the associated images and movies are courtesy of Rudy Ziegler of the University of Toronto Instructional and Research Computing Group, while the displayed data were provided by Z. Huang, J.G. Kawall, and J.F. Keffer of the Department of Mechanical Engineering at the University of Toronto.

So far, our hypertext links have been to HTML documents or to HTML documents containing images via the IMG element. What happens if the links connect to other media, such as image files, movie files, or sound files? Most World Wide Web browsers are not capable of displaying these data formats. So, what do they do?

HELPER APPLICATIONS

The answer lies in so-called *helper* or *viewer* applications, which are programs on the user's computer that can be used to display images, movies, or sounds that cannot be displayed by the browser itself. Thus, in Figure 1.26, the large-screen image was produced by clicking on the upper image icon in the browser window, causing the browser to retrieve the data accessed by this link, acknowledge the data to be an image file, and launch the appropriate helper application to display the image (in this case, the UNIX image viewing program **xv**). In the case of the movie file, the browser knew that the data was an MPEG movie, so it started up the program **mpeg_play** to display the video information.

How does the browser know what a file contains and what to do with it? If the data comes from an HTTP server, then the server, as part of the HTTP protocol, *explicitly tells the browser* the data content. It does this with a special header message that is sent to the client just before the actual data. This message is actually a *MIME content-type* header, which, for the GIF image file, looks like:

```
Content-Type: image/gif
```

and, for the MPEG movie, is:

```
Content-Type: video/mpeg
```

When the browser receives this message, it looks in its own database of *helper* applications to find the program that matches this *MIME type*. If it finds a program to help, it passes the data to the program and lets it do its job.

If the data comes from an FTP server, or if the browser is accessing the file from the local machine and not from a server, then the browser must *guess* at the data content. It does this from the filename extension. Each browser has a database that matches filename extensions to the appropriate MIME type, and uses this database to determine the MIME types of files accessed locally or via FTP. In general, this database will map the *.gif* suffix to the `image/gif` MIME type, and the suffixes *.mpeg*, *.mpg*, or *.mpe* to the `video/mpeg` MIME type. These lists have to be updated if you add a new filename extension. With Macintosh and some Windows browsers, the lists can be edited from a pull-down menu.

There are literally dozens of MIME types and dozens of possible filename extensions. A detailed list of MIME types is given in Appendix B, while the usage of MIME types is discussed in more detail in Chapter 4.

LESSONS FROM EXAMPLE 5

1. Warn a user when you present a link to a large image document or file.
2. List the data format for large image, audio, or movie files so that the user can tell if the file is in a format they can actually use.
3. You can use icons to link to larger image or movie files. This lets the user know what to expect and is often a good graphical addition to your document.

EXAMPLE 6: FILL-IN FORMS

This final example looks at the HTML FORM element, which allows you to solicit user input by constructing HTML documents containing fill-in forms. Using this element, a designer can build a document containing checkboxes, radio boxes, pull-down lists, text windows, and menus; and can configure this FORM to send the data gathered by these buttons and

boxes to a program on an HTTP server. For example, FORMs can be used to collect data for a database search; solicit data for an online questionnaire; accept electronic text for submission to a database; or solicit electronic messages for forwarding to a particular user.

The example document in Figure 1.27 shows this latter case: namely, a FORM that allows the user to type in a text message for forwarding to a recipient selected from a list. Figure 1.28 shows how this form is rendered by **Mosaic for X-Windows**, while Figure 1.29 shows the rendering by **lynx**. Finally, Figure 1.30 shows what **MacMosaic** Version 1.0.3 does with this HTML document; this older version of **MacMosaic** does not understand the **FORM** element.

Let's first look to Figure 1.27 and to the `<FORM. .>` tag beginning with the line:

```
<FORM  ACTION="http://side.edu/cgi-bin/send_note">
```

FORM ELEMENT

This line starts the **FORM** element and ties the data of the form to a particular program (***send_note***) on the indicated HTTP server. All a **FORM** element does is collect data: It doesn't do any processing of the data, so the only way you can get a form to do anything useful is to send the data gathered by it to a program on the server. The `<FORM. . .>` tag in Figure 1.27 indicates that the data gathered by the form is to be sent to the program ***send_note***. Data is sent to this server-side when the user presses the *Send Message* button at the bottom of the page. The FORM and the program ***send_note*** must be designed together for the program to understand the message sent by the FORM.

The program ***send_note*** takes the data sent by the client and processes it to complete the task. In this example, the program might take the data sent by the form and compose an electronic mail message to be sent to the intended person. We talk more about gateway programs and how data is actually sent from the FORM to the gateway program in Chapters 4 and 5.

```
<HTML><HEAD>
  <TITLE> Example of an HTML FORM  </TITLE>
</HEAD>
<BODY>
<H1> Example of an HTML FORM  </H1>
<FORM ACTION="no_action">
Data entered into a FORM is sent to a program on the server
for processing.  If you see a button at the end of this sentence
then your browser supports the HTML FORMs element.
--[<INPUT TYPE="checkbox" NAME="button" VALUE="on">]--
If you do not see a button between the square brackets go to the
<A HREF="text_only.html"> text-only interface </A>. </FORM>
<hr>
<FORM  ACTION="http://side.edu/cgi-bin/submit_abstract">
  <p> <STRONG> 1) Send this note to: </STRONG>
  <SELECT NAME="mailto_name" >
    <OPTION SELECTED> Martin Grant
    <OPTION> Jack Smith
    <OPTION> Bruce Lee
    <OPTION> Anna Mcgarrigle
    <OPTION> Kate Bush
    <OPTION> Spike Lee
    <OPTION> Diane Koziol
    <OPTION> Ross Thomson
    <OPTION> Ann Dean
  </SELECT>

  <p> 2) <STRONG>Give your e-mail address: </STRONG>
      This indicates who sent the letter
  <p> <INPUT TYPE="text"  NAME="signature"
       VALUE="name@internet.address" SIZE=60>

  <p> <STRONG> 3) Message Body: </STRONG>
```

```
    <p>
    <TEXTAREA COLS=60 ROWS=8 NAME="message_body">
     Delete this message and type your message into this
     textbox.  Press the "Send Message" button to send it
     off. You can press the "Reset" button to reset the
     form to the original values.
    </TEXTAREA>
    <P>
    <INPUT TYPE="submit" VALUE="Send Message"> <INPUT TYPE="reset"> (reset form)
    </FORM>
    </BODY>
</HTML>
```

Figure 1.27 The HTML source code for the document ***form.html***.

By comparing the HTML source document in Figure 1.27 and its rendering in Figure 1.28, you can see some of the several input items that can go inside a form. This example shows: a **SELECT** element pull-down menu (where you select the name of the person to whom you wish to send the message—the possible names being given by the **OPTION** element); a single-line text **INPUT** element (where you type in your e-mail address); and a **TEXTBOX** element (where you type the body of your message). Several other input elements are available and are described in Chapter 2. The elements, **SELECT**, **INPUT**, **OPTION**, and **TEXTBOX**, can appear only inside a **FORM**.

Every **FORM** input element, namely **INPUT**, **SELECT**, or **TEXTBOX**, takes one key attribute. This is the **NAME** attribute, which associates a *variable name* to the data associated with this input element; for example, `NAME="mailto_name"` and `NAME="button"`. These names are used to differentiate between the data associated with the different input elements. Some elements, such as the **INPUT** element, can assign a default initial *value* to the named variable using the **VALUE** attribute. An example is the element:

```
<INPUT TYPE="checkbox" NAME="button" VALUE="on">
```

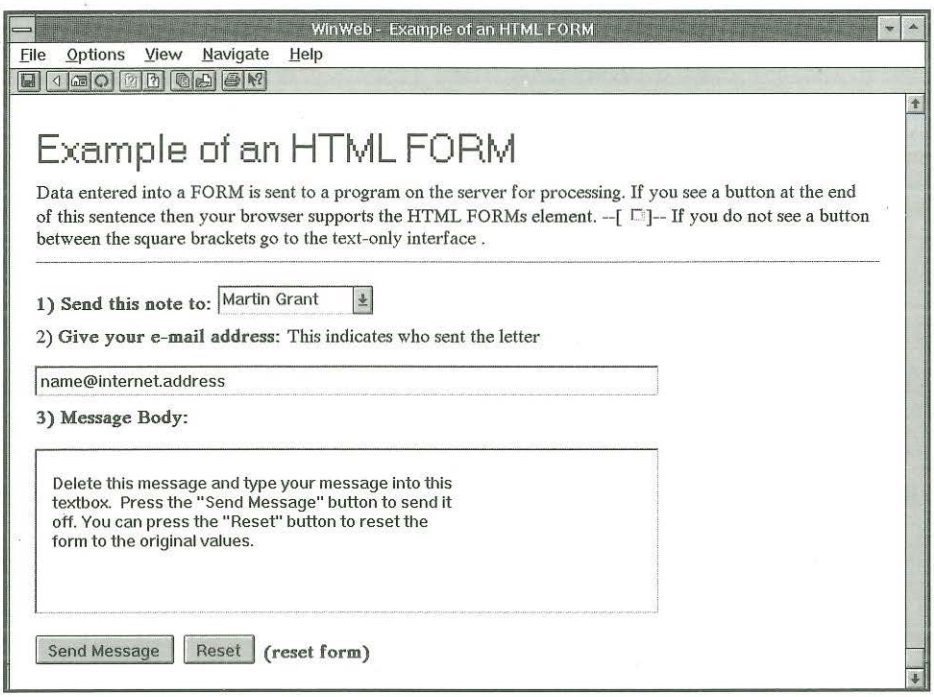


Figure 1.28 Mosaic for X-Windows rendering of the ***form.html*** document (the document source is displayed in Figure 1.27). The FORM fill-in elements are clearly evident.

which assigns the value `"on"` to the name `"button"`. These values can be subsequently changed by user input, by selecting a different entry from a pull-down menu, typing text into a box, or pushing checkboxes or buttons.

When you press the special *Send Message* button (a special **INPUT** element with the attribute `TYPE="submit"`), the data in the form is sent to the server as a collection of *name/value* pairs, where *name* is the value assigned to the **NAME** attribute of an element, and *value* is the value assigned by the user's input. For example, the preceding checkbox input element would send the pair *button* and *on*.

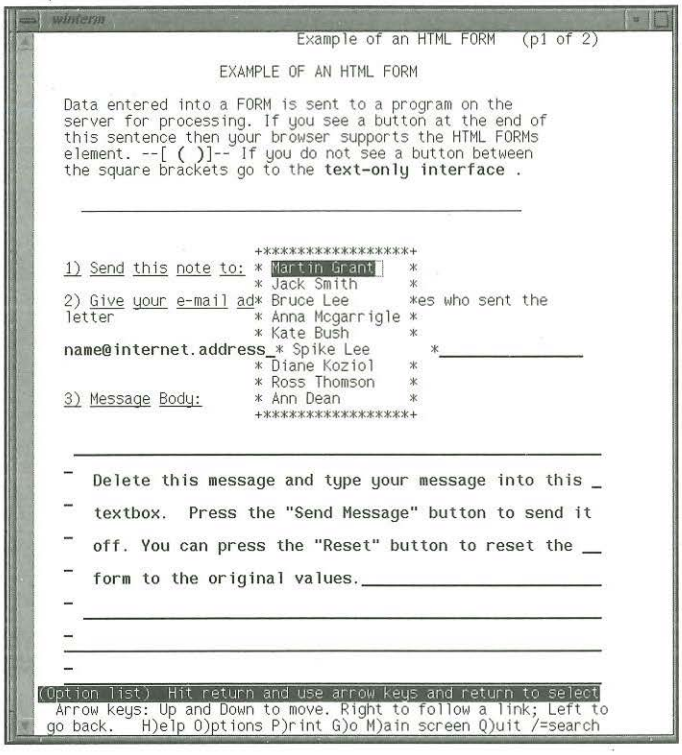


Figure 1.29 The **lynx** browser rendering of the ***form.html*** document (the document source is displayed in Figure 1.27). The FORM fill-in elements are clearly evident: In general, **lynx** gives written instructions at the bottom of the screen to help you properly manipulate the form.

The server program sorts out what the data means by matching the *names* in the message to names the program is designed to recognize. This illustrates why a form and the associated server-side program must be designed simultaneously.

FORM RESTRICTIONS

As with all HTML elements, the FORM element has restrictions on where it can be placed. A FORM cannot be inside a heading, inside another FORM, or inside character emphasis markups, such as a STRONG or EM element. However, a FORM can contain headings, character markup

elements, and even lists. Again, the details of the nesting rules are given in Chapter 2.

Figure 1.29 shows the same FORM as displayed by **lynx**. The **lynx** browser can display all the major FORM elements and provides instructions at the bottom of the screen to explain how to fill in the different FORM elements.

The last thing to notice is the little button at the top of Figures 1.28 and 1.29. This button, which does not actually do anything (note that there is no associated `<INPUT TYPE="submit" >` button), is designed to test the capabilities of the browser. There are still a few commonly used browsers that are unable to process the **FORM** element (an example is **MacMosaic** Version 1.0.3). It is therefore a good idea to test FORM capabilities and let the user know if he or she will be unable to properly view a document. For example, Figure 1.30 displays the same HTML document as Figure 1.28, but by using the **MacMosaic** 1.0.3 browser. You can see that the small box in the first paragraph is missing (as are all the other FORM boxes—in general, browsers simply ignore tags that they do not understand). The first paragraph explains why the rest of the page looks so odd, and can direct users to alternate documents designed for FORM-incapable browsers, should such pages be available.

LESSONS FROM EXAMPLE 6

1. The document developer can use the HTML **FORM** element to solicit user input. However, each FORM document must send the data it gathers to a server-side gateway program designed to analyze the form data. This gateway program is specified by the **ACTION** attribute to the **FORM** element. The program and form must be designed together.
2. An HTML **FORM** can contain several input elements, namely **INPUT**, **SELECT**, and **TEXTBOX**. These elements can appear only inside a **FORM**.

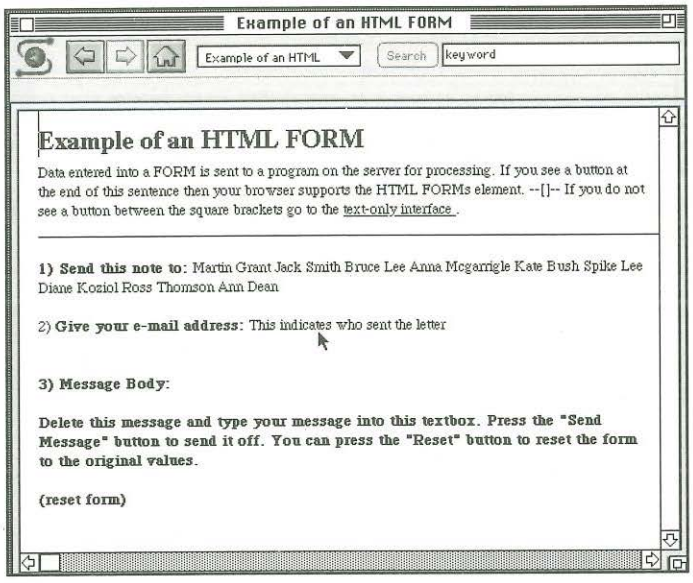


Figure 1.30 MacMosaic Version 1.0.3 rendering of the ***form.html*** document (the document source is displayed in Figure 1.27). This browser cannot display FORM elements

3. A FORM cannot be inside a heading element or inside another FORM—FORMs cannot be nested. However, heading, list, and even PRE elements can be inside a FORM.

REFERENCES

The WWW USENET newsgroups are a good source of information about HTML and the World Wide Web in general. The newsgroups contain up-to-the-minute information on WWW happenings and are a good place to post HTML questions. The newsgroups are:

comp.infosystems.www.providers	Issues related to publishing material on the Web, such as the HTML language and the development of server gateway programs.

comp.infosystems.www.users	User-related issues, such as configuring browsers, finding browsers, or discussing browsers.
comp.infosystems.www.misc	Other miscellaneous topics.
bionet.software.www	Discusses applications of WWW software in the biological sciences.

Frequently Asked Questions (FAQ) lists are posted regularly to these newsgroups. The entire FAQ list for all the newsgroups is also available in hypertext form at:

```
http://sunsite.unc.edu/boutell/faq/www_faq.html
```

User surveys of WWW software and resources can be found at:

```
http://www.gatech.edu/pitkow/survey/survey-9-1994/Survey_Home.html
```

General References:

```
http://akebono.stanford.edu/yahoo/Computers/World_Wide_Web/HTML/
http://oneworld.wa.com/htmldev/devpage/dev-page.html
http://cbl.leeds.ac.uk/nikos/doc/repository.html
http://coney.gsfc.nasa.gov/www/sswg/candy_style.html
```

CHAPTER

2

HTML IN DETAIL

This chapter is a detailed exposition of the *HyperText Markup Language*; it is written from a document developer's point of view and is designed to help document authors create valid HTML documents. It presents a detailed description of every HTML element and of allowed hierarchical relationships amongst these elements. This section assumes a basic understanding of HTML at the level outlined in Chapter 1 of this book.

INTRODUCTION TO HTML

As mentioned in Chapter 1, the HyperText Markup Language is designed to specify the *logical* organization and formatting of text documents, with extensions to include inline images, fill-in forms, and hypertext links to other documents and Internet resources. HTML is designed to be a platform-independent document representation format that:

- Is not bound to a particular hardware or software environment
- Represents the logical structure of a document, and not its presentation

This emphasis reflects the fact that, in a distributed environment, individuals viewing a document can use many different browser programs of different capabilities. For example, it is of little use specifying that a particular piece of text must be presented with a 14-point Times Roman font if the person viewing the document is sitting in front of a VT-100 terminal. For this reason, HTML does not specify details of the document typesetting and instead marks logical document structure, such as headings, lists, or paragraphs. The details of the presentation of these elements are left to the browser.

HTML is defined in terms of the International Standards Organization (ISO) Standard Generalized Markup Language (SGML). SGML is a complex system for defining types of structured documents and for defining markup languages that can represent these document types. HTML is just one instance of this process. The details of SGML are complex and, fortunately, not important to an HTML document developer. One component that is useful, however, is the SGML *definition* of the HTML syntax, which is contained in a special SGML document called a *Document Type Definition,* or *DTD*. This is a simple text file, often having an imaginative name, such as ***html.dtd***. This file can be used, in combination with a program called **sgmls,** to validate the syntax of any HTML document. The "References" section at the end of this chapter suggests places where you can obtain the official DTD file for HTML, while Chapter 6 discusses using **sgmls** to validate HTML documents.

HTML is an evolving language and has undergone substantial evolution over the past few years. The new "*standard*" version of HTML is referred to as HTML Version 2.0, or HTML2.0. This chapter (and this book) describes writing HTML documents using the HTML2.0 specification.

However, it also presents some of the proposed but not yet official HTML features that are already available on a limited number of browsers.

This book is a guide to authoring HTML documents and should not be considered the definitive reference to HTML. For comprehensive details, look at the Internet Engineering Task Force (IETF) documents listed in the "References" section at the end of this chapter.

ALLOWED CHARACTERS IN HTML DOCUMENTS

As illustrated in Chapter 1, an HTML document is just a text document and can be created and edited with any text editor. An HTML document can contain any of the valid printable characters from the 8-bit ISO Latin-1 character set (also known as ISO 8859/1—see Appendix A). The 256 characters of the ISO Latin-1 character set consist of the 128 characters of the 7-bit US-ASCII character set (ISO 646), plus 128 additional characters that use the eighth bit. This extra set of 128 contains many of the accented and other characters commonly used in western European languages.

With most keyboards, it is difficult to type these non-ASCII characters. Partly for this reason, HTML has mechanisms for representing these characters using only 7-bit ASCII characters. These are called *character references* and *entity references*. For example, the character reference for the character "é" is `é` (the semicolon is necessary and terminates the special reference), while the entity reference for this same character is `é`. This mechanism is also useful for sending HTML documents by electronic mail, since many electronic mail programs mishandle 8-bit characters. However, transferring HTML documents containing 8-bit characters is not a problem with hypertext servers, since the HTTP access protocol always allows 8-bit transfers and the browsers all understand that the character set is ISO Latin-1.

In addition, some computers, such as a Macintosh or a PC running DOS, do not use the ISO Latin-1 character set for their internal representation of

characters (Microsoft Windows does use the ISO Latin-1 set); instead, they use a proprietary mapping between the binary codes and the characters they represent. Fortunately, this affects only the 128 non-ASCII characters so that restricting yourself to ASCII characters ensures an HTML document, while the character and entity reference mechanisms allow you to include characters from the full ISO Latin-1 character set.

Character sets are discussed in more detail in Appendix A. Figure A.1 shows the use of HTML entity references in an HTML document—this document, as displayed by a browser, is shown in Figure A.2.

SPECIAL CHARACTERS

Certain ASCII characters are treated as special in an HTML document. For example: The ampersand character (&) is used to indicate an entity or character reference; the left and right angle brackets (< >) are used to denote the markup tags; and the double quotation mark (") is used to mark literal strings within the markup tags. Since these characters are interpreted in a special manner, they cannot be used as normal characters—the parser will always try to interpret them in a special way. If you want these characters to appear in your text but don't want them to be interpreted as commands, you must include them as character or entity references. The special character and entity references for these four characters are outlined in Table 2.1.

Table 2.1 Special Characters in HTML

Character	Character Reference	Entity Reference
Left angle bracket (<)	<	<
Right angle bracket (>)	>	>
Ampersand sign (&)	&	&
Double quotation sign (")	"	"

When a browser parses an HTML file, it looks for the special characters and interprets them accordingly. Thus, when it encounters the string:

```
<H1> Heading string </H1>
```

it interprets the strings in the angle brackets as markup tags and renders the enclosed string as a heading. However, when the parser encounters a string like:

```
&lt;H1&gt; Heading string &lt;/H1&gt;
```

it interprets the < and > as entity references, and produces the string

```
<H1> Heading string </H1>
```

COMMENTS IN HTML DOCUMENTS

Comments are denoted by the special character strings <!-- and -->. The first string starts a comment, while the second ends it; anything between the two is a comment and is ignored. Comments are not displayed by the browser, even if they occur in the BODY of a document. There can be spaces between the -- and the > that end a comment, but the string <!-- that starts a comment must be present without any spaces between the characters.

Comments can span more than one line, but cannot nest or overlap. However, some browsers mishandle comments that span more than one line, so, for safety, you should include comment strings around every line you wish to *comment out*. You should also not use this mechanism to comment out HTML code that would otherwise be displayed, because some browsers mistakenly use the greater than sign (>) in regular HTML markup tags to prematurely terminate the comment.

Here are some examples of comments:

```
<!--    This is a comment -->

<!--    This is also a comment

This comment spans more than one line. Note that many browsers improperly
interpret comments that span multiple lines, so this usage should be
avoided.

  -->
```

HTML AS A MIME TYPE

Finally, we note that HTML is proposed as a MIME content type. MIME, for Multipart Independent Mail Extensions, is a scheme originally designed for sending mixed media mail messages (containing pictures, text, and other formats) using the standard electronic mail protocol. The MIME scheme uses MIME *content-type* headers to define the content of each different type of data being sent. In the World Wide Web, MIME types are used by the HTTP protocol to communicate the *type* of a document being served. When an HTTP server sends a file to a client, it includes, as a header to the data, a MIME content-type header indicating the type of data being sent. For example, a JPEG format image file being sent from a server to a client would have the message string:

```
Content-Type: image/jpeg
```

as part of the HTTP header that precedes the actual data.

HTTP servers treat HTML documents as just another MIME type. When an HTML document is served, it is preceded by the header:

```
Content-Type: text/html
```

which tells the browser that the document is HTML, and not just plain text. HTTP and MIME types are discussed in more detail in Chapter 4 and in Appendix B.

HTML ELEMENTS AND MARKUP TAGS

The overall structure of the HTML was covered in Chapter 1. The following is a review of the basic concepts, using the HTML document in Figures 2.1 and 2.2 as a simple example.

An HTML document is simply a text file in which certain strings of characters, called *tags*, mark regions of the document and assign special meanings to them. In the jargon of SGML, these regions are called *elements*. The tags are strings of characters surrounded by the less than (<) and greater than (>) signs. For example:

```
<H1>
```

is the *start tag* for an **H1** (a heading) element, while

```
</H1>
```

is the *end tag* for a title element. The entire **H1** element is then the string:

```
<H1> Environmental Change Project </H1>
```

Most elements are like this example and mark regions of the document into blocks of text, which, in turn, may contain other elements containing other blocks of text, and so on. You can think of a document as a hierarchy of these elements, with the complete hierarchy defining the entire document. Elements that mark blocks of text are often called *containers*. Some elements do not have any contents and are called *empty* elements. The **IMG** element (which inserts an inline image into a document) or the **HR** element (which draws a horizontal dividing line across the screen) are examples of empty elements that do not affect a block of text.

Each element has a name, which appears inside the tags and which is related to what the element means. For example, the **H1** element is used to mark a level-1 heading. Elements may also have *attributes*, which are quantities that specify properties for that particular element. For example, the **A** (hypertext anchor) element can take the **HREF** attribute, which specifies the target of a hypertext link. Most attributes, like **HREF**, are assigned *values*. For example, **HREF** is assigned the URL of the target document for a hypertext link, as in:

```
<A HREF="http://who.zoo.do/Ozone.html"> Ozone </A> layer
```

Attributes are often optional, in which case, they can be left out.

In some cases, the end tag is optional. This is the case when the end of an element can be unambiguously determined from the surrounding elements. As an example, look at the **LI** element in Figure 2.1. This element defines a single list item inside the **UL** unordered list element and does not require a `</LI>` end tag, even though the element is not empty. This is because the end of a given list item is implied by the next `<LI>` start tag or by the `</UL>` end tag ending the list.

```
<HTML>
<HEAD>
<TITLE> Environmental Change Project </TITLE>
<NEXTID N=33>
</HEAD>
<BODY>
<h1> Environmental Change Project </h1>
<p> Welcome to the home page of the Environmental Change Project.
This project is different from other projects with similar names. In our case
we actually wish to change the climate. For example, we would like hot beaches
in Northern Quebec, and deserts near Chicago.
<p> So how will we do this. Well we do the following
<ul>
<li> <A HREF="burn.html">Burn </A> more forests.
<li> Destroy the <A HREF="http://who.zoo.do/ozone.html">Ozone</A> layer.
<li> Breed more <A HREF="ftp://foo.do.do/cows.gif">cows</a> (for extra
   greenhouse gas).
</ul>

</BODY>
</HTML>
```

Figure 2.1 An example of a simple HTML document.

CASE-SENSITIVITY

The element and attribute names inside the markup tags are case-*insensitive*. Thus, in the above examples, the strings `<H1>` and `<h1>`, or

```
<A HREF="http://who.zoo.do/Ozone.html ">
```

and

```
<a hrEF="http://who.zoo.do/Ozone.html ">
```

are equivalent. Element and attribute names are often written in uppercase to make the document easier to read.

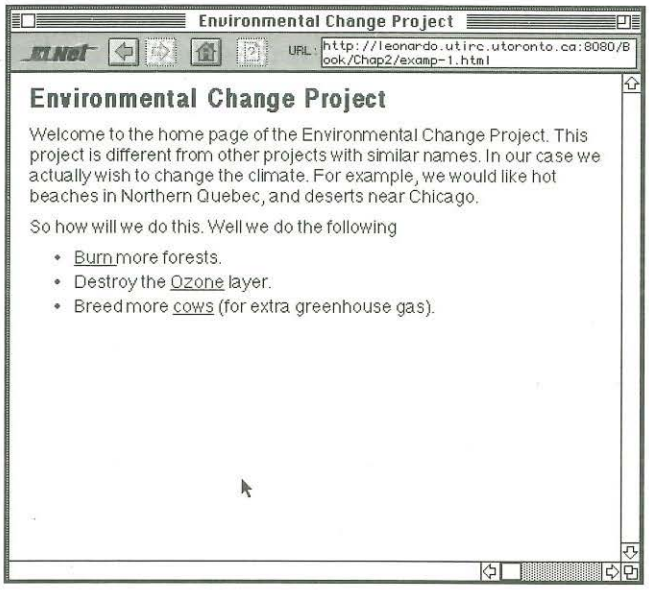


Figure 2.2 Display, using the **MacWeb** browser, of the document listed in Figure 2.1.

Attribute *values*, on the other hand, are often case-*sensitive*. An obvious example is the URL string assigned to the **HREF** attribute. A URL can contain both directory and filename information. Many computers allow both upper- and lowercase characters in file and directory names, so it is crucial that case be preserved. This is accomplished by enclosing the attribute argument in double quotes, as done in the preceding examples and in Figure 2.1. If quotes are omitted, the value may be promoted to uppercase when intepreted by a browser. Some attribute values are case-insensitive, in which case, quotation marks can be omitted.

EMPTY ELEMENTS

Some elements are considered *empty* and do not require end tags, such as the **IMG** and **HR** and **BR** (line break) elements. Formally, you can include end tags to match the start tags, provided you do not place any text between the two tags. In practice, however, several browsers do not func-

tion properly when you include end tags for empty elements, so this usage is discouraged.

ELEMENT NESTING

Elements are always *nested*, with this nesting reflecting the structure of the document (for example, emphasized text inside a paragraph inside a form inside the BODY). However, elements can *never* overlap. Thus, the structure

```
<A HREF=....> <EM> Burn </EM> </A> more forests
```

is valid HTML markup, while

```
<A HREF=....> <EM> Burn  </A> </EM> more forests
```

is not. In addition, all elements impose restrictions on what can be nested inside them, and on where they themselves can be nested. Details of allowed nestings will be presented later in this chapter.

Some browsers will let you get away with nesting errors so that such errors are often hard to spot. If you are lucky (or unlucky), you will get mail from someone who is wondering why he or she cannot properly view your document.

UNKNOWN ELEMENTS OR ATTRIBUTES

What does a browser do if it encounters tags or attributes within a tag that it does not understand? The HTML specification states that the browser should ignore these tags or attributes. Thus, if a browser does not understand the underline element U, it will take the string

```
<U> text stuff </U>
```

and display it as regular text, without the underline.

OVERALL DOCUMENT STRUCTURE

Every HTML document can be divided into two main parts: the *body*, which contains the part of the document to be displayed by a browser; and

the *head*, which contains information about the document but which is not displayed with it. These parts are defined by the BODY and HEAD elements, respectively. The resulting overall structure of an HTML document should then be:

```
<HTML>
    <HEAD>
        .... elements valid in the document HEAD
    </HEAD>
    <BODY>
        .... elements valid in the document BODY
    </BODY>
</HTML>
```

Note how Figure 2.1 follows this outline. The outer HTML element declares the enclosed text to be an HTML document. Directly inside this lies the HEAD and the BODY. The BODY contains the text and associated HTML markup instructions of the material you want displayed. The HEAD contains elements that define information *about* the document, such as its title or logical relationships with other documents. Certain elements can appear only in the HEAD, while others can appear only in the BODY. These elements and the overall structure of the BODY and HEAD are the focus of the remainder of this chapter.

HYPERTEXT MARKUP LANGUAGE SPECIFICATION: ELEMENT BY ELEMENT

This section contains a list of each HTML element, a description of the purpose of the element, a list of where the element can go, as well as examples of its use. This element list primarily describes elements from the HTML2.0 specification. Within this specification, there are several attributes that are marked as *proposed*. These are included here, but are marked

as proposed to point out their special status; they are less likely to be understood by browsers, so you should be cautious in their use.

Much work is underway on extension and revisions to the HTML—these efforts fall under the rubric HTML+. The HTML+ effort has defined elements for tabular data, sub- and superscripts, margin notes, and so on that will add important functionality to HTML documents. Indeed, several (but not all) browsers already implement some of these HTML+ features. This section mentions these more popular HTML+ elements and attributes, but marks them as HTML+ contributions to denote their experimental nature. In general, if you want your documents to be viewable by a wide audience, do not use these HTML+ elements, or provide alternate HTML documents for those people who are unable to view your more sophisticated creations.

KEY TO THIS SECTION

Because HTML is a hierarchical language, is it important to know not only how to use an element, but also where it can be used. This information is given in the four lines at the beginning of the description of each element. The general format looks like the following:

Usage:	`<NAME> ... </NAME>`
Can Contain:	**element list**
Can Be Inside:	**element list**
Attributes:	**attribute list**

Here is an example, using the LI list item element:

LI ELEMENT: LIST ITEM

Usage:	`<LI> ... (</LI>)`
Can Contain:	**characters, character highlighting, A, BR, IMG, DIR, DL, MENU, OL, UL, BLOCKQUOTE, FORM, P, PRE, [ISINDEX]**
Can Be Inside:	**DIR, MENU, UL, OL**
Attributes:	(**SRC**: proposed HTML+)

These four fields define the rules for using the different elements. The meanings of the four fields are:

Usage:	Shows how the element is used. An end tag indicates that an element is a container (the *Can Contain* field states what can go inside the element). If the end tag is surrounded by round brackets, then it is optional. If no end tag is given, then the element is empty.
Can Contain:	This field indicates what elements can go inside this element. The string "**characters**" indicates elements that can contain text. If the element is empty, the word "empty" appears here. In several places, the **ISINDEX** name appears enclosed by square brackets. This indicates that the element is allowed but that it should more appropriately appear in the **HEAD**.

Can Be Inside:	Indicates inside which elements this element can be placed. The preceding example indicates that the **LI** element can be inside a **DIR**, **MENU**, **OL**, or **UL** element.
Attributes:	This lists the names of the attributes that can be taken by the element. The word "none" means that the element takes no attributes. Elements that are proposed as part of the HTML2.0 specifications or as part of HTML+ are so indicated.

The following short forms are used in the Can Contain and Can Be Inside fields:

Hn:	The six heading elements **H1**, **H2**, **H3**, **H4**, **H5**, and **H6**.
characters:	Any valid ISO Latin-1 character, character reference, or entity reference.
character highlighting:	**CITE**, **CODE**, **DFN** (proposed HTML2.0), **EM**, **KBD**, **SAMP**, **STRIKE** (proposed HTML2.0), **STRONG**, **VAR**, **B**, **I**, **TT**, and **U** (proposed HTML2.0).

These are the physical highlighting elements (italics, boldface, etc.) and also the logical highlighting elements that assign to text strings logical meanings that are generally interpreted as changes in text formatting.

The following sections are organized hierarchically. Thus, the first section describes the HTML element that contains the entire document. The next section describes the HEAD element, and this is followed by descriptions of the elements that can lie inside the HEAD. Following this is a description of the BODY element, followed by sections describing the elements that go inside the BODY.

HTML Element: The HTML Document

Usage: `<HTML> ... </HTML>`

Can Contain: HEAD, BODY

Can Be Inside: nothing

Attributes: none

The HTML element declares the enclosed text to be an HTML document. It may directly contain only two elements: HEAD and BODY. Although formally optional, you should always include an HTML element in new documents.

Example of HTML:

```
<HTML>
    <HEAD>
... head content....
    </HEAD>
    <BODY>
... body content....
    </BODY>
</HTML>
```

HEAD ELEMENTS

HEAD Element: Document Meta-information

Usage: `<HEAD> ... </HEAD>`

Can Contain: TITLE, ISINDEX, BASE, NEXTID, LINK, META

Can Be Inside: HTML

Attributes: none

HEAD contains general information about the document. The information found in the head is not displayed as part of the document text; consequently, only certain elements are appropriate within the HEAD. All the head elements are empty, except for TITLE, and they can appear in the HEAD in any order. The only mandatory HEAD element is TITLE; all others are optional. The possible HEAD elements are, in decreasing level of importance:

TITLE	The title of the document.
ISINDEX	Indicates the document is searchable.
BASE	A record of the original URL of the document.
NEXTID	Used by automated HTML editors to create unique document identifiers.
LINK	Defines a relationship between the document and another document.
META	Used for embedding, within the HEAD, information about the document that cannot be expressed in the preceding elements.

You are most likely to see and use the **TITLE**, **ISINDEX**, and **BASE** elements. The **NEXTID** element is designed for use by special HTML editing tools. The **LINK** element is not widely supported. The **META** element is new to HTML2.0 and is also not widely supported.

The division between the **HEAD** and the **BODY** is important, as there will soon be ways to access just the information in the **HEAD**. This will be much faster than accessing the entire document and extremely useful for generating catalogs or indexes.

BASE ELEMENT: BASE URL

Usage: `<BASE>`

Can Contain: empty

Can Be Inside: **HEAD**

Attributes **HREF**

BASE is an empty element and is optional. If present, **BASE** has a single, mandatory attribute, **HREF**, which is assigned the *base URL* of the document. The base URL indicates where a document was originally located. This is useful if a document is moved away from its original URL and related documents, in which case, relative URLs used to reference these neighboring documents are no longer valid. However, if the original URL address is specified in the **BASE** element, then relative URLs from this document are evaluated relative to this *base* URL and will be correctly found.

If the BASE element is absent, the browser assumes the base URL to be that used to access the document, and determines relative URLs with respect to this location. You should note that many browsers, such as **MacWeb** and **lynx**, do not support the **BASE** tag.

Example of BASE:

If a document was originally found at the URL:

```
http://somewhere.org/Dir/Subdir/file.html
```

the appropriate BASE element is then:

```
<HEAD>
.
<BASE HREF="http://somewhere.org/Dir/Subdir/file.html">
</HEAD>
```

ISINDEX Element: Searchable Document

Usage: <ISINDEX>

Can Contain: empty

Can Be Inside: HEAD [BODY, BLOCKQUOTE, FORM, LI, DD]

Attributes: none

ISINDEX is an empty element and is optional. Because there are many older documents that have the ISINDEX declaration in the BODY, the HTML definition still allows this form. New documents should place ISINDEX in the HEAD. This element informs the browser program that the document can be examined using a keyword search, and that the browser should query the user for a search or query string.

ISINDEX does *not* mean a search of the text you are reading. Documents containing ISINDEX elements are usually sent to the client from server-side gateway programs designed for database searches. You can think of such a document as a *front end* to a gateway program, and the *document* you search as the database *represented* by the document you see.

When you submit the search, the keywords are sent from the client to the server by appending them to the document URL. This mechanism is discussed in the URL portions of Chapters 3 and 4.

Example of **ISINDEX**:

```
<HEAD>
<ISINDEX>
   .
   .
</HEAD>
<BODY>
   .
body of document
   .
</BODY>
```

LINK Element: Relationship to Other Documents

Usage:	<LINK>
Can Contain:	empty
Can Be Inside:	HEAD
Attributes:	Same as A (anchor) element: HREF, METHODS, NAME, TITLE, URN, (REL, REV: proposed HTML2.0), (ID, EFFECT, PRINT, TYPE, SIZE, SHAPE: proposed HTML+)

The **LINK** element describes a relationship between the document and other documents or objects. For example, you might use **LINK** to indicate a related index, a glossary, or perhaps different versions of the same document. Alternatively, you could use it to point to likely *next* or *previous*

documents. This could be used by a browser to predict and preload documents it is likely to need (although at present there are no browsers that do this). A document may have any number of LINK elements to represent these various relationships to other documents.

LINK is an empty element, and is optional. LINK takes the same attributes as the anchor (A) element; these are discussed in detail in the anchor element section.

Example of LINK:

```
<HEAD>
<LINK HREF="file1.html" TITLE="Title of Related Document" >
</HEAD>
```

LINK is, as yet, rarely used and is largely unsupported.

META Element: Document Meta-information

Usage: <META>

Can Contain: empty

Can Be Inside: HEAD

Attributes: HTTP-EQUIV, NAME, CONTENT

The META element provides a place to put meta-information that is not defined by the other HEAD elements. This allows you to more richly describe the document content for indexing and cataloging purposes, as illustrated in the following. You should not, however, use this as a substitute for the other HEAD elements.

The META element is optional. If present, it must take the CONTENT attribute and one of the NAME or HTTP-EQUIV attributes (but not both). The meanings of the attributes are:

NAME="name"—This specifies the meta-information name. The client (browser or other program) must understand what this name means.

HTTP-EQUIV="string"—This can be used instead of the NAME attribute. META elements with HTTP-EQUIV attributes are parsed by the HTTP server and are converted into HTTP *response headers*.

CONTENT="string" (mandatory)—This assigns the *content* associated with the META element.

An example using the NAME attribute is:

```
<META NAME="DocumentBranch" CONTENT="Volume1_Branch_4X3">
```

This might tell the client that the document is a derivative in some defined way of Volume 1 of a master text. The client or indexing program that is accessing the HEAD of this document must consequently understand the meanings behind these names and their content.

The attribute HTTP-EQUIV allows the document to pass information to the server delivering the document. An example is:

```
<META HTTP-EQUIV="Last-Modified" CONTENT="23-Sep-94 18:28:33 GMT">
```

By this element, the document is asking the server to take the CONTENT (the string `"23-Sep-94 18:28:33 GMT"`) and include this information as part of the `Last-Modified` header field that is sent with the HTTP *response header* that precedes the document during an HTTP transaction. (The HTTP protocol and HTTP response headers are discussed in Chapter 4.) HTTP response headers contain information about the document and server, such as the type of data being sent from the server, the date it is being sent, the type of server being used, and so on. The response headers can also contain more descriptive information about the document, comparable to the information found in the document HEAD.

Most servers do not currently parse documents for META or other elements.

NEXTID Element: Counter for Automated Editors

Usage: <NEXTID>

Can Contain: empty

Can Be Inside: HEAD

Attributes: N

NEXTID is an empty element and is optional. It is used by HTML editing programs to uniquely identify documents created in the editing process—it is not designed for use by mere humans, by WWW browsers, or by hypertext servers.

NEXTID has a single mandatory attribute N that specifies a numeric identifier for the document. Since the numbers should uniquely identify documents, an HTML editor should never reuse old identity numbers.

Example of NEXTID:

```
<HEAD>
<NEXTID N=132>
.
.
</HEAD>
```

TITLE Element: Document Title

Usage: <TITLE> ... </TITLE>

Can Contain: characters

Can Be Inside: HEAD

Attributes: none

The title of a document is specified by the TITLE element. Every document must have a TITLE and can have only one. The TITLE should indicate the document content in a concise and general way. It serves several purposes:

- To label the display window or text screen
- To serve as a record in a history list marking documents you have viewed
- To allow quick indexing of a document, in place of indexing the entire text

The TITLE is not part of the document text and cannot contain hypertext links or any other markup commands—it can contain only text, including entity or character references.

The TITLE should be short so that it can easily label a window or fit in a history list—preferably less than 60 or so characters. You should also be able to determine the content of the document from the TITLE itself. Otherwise, a person reviewing their history will see the TITLE but not know to what it refers. Here are some examples:

Good TITLEs:

```
<TITLE>Paper on Rings by Baggins and Gandalf, 1989 </TITLE>

<TITLE>Introduction to MIME types </TITLE>
```

Bad TITLEs:

```
<TITLE>Introduction</TITLE>

<TITLE>A Summary of the Ring-Ring Interaction Cross-Section Measurement of B.
Baggins, et al. in both Low-Temperature and High-Temperature Studies, including
Water Immersion and Non-Destructive Testing: A Brief Review plus Commentary on
the "Missing Ring" Problem.</TITLE>
```

BODY ELEMENTS

BODY Element: The Displayed Text Body

Usage: `<BODY> ... </BODY>`

Can Contain: Hn, P, HR, DIR, DL, MENU, OL, UL, ADDRESS, BLOCKQUOTE, FORM, PRE, [ISINDEX]

Can Be Inside: HTML

Attributes: none

The BODY contains the document proper, as opposed to the information about the document found in the HEAD. Formally, the BODY cannot directly contain text. Instead, it must contain elements that themselves contain the text, because the BODY element states only "this is the body of the document" and supplies no additional meaning to its contents. It is the job of the other elements nested within the BODY to organize the text and assign it meaning. This is accomplished by the elements that define headings, lists, addresses, paragraphs, and so on.

The contents of the HEAD and BODY are exclusive—elements that belong inside the HEAD cannot go inside the BODY, and vice versa.

ADDRESS Element: Provide Address Information

Usage: `<ADDRESS> ... </ADDRESS>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: BLOCKQUOTE, BODY, FORM

Attributes: none

The ADDRESS element is used to mark out information, such as addresses, electronic signatures, lists of authors, and so on. Typically, a document author will use the ADDRESS to sign his or her documents. In this case, the ADDRESS is often placed at the bottom of the HTML document to keep it separate from the main text. In a family of documents, the ADDRESS may contain just the author's initials connected by a hypertext link to a biographical page. Alternatively, a collection of documents may have an introductory document that has ADDRESS elements containing detailed contact information for the author or authors, with the remaining documents having ADDRESS elements containing hypertext links back to this page.

The rendering of the contents of the ADDRESS is left up to the browser. For example, Mosaic usually renders the ADDRESS in italics. It may also be right-justified (as with Cello) or indented.

Figures 2.3 and 2.5 show some typical applications of the ADDRESS element. Browser renderings of these documents are shown in Figures 2.4 and 2.6 respectively.

BLOCKQUOTE Element: Block Quotations

Usage:	`<BLOCKQUOTE> ... </BLOCKQUOTE>`
Can Contain:	Hn, P, HR, DIR, DL, MENU, OL, UL, ADDRESS, BLOCKQUOTE, FORM, PRE, [ISINDEX]
Can Be Inside:	BLOCKQUOTE, BODY, DD, FORM, LI
Attributes:	none

The BLOCKQUOTE element is used to mark a block of text as a quotation. Browsers can render this in various ways; for example, by indenting and/or italicizing the BLOCKQUOTE contents, and by offsetting it from

the preceding and following text. A **BLOCKQUOTE** also causes a paragraph break.

```
<html>
<head>
<title> Examples of ADDRESS and BLOCKQUOTE elements</title>
</head>

<body>

<h1> Example 2: The Meaning of Life </h1>

<p> How many times have you sat down and asked yourself <quote>
What is the meaning of life? </quote>.  I certainly have.  I've
even read many of the good books, from C.S. Lewis, to Kant, to
Sartre to Zoltan the Magnificent.  But I think the most profound
statement about life was made by Jack Handley, who said:

<BLOCKQUOTE>
I can still recall old Mister Barnslow getting out every morning and
nailing a fresh load of tadpoles to that old board of his.  Then he'd
spin it around and around, like a wheel of fortune, and no matter where
it stopped he'd yell out, "Tadpoles!  Tadpoles is a winner!"
We all thought he was crazy.  But then, we had some growing up to do.
</BLOCKQUOTE>

That pretty well sums it up.

<HR>
<ADDRESS>  <A HREF="about_the_author.html"> C.S.O </A> </ADDRESS>

</body>
</html>
```

Figure 2.3 HTML example document illustrating headings, **BLOCKQUOTE**, and **ADDRESS** elements. Figure 2.4 shows this document viewed by **Cello**.

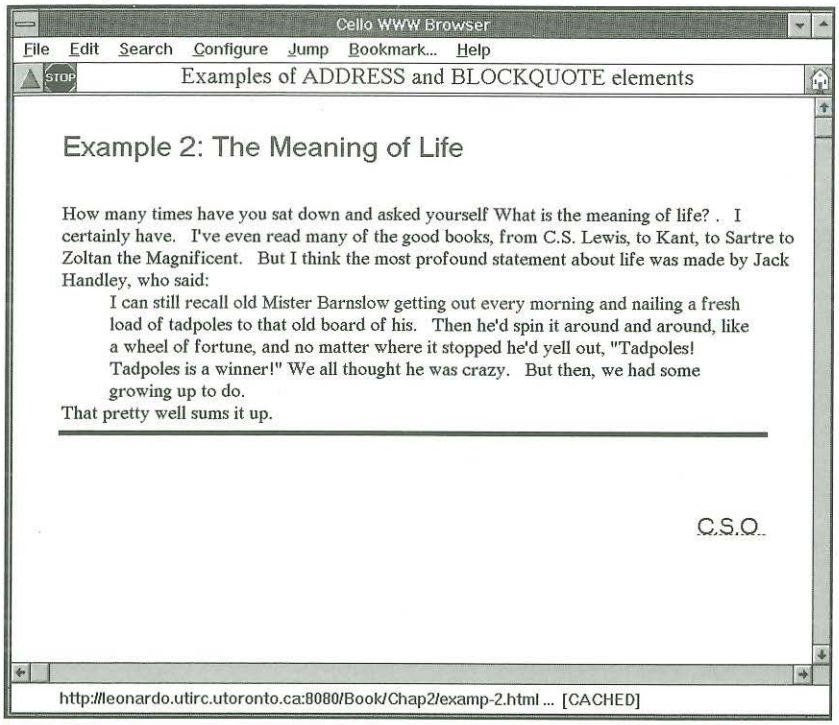


Figure 2.4 Display, by the **Cello** browser, of the document shown in Figure 2.3.

Note that the definition does not allow text directly inside a BLOCKQUOTE; instead, the text must lie inside other elements lying inside the BLOCKQUOTE. Thus, the form:

```
<BLOCKQUOTE>
<P> This is the quotation. ...
..
</BLOCKQUOTE>
```

is correct, while the form:

```
 <BLOCKQUOTE>
This is the quotation. ...
..
</BLOCKQUOTE>
```

is not. A typical BLOCKQUOTE is shown in Figures 2.3 and 2.4.

```
<html>
<head>
<title> Some examples of ADDRESS and heading elements </title>
</head>

<body>

<h1> Example 3: The Truth About Santa  </h1>
<p> Breaking the news to a small child that Santa Claus is
merely a tool of the modern capitalist is one of
the saddest moments in raising children.  Nevertheless, such
truths must be brought to life, for fear that your child
become another Pangloss lost in the idealism so prevalent
amongst our youth. Here are some different methods to
introduce this topic.

<h2> Santa's Exploitation of the Working Class </h2>
<p>  Begin by talking about Santa's enslaved workforce.  How
can those poor gnomes make all those gifts?  Clearly
by driven overwork.....

<h3> Elves and the Union Movement </h3>
<p> and so on.......

<h4> Elf Exploitation </h3>
<p> And still more text.

<hr>
<ADDRESS>
Santa Claus<br>
Christmas Holiday Specialist <br>
North Pole, CANADA H0H 0H0<br>
Tel (555) 555 POLE
</ADDRESS>

</body>
</html>
```

Figure 2.5 HTML example document illustrating **TITLE**, heading, and **ADDRESS** elements. Figure 2.6 shows this document viewed by **WinMosaic**.

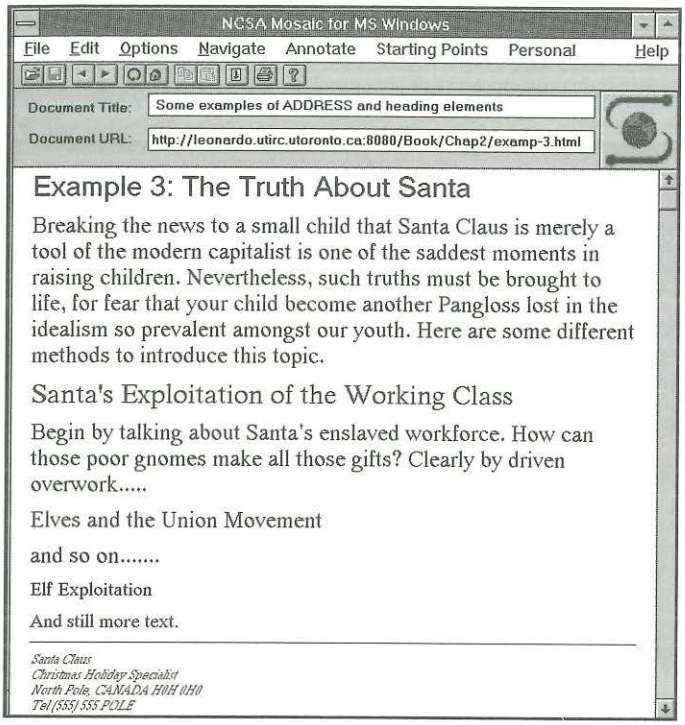


Figure 2.6 Display, by the **WinMosaic** browser, of the document shown in Figure 2.5.

FORM Element: Fill-In Forms

Usage:	`<FORM> ... </FORM>`
Can Contain:	INPUT, SELECT, TEXTAREA, Hn, P, HR, DIR, DL, MENU, OL, UL, ADDRESS, BLOCKQUOTE, PRE, [ISINDEX]
Can Be Inside:	BLOCKQUOTE, BODY, DD, LI
Attributes:	ACTION, ENCTYPE, METHOD

The **FORM** element marks out the content of an HTML *fill-in form*. This is the element you use to create fill-in forms with checkboxes, radio boxes, text input windows, and buttons. Data from a **FORM** must be sent to server-side gateway programs for processing; recall that **FORM** collects data, but does not process it. In general, a **FORM** and the server-side program handling the **FORM** output must be designed together so that the program understands the data being sent from the **FORM**. Some simple examples showing the variety of possible **FORM**s are shown in Figures 2.7 through 2.10.

The **FORM** element takes three attributes. These determine where the **FORM** input data is to be sent; what HTTP protocol to use when sending the data; and the data type of the content (as a MIME content-type). Note that **FORM**s do not nest—you *cannot* have a **FORM** within a **FORM**.

The attributes are:

ACTION= "*URL*" (mandatory)—The **ACTION** specifies the URL to which the **FORM** content is to be sent. Usually this is a URL pointing to a program on an HTTP server, since only HTTP servers allow significant interaction from the client to the server. However, the **ACTION** can specify other URLs. For example, in the case of a **mailto** URL, the **FORM** content would be mailed to the indicated address.

METHOD= "GET" or "POST" (optional)—When the **ACTION** indicates an **http** URL, the **METHOD** gives the HTTP *method* for sending information to the server. HTTP methods are discussed in Chapter 4. The default value for **ACTION** is GET. With GET, the content of the form is appended to the URL in a manner identical to query data from an **ISINDEX** search. With the POST method, the form content is sent to the server as a message, and not as part of the URL.

ENCTYPE= "*MIME Type*" (optional)—ENCTYPE specifies the MIME-type of data sent, using the POST method. The default value is `application/x-www-form-urlencoded`. At present, this is the only allowed value.

INPUT Element: Text Boxes, Checkboxes, and Radio Buttons

Usage:	<INPUT>
Can Contain:	empty
Can Be Inside:	FORM, any nonempty element allowed inside a FORM
Attributes:	ALIGN, CHECKED, NAME, MAXLENGTH, SIZE, SRC, TYPE, VALUE

The INPUT element specifies a variety of *editable fields* inside a form. It takes several attributes that define the type of input mechanism (text field, buttons, checkboxes, etc.), the *variable name* associated with the input data, and the alignment and size of the input element when displayed. Although the INPUT element can appear only inside a FORM element, BLOCKQUOTE, P, and list elements can also be inside a FORM and can be used to organize the INPUT elements into lists or paragraphs. Some examples of the INPUT element, and the organization allowed by these other elements, are shown in Figures 2.7 through 2.10.

The most important attribute to the INPUT element is NAME, which assigns a *variable name* to the *value* entered into the element. The data entered into a FORM are sent to the server as a collection of *name/value* pairs. The program parsing the FORM data uses the variable *name* to interpret the contents of the corresponding *value* and must therefore understand the different *names*. This is why a FORM and the gateway program that handles the FORM data must be designed together.

The other main attribute is TYPE, which selects the type of the INPUT element. There are several other attributes, but their usage and relevance depends on the TYPE.

The attributes are:

NAME = "*name*" (mandatory)—This assigns the variable name "*name*" to the data contents of this **INPUT** element.

TYPE = "checkbox", "hidden", "image", "radio", "reset", "submit", "text" (mandatory):

"*checkbox*"—*Checkbox* **INPUT** elements are Boolean (*on/off*) quantities; the default value is *off* (this can be modified with the **CHECKED** attribute). The **VALUE** attribute sets the *value* assigned to an *on* checkbox. When you submit a **FORM**, the *name/value* pair is sent only if the checkbox is *on*.

Different checkboxes may associate different *values* with the same variable *name*. This is convenient, for example, if you have six different databases to search and want to allow the user to select one, two, or all of them. When the **FORM** is submitted, the browser sends all the *values* from the *on* checkboxes yielding several *name/value* pairs with the same *name*. An example is shown in Figure 2.7.

"*hidden*"—This **INPUT** element is not displayed by the browser and is hence *hidden* from the user. The contents of a "*hidden*" element, set with the **VALUE** attribute, are always sent to the server when the **FORM** is submitted. This is useful for passing information back and forth between the client and server, and is typically used to record the *state* of the client-server interaction. Recall that the HTTP protocol is stateless so that, without such passed information, the gateway program handling the **FORM** data has no record of any past interaction. Typically, a "*hidden*" **INPUT** element is placed in a **FORM** by a server-side gateway program that assembles the **FORM**. This point is discussed in more detail in Chapter 4.

"*image*"—This **INPUT** element is an active inline image (analogous to the **ISMAP** attribute of the **IMG** element). The **SRC** attribute specifies the URL of the image.

Clicking on the image immediately submits the FORM data, including the coordinates of the mouse pointer (measured in pixels from the upper left-hand corner of the image). The coordinates are sent in two *name/value* pairs. The *name* is created by taking the NAME attribute and appending the strings *.x* and *.y* to indicate the *x* or *y* coordinate. Thus, if the NAME was set to "*king*", the coordinates are sent in the *name/value* pairs *king.x/x_coord* and *king.y/y_coord*.

"*password*"—This INPUT element is a single-line text field, but the text typed into the field is obscured by asterisks or by some other method. This is used for password entry. An example password field is shown in Figures 2.9 and 2.10.

"*radio*"—This INPUT element is a radio button. Radio buttons are linked together by assigning them the same NAME. By definition, radio buttons can select only one from among all possible values; therefore, when you turn *on* one radio button, any other button associated with the same NAME is turned *off*. Each radio button must have a value so that each INPUT element of TYPE="*radio*" must have a VALUE attribute.

"*reset*"—This INPUT element is a reset button. When pressed, all the fields in the FORM are reset to the values given by their VALUE attributes.

```
<html>
<head>
<title> Examples of HTML FORMS </title>
</head>

<body>
```

```
<h1> Example 4:  Examples of FORMS</h1>
<p> sends search information to a script.

<FORM  ACTION="http://side.edu/cgi-bin/script">

   <p> Search string: <INPUT TYPE="text" NAME="search_string" SIZE=24>
   <p> Search Type:
          <SELECT NAME="search_type">
              <OPTION> Insensitive Substring
              <OPTION SELECTED> Exact Match
              <OPTION> Sensitive Substring
              <OPTION> Regular Expression
          </SELECT>
   <p> Search databases in:
    [<INPUT TYPE="checkbox" NAME="servers" VALUE="Canada" CHECKED>Canada]
    [<INPUT TYPE="checkbox" NAME="servers" VALUE="Russia">Russia]
    [<INPUT TYPE="checkbox" NAME="servers" VALUE="Sweden">Sweden]
    [<INPUT TYPE="checkbox" NAME="servers" VALUE="U.S.A.">U.S.A.]
    <em>(multiple items can be selected.)</em>
   <p> Niceness:
       <menu>
        <li> <INPUT TYPE="radio" NAME="niceness" VALUE="nicest" CHECKED >Nicest
        <li> <INPUT TYPE="radio" NAME="niceness" VALUE="nice" >   Nice
        <li> <INPUT TYPE="radio" NAME="niceness" VALUE="not nice"> Not Nice
        <li> <INPUT TYPE="radio" NAME="niceness" VALUE="nasty" >  Nasty
      </menu>
   <P> <INPUT TYPE="submit"> <INPUT TYPE=reset>.

</FORM>

<HR>
<ADDRESS>  Form by <A HREF="about_the_author.html"> I.S.G</A> </ADDRESS>

</body>
</html>
```

Figure 2.7 HTML example document illustrating several FORM **INPUT** elements and the **SELECT** element. Figure 2.8 shows this document as displayed by the **Mosaic for X-Windows** browser.

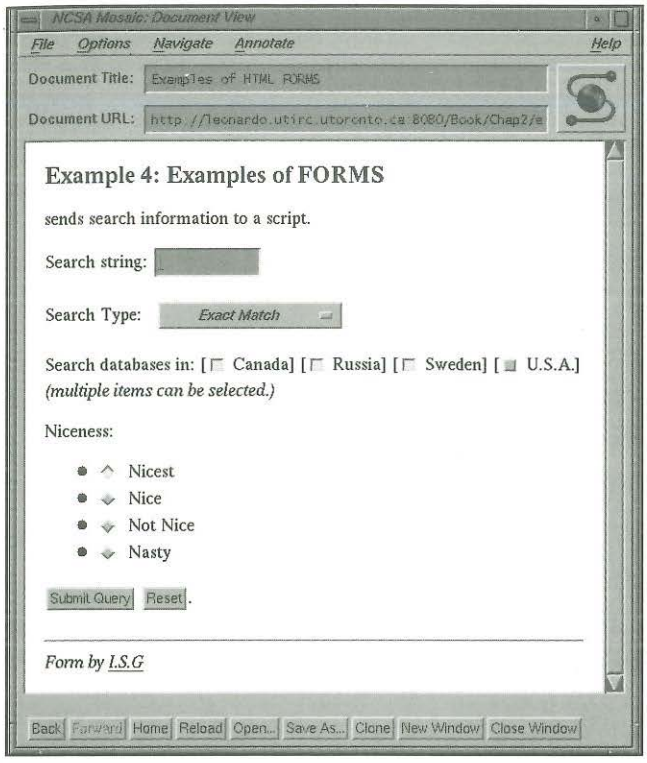


Figure 2.8 Display of the document shown in Figure 2.7 by the **Mosaic for X-Windows** browser.

"*submit*"—This **INPUT** element is a submit button. Pressing the submit button sends the **FORM** data to the specified URL. A form can have more than one such button, each with different **NAME** and **VALUE** attributes: The **FORM** sends only the *name/value* pair associated with the pressed submit button. The *value* is not editable by the client and is displayed as the button label.

Many browsers do not yet support multiple submit buttons with different *name/value* pairs.

"*text*"—This **INPUT** element is a single-line text entry field. The physically displayed size of the input field is set by the **SIZE** attribute.

ALIGN="*top*", "*middle*", "*bottom*" (optional)—ALIGN specifies the alignment of the image with respect to the surrounding text. The usage is equivalent to the ALIGN attribute to the IMG element (described later in this section). This is a valid attribute only with TYPE="*image*".

CHECKED (optional)—The CHECKED attribute indicates that a checkbox or radio button is selected (turned *on*). This attribute is valid only with TYPE="*checkbox*" or TYPE="*radio.*"

MAXLENGTH="*n*" (optional)—MAXLENGTH specifies the length of the character buffer for a text box, where "*n*" is the buffer length. This is a valid attribute only with TYPE="*text*" or TYPE="*password.*" MAXLENGTH can be larger than the displayed text box, in which case, the arrow keys may be used to scroll the text. The default length is unlimited.

SIZE="*n*" (optional)—SIZE specifies the actual size of the displayed field. When TYPE="*text*" or TYPE="*password*" SIZE specifies the width of the input text box in characters.

SRC="*URL*" (mandatory with TYPE="*image*")—SRC specifies the URL of the image to be included inline and is valid only with TYPE="*image*".

VALUE="*value*" (mandatory with TYPE="*radio*")—VALUE specifies the initial value of the input element.

Figures 2.7 through 2.10 give typical examples of INPUT element usage. FORMs are also discussed in Chapters 1 and 4.

SELECT Element: Select from among Multiple Options

Usage:	`<SELECT> ... </SELECT>`
Can Contain:	OPTION
Can be Inside:	FORM
Attributes:	MULTIPLE, NAME, SIZE

The SELECT element allows the user to select from amongst a set of values presented as a selectable list of text strings: The possible *values* are specified by the OPTION element. The attribute MULTIPLE allows multiple values to be selected; otherwise, only one value can be chosen. As with the INPUT elements, the selected data is sent to the server as one or more *name/value* pairs.

The attributes are:

MULTIPLE (optional)—If MULTIPLE is present, the SELECT element allows multiple items to be selected. If MULTIPLE is not present, you can select only a single item from the SELECT list.

NAME="*name*" (mandatory)—NAME specifies the variable name associated with the SELECT element.

SIZE="*n*" (optional)—SIZE specifies the number of displayed text lines. The default value is 1; consequently, the list is often presented as a pull-down menu. For other values, the list is usually presented as a scrollbox. If MULTIPLE is set, browsers choose a minimum SIZE greater than 1 and will not let you use SIZE to select a smaller value.

Figures 2.9 and 2.10 show typical examples of SELECT (and OPTION) elements.

OPTION ELEMENT: LIST OF OPTIONS FOR SELECT

Usage:	`<OPTION> ... (</OPTION>)`
Can Contain:	**characters**
Can Be Inside:	SELECT
Attributes:	VALUE, SELECTED, (DISABLED: proposed HTML2.0)

The OPTION element sets the different character-string options for a SELECT element. This element is not empty but the terminating </OPTION> is optional, as the element is by default terminated by the next <OPTION> tag or by the </SELECT> tag ending the list. OPTION can contain characters, character references, or entity references only; it cannot contain markup. The content of OPTION is used as the *value* unless a VALUE attribute is explicitly set.

The attributes are:

DISABLED (optional) (proposed HTML2.0)—DISABLED marks a particular OPTION as disabled—if displayed, it may be shown as grayed or faded.

SELECTED (optional)—This marks the OPTION as selected. If the SELECT element has the MULTIPLE attribute, more than one OPTION can be marked as selected. Figures 2.9 and 2.10 show examples.

VALUE= "*value*" (optional)—Specifies the *value* assigned to the OPTION. If absent, the content of OPTION is sent as the *value*.

TEXTAREA Element: Input a Block of Text

Usage:	<TEXTAREA> ... </TEXTAREA>
Can Contain:	characters
Can Be Inside:	FORM
Attributes:	COLS, NAME, ROWS

TEXTAREA allows the user to enter a block of text. The input block of text can grow to almost unlimited size, and is not limited by the size of the area displayed on the screen. Scrollbars are often present if the text area is bigger than the displayed region.

A **TEXTAREA** window displays characters in fixed-width fonts so that the attributes **COLS** and **ROWS** specify text area window sizes in character widths and heights. **TEXTAREA** can contain any printable characters.

The attributes are:

COLS="*n***"** (mandatory)—**COLS** specifies the width of the **TEXTAREA** in columns.

NAME="*name***"** (mandatory)—**NAME** specifies the variable name associated with the **TEXTAREA** contents.

ROWS="*n***"** (mandatory)—**ROWS** specifies the height of the **TEXTAREA** in rows.

Text placed within a **TEXTAREA** is displayed as an initial value. Note that HTML markup tags are not valid here. A browser provides some way to edit the displayed text. Figures 2.9 and 2.10 show a typical example of a **TEXTAREA** element.

```
<html>
<head>
<title> Examples of HTML FORMS </title>
</head>

<body>

<h1>Example 5: More Forms Examples</h1>
<p> This might send an abstract for registration to a selection
of databases.

<FORM  ACTION="http://side.edu/cgi-bin/submit_abstract">
   <h2> Please give name and password: </h2>
   <p> Name:      <INPUT TYPE="text"  NAME="userid" VALUE="guest" SIZE=20>
       Password: <INPUT TYPE="password" NAME="password" VALUE="bozo..." SIZE=8>
```

```
    <hr>
    <h2> Select Databases: </h2>
    <p> Physics: <SELECT NAME="physics_database" MULTIPLE SIZE=3>
                        <OPTION SELECTED> Condensed-Matter
                        <OPTION> High Energy
                        <OPTION> Solid-State
                        <OPTION> Quantum Cosmology
                        <OPTION> Astrophysics
                        </SELECT>
    Chemistry: <SELECT NAME="chemistry_database" MULTIPLE SIZE=3>
                        <OPTION> Surface Dynamics
                        <OPTION> Quantum Chemistry
                        <OPTION SELECTED> Polymer Dynamics
                        <OPTION> Biochemistry
                        <OPTION> Nuclear Chemistry
                        </SELECT>
    <h2> Enter Abstract: </h2>
    <p> <TEXTAREA COLS=60 ROWS=4> If you are submitting an abstract, select
the desired databases from the above list, delete this text,
type (or paste) the abstract into this box and press the
"Deposit Abstract" button.
        </TEXTAREA>
    <P> <STRONG> Press </STRONG> <INPUT TYPE="submit" VALUE="Deposit Abstract">
        to deposit, or <INPUT TYPE=reset> to reset form.

</FORM>

<HR>
<ADDRESS>  Form by <A HREF="about_the_author.html"> I.S.G</A> </ADDRESS>

</body>
</html>
```

Figure 2.9 HTML example document illustrating FORM **INPUT**, **SELECT**, and **TEXTBOX** input elements. Figure 2.10 shows this document as displayed by the **Mosaic for X-Windows** browser.

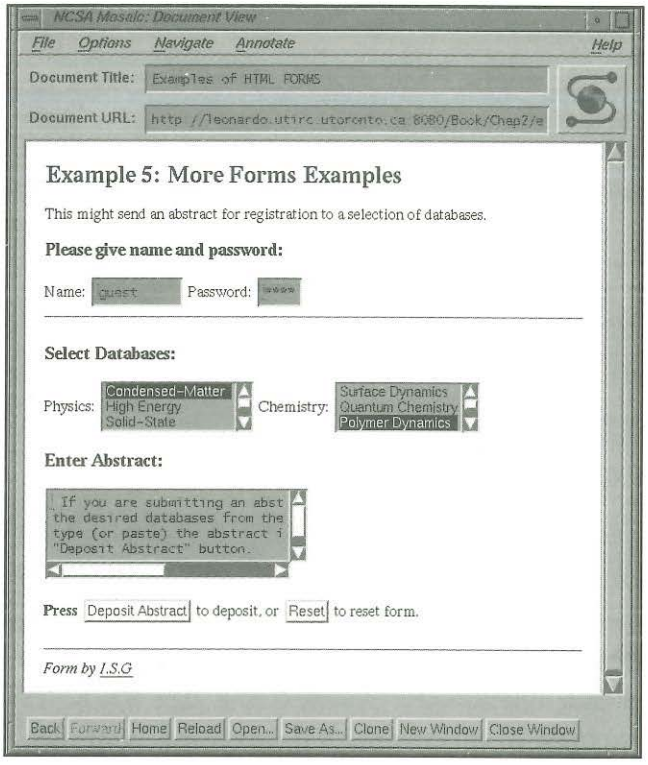


Figure 2.10 Display of the document shown in Figure 2.9 by the **Mosaic for X-Windows** browser.

HN ELEMENTS: HEADINGS

Usage:	`<Hn> ... </Hn>`
Can Contain:	**characters, character highlighting, A, BR, IMG**
Can Be Inside:	BLOCKQUOTE, BODY, FORM
Attributes:	(ID: proposed HTML+)

HTML allows six levels of headings from **H1** through **H6**. There is no forced hierarchy in these headings, but for consistency you should use the

top level (**H1**) for main headings, and lower levels for progressively less important ones. You should also avoid skipping a heading level within a given document, as this breaks the logical structure of the document and may cause problems when converting the document into another form, or if you want to automatically generate HTML Table of Contents documents. **Hn** elements have no attributes under the HTML2.0.

Current renderings of headings are very much browser-dependent. For example, **MacMosaic** renders **H1** headings with a large font and left-justified, while **lynx** renders **H1** headings as capitalized strings centered on the page. Some typical renderings are shown in Figures 2.2 through 2.24.

As a general rule, hypertext documents should be broken up so that each page does not occupy more than one or two browser screen areas. You can then use the **H1** heading to mark the main heading for the collection of documents, and the others to mark subheadings.

HTML+ proposes an **ID** attribute for the heading elements, which allows the heading to be the *named* destination of a hypertext link. The **ID** attribute is similar to the **NAME** attribute of the **A** (anchor) element. The **ID** attribute is optional.

Example:

```
<H1 ID="shops"> Surf Boarding Shops " </H1>
```

This labels the heading as the possible target of a hypertext link and is equivalent to:

```
<H1> <A NAME="shops"> Surf Boarding Shops " </A> </H1>
```

Some browsers, such as **emacs-w3**, support the **ID** attribute in headings but most do not.

HTML+ also proposes the **ALIGN** attribute, which suggests an alignment of the heading on the page. **ALIGN** can take three values: **ALIGN="left"**, which left-justifies the heading, **ALIGN="right"**, which right-justifies the

heading, and ALIGN="center", which centers the heading. This attribute is currently supported by the emacs-w3 and Netscape browsers.

HR ELEMENT: HORIZONTAL RULE

Usage: <HR>

Can Contain: empty

Can Be Inside: BODY, BLOCKQUOTE, FORM

Attributes: none

The HR element draws a horizontal line completely across the screen. An <HR> terminates any preceding paragraph, so a new paragraph mark should follow an <HR> if there is subsequent text. An <HR> is commonly used to divide sections within a single document. One common example is to use an <HR> at the bottom of a document, followed by an ADDRESS element. This is illustrated in Figures 2.4 through 2.10.

The HR element is empty.

LIST ELEMENTS

HTML has several elements for defining different types of lists. They can be divided into two types: glossary lists (the element DL) and regular lists (the elements DIR, MENU, OL, and UL). Lists of the same or different types can be nested. Thus, you can have a regular list within a regular list, a regular list within a glossary list, and so on. Some examples are shown in the following.

DL ELEMENT: GLOSSARY LISTS

Usage: <DL> . . . </DL>

Can Contain: DT, DD

Can Be Inside: BLOCKQUOTE, BODY, DD, FORM, LI

Attributes: COMPACT

This list type, also known as a definition list, presents a list of items, each with a descriptive paragraph. This can be used, for example, for traditional glossaries.

DL takes a single optional attribute, COMPACT, to signify that the list should be rendered in a physically compact way. You could use this to compact a list of small items, or to compact a large list that would be easier to read if rendered in a compact manner.

A DL list can contain two elements:

DT The term being defined

DD The definition of the term

DT and DD elements should appear in pairs. The specification allows for successive DT elements without a matching DD, but this use is discouraged. Figures 2.11 and 2.12 show an example of a DL list.

DT Element: Term in a Glossary List

Usage: `<DT> ... (</DT>)`

Can Contain: characters, character formatting, A, BR, IMG

Can Be Inside: DL

Attributes: none

The DT element contains the term part of a glossary or description list entry. The contents of a DT element should be short; typically, a few words and certainly shorter than a line. The element can contain standard character markup, images, line breaks, and hypertext anchors.

The DT element is not empty, but the terminating </DT> is optional, as it is implied either by the start of another <DT> or <DD> element or by the </DL> ending the list.

DD Element: Description in a Glossary List

Usage:	<DD> ... (</DD>)
Can Contain:	characters, character formatting, A, BR, IMG, DIR, DL, MENU, OL, UL, P, BLOCKQUOTE, FORM, P, PRE, [ISINDEX]
Can Be Inside:	DL
Attributes:	none

The DD element gives the description corresponding to the previous DT element. It can be a long description, broken into paragraphs and containing other lists, FORMs, quotations, and so on. A DD element must always follow a DT element; it cannot occur alone.

The DD element is not empty, but the terminating </DD> is optional, since the end of a DD element is implied either by the <DT> tag starting another DT element or by the </DL> tag terminating the list.

```
<html>

<head>
<title> Example of Glossary List elements </title>
</head>

<body>
```

```
<h1> Example 6: Example of Glossary Lists </h1>

<p> Here is an example of a glossary list. The third item in the list
has a regular unordered list nested within it. Note that the first
term (marked by the DT element) does not have a matching description
(marked by the DD element).  This is perfectly legal.  The converse:
a DD without a matching DT, is illegal.

<dl>
<dt> Things to do:
<dt> Things to Avoid:
    <dd> You should not use elements that define paragraph
    formatting within the PRE element. This means you should
    not use <code> &lt;P>, &lt;ADDRESS>, &lt;Hn> </code> and so on.
    You should avoid the use of tab characters -- use single blank
    characters to space text apart.
<dt> Things That are OK:
    <dd>You <em> can </em> use the anchor element A.  A typed carriage
    return will cause a new line in the presented text.
    People you should never let format lists include:
    <ul>
        <li> Bozo the Clown
        <li> Uncle Fester
        <li> Knights who go nii
    </ul>
    as they generally do a poor job.

</dl>

</body>
</html>
```

Figure 2.11 HTML example document illustrating the **DL** glossary list elements, and a **UL** unordered list nested inside a glossary list. Figure 2.12 shows this document as displayed by the **Cello** browser.

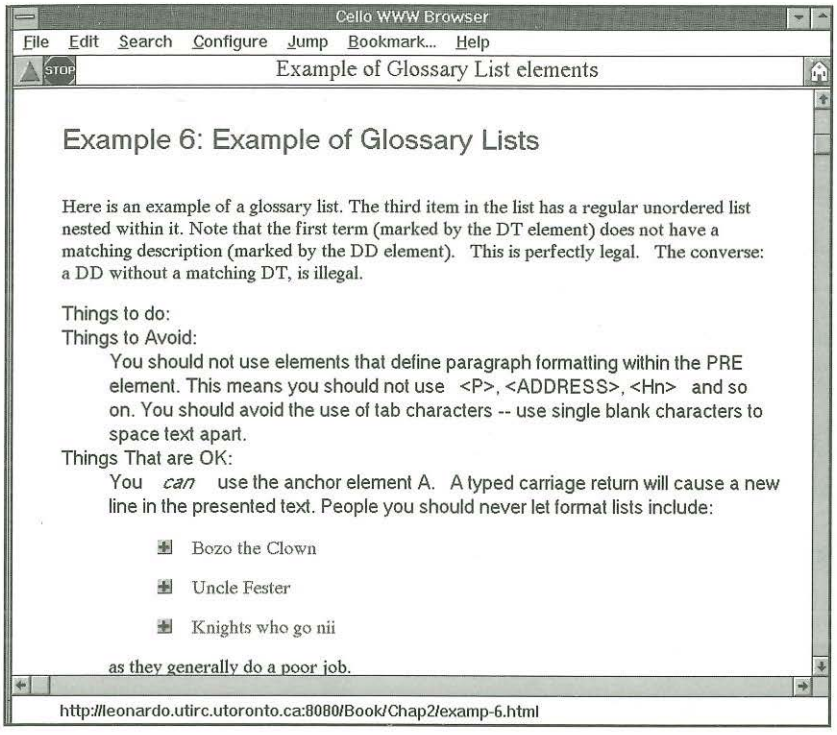


Figure 2.12 Display, by the **Cello** browser, of the document shown in Figure 2.11.

OL Element: Ordered List

Usage: `<OL> ... </OL>`

Can Contain: LI

Can Be Inside: BLOCKQUOTE, BODY, DD, FORM, LI

Attributes: (COMPACT: proposed HTML+)

The OL element defines an ordered list. A browser indicates ordering by numbering the items, assigning them ascending letters, and so on. The OL element has no attributes under the HTML2.0 specifications.

Each item in an OL list is contained within an LI (list item) element; the LI element is the *only* thing that can appear inside an OL list. Items can be paragraphs of text, but should be kept reasonably short; otherwise, the idea of a list is lost. If the list items are big, perhaps it is not really a list: Try paragraphs with appropriate section headings.

The HTML+ specifications propose adding the optional attribute COMPACT to the OL element. This would direct the browser to compact the list by, for example, reducing white spaces between list entries. This attribute is not understood (and is ignored) by most browsers.

A typical ordered list is show in Figures 2.13 and 2.14.

UL ELEMENT: UNORDERED LIST

Usage: `<UL> ... </OL>`

Can Contain: LI

Can Be Inside: BLOCKQUOTE, BODY, DD, FORM, LI

Attributes: (COMPACT, PLAIN, WRAP: proposed HTML+)

The UL element defines an unordered list of items, where each list item is indicated by a special symbol, such as a bullet or an asterisk. The UL element has no attributes under the HTML2.0 specifications.

Each item in a UL list is contained within an LI (list item) element; the LI element is the *only* thing that can appear inside a UL list. Items can be paragraphs of text, but should be kept reasonably short; otherwise, the idea of a list is lost. If the list items are big, perhaps it is not really a list: Try paragraphs with appropriate section headings.

HTML+ proposes three optional attributes to the UL element:

COMPACT: (optional) (proposed HTML+)—Compact the list, for example, by reducing white spaces between list entries.

PLAIN: (optional) (proposed HTML+) —Leave the list items unmarked (i.e., leave out the bullets).

WRAP= "*horiz*", "*vert*" (optional) (proposed HTML+)—Used for multicolumn lists, and declares how the items should be wrapped on the screen.

These attributes are currently not understood by most browsers.

Figures 2.13 to 2.16 show examples of unordered lists.

DIR Element: Directory List

Usage: `<DIR> ... </DIR>`

Can Contain: LI

Can Be Inside: BLOCKQUOTE, BODY, DD, FORM, LI

Attributes: (COMPACT: proposed HTML+)

The DIR element defines a directory list: a list of short items, each no more than about 20 characters. If possible, a browser can display the items in columns across the screen, as opposed to one above the other. In HTML2.0 the DIR element has no attributes.

Each item in a DIR list is contained within an LI (list item) element, where the LI element is the *only* thing that can appear inside a DIR list.

```
<html>

<head>
<title> Example of Regular List elements </title>
</head>
```

```
<body>
<h1> Example 7: Examples of Regular Lists </h1>

<h2> Ordered Lists </h2>
<p> This shows an ordered list, with another ordered list nested
within it.
<OL>
  <LI> First item.  Note that there can be lots of stuff in a list
  item, including images, paragraph breaks, BLOCKQUOTEs, and even
  other lists.
  <LI> A Second item in the list.
  <LI> And a third item.  This item breaks down into some
  subcategories:
     <OL>
     <LI> The first sub-item
     <LI> The second sub-item, and so on.....
     </OL>
</OL>
<hr>
<h2> Unordered Lists </h2>
<p> This also contains an ordered sublist.
<UL>
   <LI> A list item.
   <LI> Another list item; again these can containg IMG elements,
  Paragraphs, and so on.
   <LI> And sub-lists like this:
   <UL>
      <LI> An item in the list.
      <LI> Something else that is important, and so on.
   </UL>
</UL>


</body>
</html>
```

Figure 2.13 HTML example document illustrating **UL** and **OL** lists and the nesting of list elements. Figure 2.14 shows this document as displayed by the **WinWeb** browser.

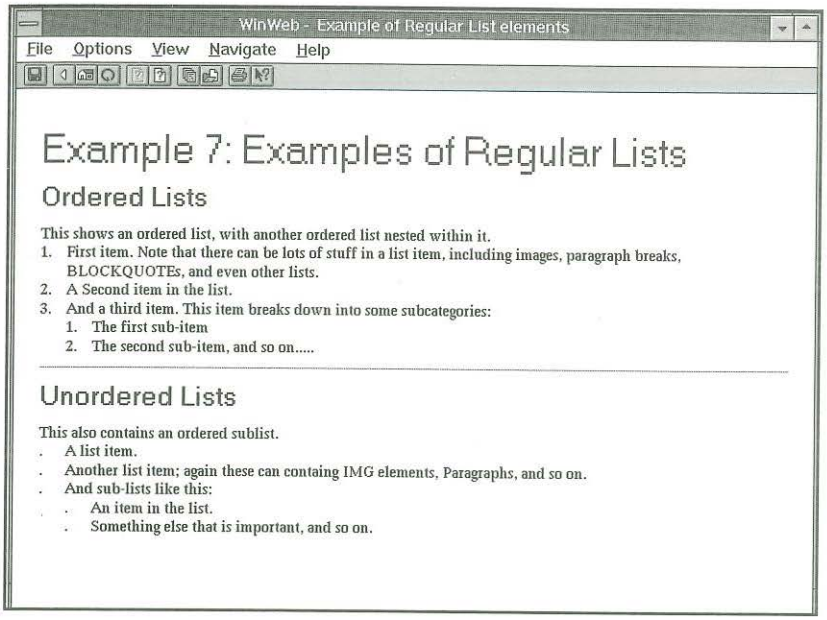


Figure 2.14 Display, by the **WinWeb** browser, of the document shown in Figure 2.13.

HTML+ proposes the optional attribute COMPACT. This would direct the browser to compact the list by, for example, reducing white spaces between list entries. This attribute is not understood by most browsers.

An example of a directory list is shown in Figures 2.15 and 2.16.

MENU Element: Menu List

Usage: `<MENU> ... </MENU>`

Can Contain: LI

Can Be Inside: BLOCKQUOTE, BODY, DD, FORM, LI

Attributes: none

```
<html>

<head>
<title> Example of Regular List elements </title>
</head>

<body>

<h1> Example 8: Still More Lists </h1>

<h2> Unordered Lists </h2>
<p> This also contains an ordered sublist.
<UL>
    <LI> A list item.
    <LI> Another list item,
    <LI> and still more items.
</UL>
<hr>
<h2> Directory Lists </h2>
<DIR>
    <LI> Abraham - Carbon
    <LI> Cardshark - Elegant
    <LI> Elegiac - Food
    <LI> Foot - Hogs
</DIR>
<hr>
<h2> Menu Lists </h2>
<MENU>
    <LI> First item
    <LI> Second item
    <LI> Third item
    <LI> Fourth item
</MENU>


</body>
</html>
```

Figure 2.15 HTML example document illustrating the **MENU** and **DIR** lists. Figure 2.16 shows this document as displayed by the **MidasWWW** browser.

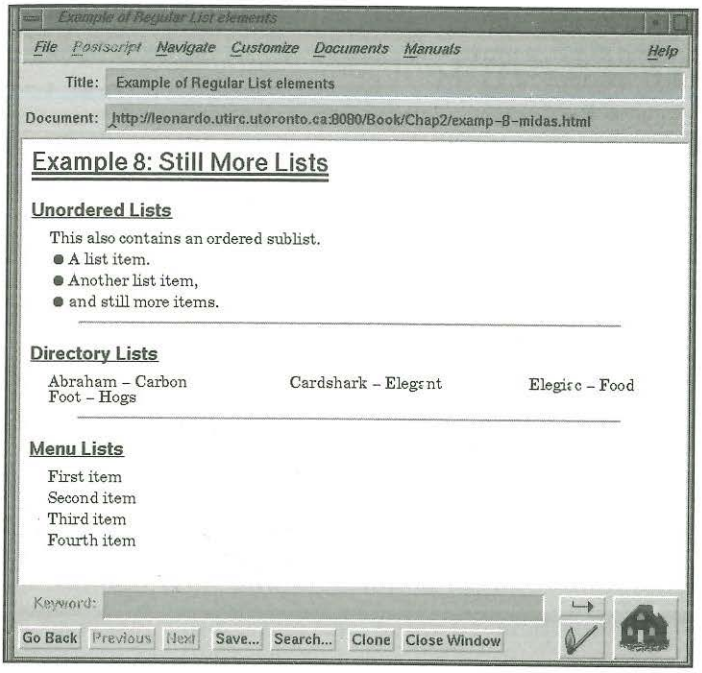


Figure 2.16 Display of the document shown in Figure 2.15 by the **MidasWWW** browser.

The MENU element defines a list of short menu items, each preferably less than a sentence long. The MENU list is designed to work like the UL but to be formatted in a more compact manner similar to a `<UL COMPACT>` list, except that formatting may be optimized to favor short list items.

MENU can contain only the LI elements. It is an error to place text directly inside a MENU. Figures 2.15 and 2.16 give an example of a MENU list.

LI Element: List Item

Usage: `<LI> ... (</LI>)`

Can Contain: characters, character highlighting, A, BR, IMG, DIR, DL, MENU, OL, UL, BLOCKQUOTE, FORM, P, PRE, [ISINDEX]

Can Be Inside: **DIR, MENU, UL, OL**

Attributes: (**SRC**: proposed HTML+)

The **LI** element marks an item within a list. The item can contain text, character markup, and hypertext anchors, as well as subsidiary lists and text blocks. In HTML2.0, the **LI** element has no attributes.

HTML+ proposes the attribute **SRC**, for use only when the **LI** element is within list elements **UL**, **MENU**, or **DIR**. The use of the tag `<LI SRC="image.gif">` would allow the browser to replace the default marker for each list item by the defined image file, as specified by a URL. **SRC** is optional and, in its absence, the browser would use the default marker. Most current browsers ignore the **SRC** attribute.

P Element: Paragraphs

Usage: `<P> ... (</P>)`

Can Contain: **characters, character highlighting,A, BR, IMG**

Can Be Inside: **BODY, BLOCKQUOTE, FORM, DD, LI**

Attributes: (**ID, ALIGN, WRAP**: proposed HTML+)

The **P** element marks the beginning of a paragraph and implies a paragraph break. This is different from the **BR** element that represents a simple line break. Paragraphs should be thought of as logical blocks of text, similar to a **BLOCKQUOTE, ADDRESS**, or heading **Hn**, while a **BR** is simply a *character* that causes a line break. In HTML2.0, the **P** element has no attributes.

Typically, a paragraph is rendered with extra space between the previous and following blocks of text. Sometimes, the first line is also indented. For historical reasons, an end tag </P> is not required. Instead, the end of a paragraph is implied by the beginning of another paragraph or by another element marking a block of text. An end tag </P> can be used to mark the end of a paragraph, but its use is optional. Examples of the use of paragraph tags are given in Figures 2.1 through 2.24.

You should not use paragraphs to add spacing, for example, by writing:

```
....text
<P>
<P>
<P>
<H2> And another thing of Interest </H2>
```

Formally, a paragraph cannot be empty, so the preceding is illegal. Most browsers will tolerate this, but their interpretations will vary: Some leave extra spaces and some ignore the extra <P> tags completely.

HTML+ suggests the following attributes:

ID (optional) (proposed HTML+)—This marks the paragraph for a possible reference via a URL. (NOTE: In HTML+, **ID** is designed to replace the **NAME** attribute currently used to mark hypertext link targets.) This option works with some current browsers; notably, **emacs-w3**. For example:

```
<P ID="paragraph_1">
```

ALIGN="*center*", "*left*", "*right*", "*justify*", "*indent*" (optional) (proposed HTML+)—Change the paragraph alignment (the default is left) or paragraph justification. This option is implemented on a few browsers; notably, **emacs-w3** and **Netscape**. **ALIGN** is a formatting hint that can be ignored by the browser.

WRAP="*on*", "*off*" (optional) (proposed HTML+)—**WRAP** disables or enables wordwrapping. **WRAP="*on*"** enables word wrapping (the default) while **WRAP="*off*"** disables word wrapping. **WRAP** is only supported by **emacs-w3**.

PRE ELEMENT: PREFORMATTED TEXT

Usage:	<PRE> ... </PRE>
Can Contain:	**characters, character highlighting, A**
Can Be Inside:	**BODY, BLOCKQUOTE, FORM, DD, LI**
Attributes:	**WIDTH**

The **PRE** element marks text to be displayed with a fixed-width typewriter font. In particular, the **PRE** environment preserves the line breaks and space characters in the original text. This is the only element in HTML that does so. **PRE** is therefore useful for presenting text that has been formatted for a fixed-width character display, such as a plain text terminal.

With HTML2.0, the **PRE** element is the only mechanism for displaying tabular data or any other information that requires well-defined columns or relative positions of the text. All the other elements format text with variable-width fonts and adjust white spaces between words.

PRE takes the single optional attribute: **WIDTH**. This specifies the maximum number of characters that can be displayed on a single line, and tells a browser that it can wrap the line at this point. A default value of 80 is often assumed, depending on the browser. Graphical browsers often ignore the **WIDTH** attribute completely and let the lines in a **PRE** element be as long as they wish. You can then use a scroll bar to see text that runs off display.

TIPS

THINGS TO AVOID:

You cannot use elements that define paragraph formatting within the PRE element. This means you should not use <P>, <ADDRESS>, <Hn>, and so on. You should avoid tab characters, since different browsers interpret the size of a tab differently. Instead, use space characters to vertically align text.

TIPS

USEFUL FEATURES:

You can use the A (anchor) to create hypertext anchors inside PRE and all the character highlighting elements (STRONG, EM, and so on). These highlighting elements may be ignored by the browser if appropriate rendering is not possible.

An example of the PRE element is shown in Figures 2.17 and 2.18. Note the use of character highlighting. Character highlighting inside a PRE element contributes zero character width.

```
<html>
<head>
<title> Example of the PRE elements </title>
</head>

<body>
```

```
<h1> Example 9: The Use of PRE </h1>
<p> The PRE element is often used to include blocks
of plain text.  For example you can use it to include
examples of typed code:
<hr>
<PRE>
/* main program for fitting program */

extern int *sharv;
static char boggle[100];

main (int argc, char *argv)
double x_transpose, y_transpose, f_ack=2.3;
{
   .
</PRE>
<hr>
<p> PRE is also used for <em> tables</em>:  HTML2.0 does not have
elements for defining tables:
<PRE>
   Item           Price      Tax     Total          Category

   fileserver     10000      300     10300            <a href="cat_a.html">A</a>
   disk drive       900       30       930            <a href="cat_b.html">B</a>
  <strong>transmission</strong>     4400      110      4510             C
  <em>fertilizer</em>       5500      100      5600              F
</PRE>
The markup codes take up no space: if you delete everything between the
angle brackets (in the raw HTML) all the columns align perfectly.


</body>
</html>
```

Figure 2.17 HTML example document illustrating the use of the **PRE** element. Figure 2.18 shows this document as displayed by the **Mosaic for X-Windows** browser.

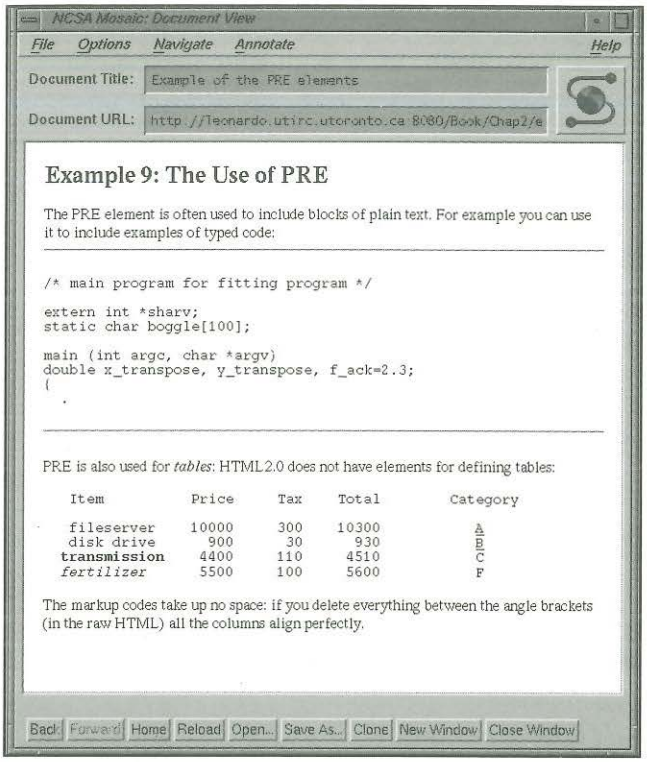


Figure 2.18 Display, by the **Mosaic for X-Windows** browser, of the document shown in Figure 2.17.

A Element: Hypertext Anchors

Usage:	`<A> ... </A>`
Can Contain:	characters, character highlighting, BR, IMG
Can Be Inside:	ADDRESS, Hn, P, PRE, DT, DD, LI, character highlighting
Attributes:	HREF, METHODS, NAME, TITLE, URN, (REL, REV: proposed HTML2.0), (ID, EFFECT, PRINT, TYPE, SIZE, SHAPE: proposed HTML+)

The **A**, or *anchor*, element marks a block of the document as the beginning and/or end of a hypertext link. This block can be highlighted text or an image. More complex elements, such as headings, cannot be inside an anchor.

A can take several attributes, where at least one of these *must* be either **HREF** or **NAME**; these specify the destination of the hypertext link or indicate that the marked text can be the target of a hypertext link. Both can be present, indicating that the anchor is both the start and destination of a link.

Anchors containing **HREF**s are rendered differently than plain text: Often, the anchored text or image is underlined, bold-faced, or rendered with a different color. Some browsers change this rendering once a link has been accessed, to inform the user that the link has been *explored*. Anchors with only a **NAME** attribute are usually not rendered in a special way. Several examples of anchor elements are given in Figures 2.2, 2.18, and 2.20.

The element attributes are:

HREF="*URL***"** (mandatory if **NAME** is absent)—**HREF** gives the target of a hypertext link. URL is the Uniform Resource Locator referencing the target object.

NAME="*string***"** (mandatory if **HREF** is absent)—**NAME** is used to mark a place in an HTML document as a specific destination of a hypertext link. The value "*string*" identifies this destination. For example, the anchor element:

```
<A NAME="poison"> Deadly Toadstools </A>
```

marks this string as a possible hypertext target, referenced by the string "poison". This string is called a *fragment identifier*. Inside this document, the location can be referenced by the hypertext reference:

```
<A HREF="#poison"> Poisonous non-mushrooms </A>
```

Clicking on the phrase "Poisonous non-mushroom" will then link you back to the place in the current document marked by the `<A NAME="poison"> ... </A>` anchor.

From another document, you would write:

```
<A HREF="http://www.site.edu/slimy/toads.html#poison"> Poisonous </A>
```

where

```
http://www.site.edu/slimy/toads.html
```

is the URL for this document.

Clicking on the word "Poisonous" will then link you to the document ***toads.html*** and to the particular location indicated by the `<A NAME="poison"> ... </A>` anchor.

REL="*string***"** (optional) (proposed HTML2.0)—The **REL** attribute gives the relationship(s) described by the hypertext link; consequently, **REL** cannot be used unless an **HREF** is present. The relationship is defined between the two entire documents and is not just related to the particular link. As an example, **REL** could indicate that the linked document is an index for the current one, or that a document is an annotation to the current one (which a browser might want to display as a pop-up), and so on. It is a pity that the **REL** (and **REV**, which is the converse of **REL**) attribute is so little used, as it can impart significant meaning and organization to large sets of related documents.

The value for **REL** is a comma-separated list of case-insensitive relationship values. An example is:

```
<A HREF="http://foo.edu/fe.html" REL="Useindex">sdfsddf</a>
```

This would mean that the document ***fe.html,*** at the given URL, is an index document related to the current document. Values for the relationships and their semantics will be registered by the HTML registration authority. Here are some other examples:

Example:

```
<A HREF="http://foo.edu/note1.html" REL="annotation">related notes </a>
```

The information in the document ***note1.html*** is additional and subsidiary to the current document. A browser might display this as margin notes.

Example:

```
<A HREF="http://foo.edu/vers2.html" REL="supersedes">previously </a>
```

The document ***vers2.html*** is an earlier version of the document containing this link.

At present, the **REL** and **REV** attributes are rarely used, and most browsers do not understand them. They will be of growing importance as HTML documents and document development environments become more sophisticated. For additional information on **REL** and **REV** you should consult the HTML 2.0 draft specification, referenced at the end of this chapter.

REV="*string***"** (optional) (proposed HTML2.0)—**REV** is like **REL**, but with the relationship reversed. For example,

```
<A HREF="http://foo.edu/vers2.html" REV="supersedes"> later </a>
```

means that the document ***vers2.html*** is a later version of the document containing this link. Most browsers do not understand **REV** or **REL**.

URN="*URN***"** (optional)—**URN** is designed to specify *Universal Resource Numbers* for the linked document. URNs are designed to be universal document references that do not depend on server specification, as do URLs. Unfortunately, the specification for URNs is still under discussion, so this attribute is not functional.

TITLE="*string***"** (optional)—The **TITLE** attribute gives the **TITLE** of the linked document, and can be used by the browser to *preview* the title before

contacting the server for the actual document. Note that you cannot guarantee that the **TITLE** is correct until you actually access the linked document. Alternatively, **TITLE** can be used to give a title to a document that would otherwise not have a title, such as a plain text file, an image file, or a directory accessed via FTP or a Gopher menu. Most browsers do not understand the **TITLE** attribute.

METHOD="*method***"** (optional)—The **METHOD** attribute specifies a comma-separated list of HTTP *methods* supported by the target of the hypertext link. HTTP *methods* specify the way a resource is accessed and are discussed in Chapter 4.

The **METHOD** attribute is only a hint to the browser, as the actual methods that can be used can only be determined after communicating with the HTTP server. The intent is to give the browser a hint of what the link will do so that it can, for example, render the link anchor differently (perhaps using a different font, or apply a special icon) to represent the different methods. Most browsers do not understand the **METHOD** attribute.

The following anchor attributes are part of the HTML+ proposals. Most browsers do not understand these attributes.

ID="*string***"** (optional) (Proposed HTML+)—HTML+ proposes the attribute **ID** in place of **NAME**, to specify identifier strings for hypertext anchors.

EFFECT="*replace***", "***new***", "***overlay***"** (optional) (Proposed HTML+)—**EFFECT** tells the browser how to display the linked document upon following the link. "*Replace*" (the default) means replace the current document with the linked one. "*New*" means open a new window (cloned if possible) and display the document there. "*Overlay*" means that the linked document should be displayed in a pop-up window.

PRINT="*reference***", "***footnote***", "***sidebar***", "***section***", "***silent***"** (optional) (Proposed HTML+)—**PRINT** is used in generating printed versions of HTML documents, and instructs the document generator as to how to represent the

hypertext links. "*Reference*" (the default) means to print the URL of the document as a footnoted reference, while "*silent*" means that the reference should be ignored completely. The other options imply including the link document and printing it as a sidebar, footnote, or an inserted section.

TYPE="*MIME_type***"** (optional) (Proposed HTML+)—**TYPE** specifies the MIME type of the linked document. This information can be used by the browser to change the link representation (for example, to display an image icon if the link is to an image file).

CHARACTER-RELATED ELEMENTS

There are two character-like elements: **IMG**, which includes an image within a document, and **BR**, which forces a line break in the text. These are both empty elements. In HTML these are treated much like characters and can be inserted almost anywhere that characters are found.

IMG Element: Inline Images

Usage:	`<IMG>`
Can Contain:	empty
Can Be Inside:	**character highlighting**, A, Hn, P, ADDRESS, DD, DT, LI
Attributes:	ALIGN, ALT, ISMAP, SRC

The **IMG** element includes an image file inline with the text of an HTML document. You cannot use **IMG** to include other HTML text within an HTML document.

There are currently only three image formats that can be included inline by all graphical browsers. These are: GIF format (with the filename suffix *.gif*); X-Bitmaps (with the filename suffix *.xbm*); and X-Pixelmaps (with the

filename suffix *.xpm*). To make your documents universally viewable, convert all inlined images into one of these three formats. Note also that the **ISMAP** active image mapping works with GIF images only.

Images within a document are treated like *honorary* words or characters. Almost anywhere you have regular text, you can also have an image. The **PRE** environment is the only place where this is not true: **IMG** elements are not allowed inside a **PRE**. Images can be included within a hypertext anchor; thus, you can make an image icon behave as a button that links other documents.

The **IMG** element can take four attributes: **SRC** is mandatory and specifies the URL of the image file to be included. **ALIGN** specifies the alignment of the image with the surrounding text, while **ALT** gives an alternative text string for browsers that cannot display images. Finally, **ISMAP** indicates that the image is an active image. Examples of the **IMG** element are shown in Figures 2.19 and 2.20.

The attributes are:

SRC="*URL*" (mandatory)—**SRC** gives the URL of the image file.

ALIGN="*bottom*", "*middle*", "*top*" (optional)—**ALIGN** specifies the alignment of the image with the neighboring text. "*Bottom*" aligns the bottom of the image with the bottom of text and is the default. "*Middle*" aligns the middle of the image with the middle of the text, and "*top*" aligns the top of the image with the top of text. The effect of these values is shown in Figure 2.20. Note that text does not wrap around an image, so images within a sentence can leave big gaps between adjacent lines.

ALT="*text alternative*" (optional)—**ALT** provides a text alternative to the image, for use by text-only browsers. This attribute is optional but should be included to let users with text-only browsers know what they are missing. In addition, many graphical browsers can delay image loading and can take advantage of the **ALT** value to *preview* the contents of the image.

```
<html>

<head>
<title> Example of IMG Element </title>
</head>

<body>

<h1> Example 10: Examples of IMG Elements</h1>

<P> <IMG SRC="icon-help.gif" ALT="[Test image]" ALIGN=TOP>
Here is some text related to the test image.   The text is aligned with
the top of the image. Note that the text does not flow around the image.
<P> <IMG SRC="icon-help.gif" ALT="[Test image]" ALIGN=MIDDLE>
Here is some text related to the test image.   The text is aligned with
the middle of the image. Note that the text does not flow around the image.
<P> <A HREF="http://www.bozo.edu/test.html">
<IMG SRC="icon-help.gif" ALT="[Test image]"
ALIGN=BOTTOM> Here is some text </A>  related
to the test image.   The text is aligned with the bottom of
the image, and is also part of the <em> hypertext link</em>.


</body>
</html>
```

Figure 2.19 HTML example document illustrating the **IMG** inline image element. Figure 2.20 shows this document as displayed by the **MacMosaic** browser.

ISMAP (optional)—This attribute marks the image as an *active* image. This allows the user to click the mouse over the image and have different regions of the image cause different actions. When a user clicks on an ISMAP image, the browser measures the coordinates of your mouse pointer (relative to the upper left-hand corner of the image) and sends this information to the server for further processing. Typical markup for an active image is:

```
<A HREF="http://www.utirc.ca/cgi-bin/imagemap/map_bozo">
     <IMG SRC="bozo.gif" ISMAP>
</A>
```

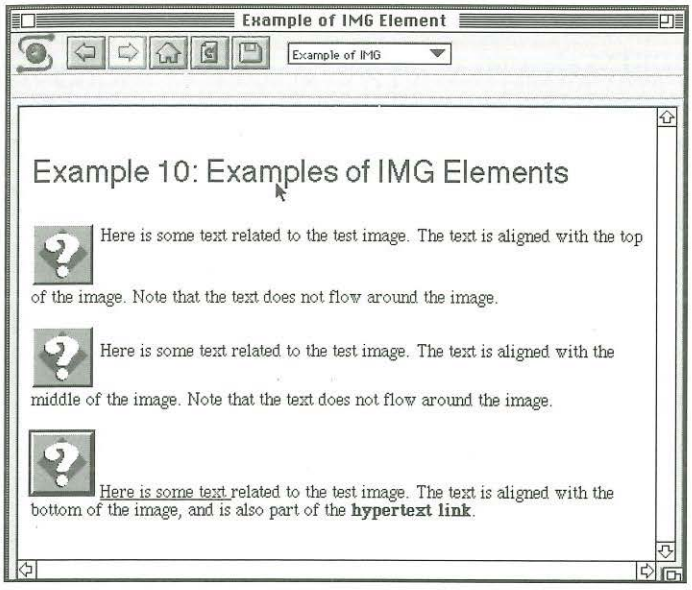


Figure 2.20 Display, by the **MacMosaic** browser, of the document shown in Figure 2.19.

where *imagemap* is a server-side gateway program that can interpret the map coordinates. The details of this procedure are discussed in Chapter 5.

BR ELEMENT: LINE BREAK

Usage:

Can Contain: empty

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, DD, DT, LI

Attributes: none

BR indicates a line break. This is fundamentally different from a paragraph mark. A
 is treated as a character (like a *hard* carriage return),

whereas P defines a block of text as a paragraph element.
 is commonly used in the ADDRESS element; an example of this usage is shown in Figure 2.6. BR can also break lines in a poem, with P elements marking the different verses.

CHARACTER HIGHLIGHTING ELEMENTS

HTML allows you to specify special character highlighting or emphasis (e.g., boldface, italics, etc.). These elements do not cause page breaks or in any way affect the structural layout of the document. Their purpose is to either assign special meanings to the enclosed text, or to request special physical formatting, such as boldface or italics.

These two methods can be described as logical highlighting (also called information-type formatting) and physical highlighting (also called character formatting). Logical highlighting is more in keeping with the markup language model. That is, you can use logical highlighting to mark a block of text as a piece of typed computer code, a variable, or as something to be emphasized. The rendering details are then left to the browser, although hints to appropriate renderings are part of the HTML specifications. You are strongly encouraged to use logical highlighting elements over the physical ones.

Physical highlighting requests a specific physical format, such as boldface or italics. This, of course, gives no clue to the underlying meaning behind the marked-up phrase. Thus, if a browser is unable to implement the indicated markup (e.g., a dumb terminal cannot do italics), it cannot easily determine an alternative logical highlighting.

Logical styles may not be distinct (i.e., different logical styles may be rendered in the same way). Also, some browsers do not support all physical styles. For example, lynx does not support italics and renders it as underlined.

LOGICAL HIGHLIGHTING

Here are the different logical highlighting elements, and typical renderings:

CITE	A citation (usually italics)
CODE	Example of typed code (usually fixed-width font)
DFN	(proposed HTML2.0) The defining instance of a term (often, bold or bold italics)
EM	Emphasis (usually, italics)
KBD	Keyboard input (for example, in a manual)
SAMP	A sequence of literal characters
STRIKE	(proposed HTML2.0) Struck-out text. This might be rendered as text with a line drawn through it
STRONG	Stronger emphasis (usually boldface)
VAR	A variable name

HTML+ proposes the following additional elements. These are not understood by most current browsers:

ARG	A command argument (for a computer command: see CMD following)
ABBREV	An abbreviation
CMD	A command name (for a computer command)
ACRONYM	An acronym for an organization, protocol, and so on
PERSON	A proper name (e.g., `<PERSON>` Ian Graham`</PERSON>`)
Q	A short quotation that could be included inline with the text

You can nest highlighting modes inside one another. This is most often not sensible, given the rather specific meanings assigned to the elements. Also, some browsers misinterpret these nestings and produce inappropriate text formatting.

Examples of the different highlighting elements are shown in Figures 2.21 and 2.22.

```
<html>

<head>
<title> Example of Highlighting Markup elements </title>
</head>

<body>

<h1> Example 11 - Highlighting Elements </h1>
<h2> Examples of Logical Highlighting </h2>

<ul>
<li> CITE - This is <CITE> citation </CITE>text
<li> CODE - This is <CODE> typed computer code </CODE>text.
<li> DFN - This is <DFN> a defining instance </DFN>text. (Proposed HTML2.0)
<li> EM - This is <EM> emphasized </EM>text.
<li> KBD - This is <KBD> keyboard input </KBD>text.
<li> SAMP - This is <SAMP> literal character </SAMP>text.
<li> STRIKE - This is <STRIKE> strike-out</STRIKE> text. (Proposed HTML2.0)
<li> STRONG - This is <STRONG> strongly emphasized </STRONG>text.
<li> VAR - This is <VAR> a variable </VAR>text.
</ul>

<h2> Examples of Character Highlighting </h2>

<ul>
<li> B - This is <B> boldfaced </B>text.
<li> I - This is <I> italicized </I>text.
<li> TT - This is <TT> fixed-width typerwriter font </TT>text.
<li> U - This is <U> underlined </U>text. (Proposed HTML2.0)
</ul>


</body>
</html>
```

Figure 2.21 HTML example document illustrating the different text highlighting elements. Figure 2.22 shows this document as displayed by the **MacWeb** browser.

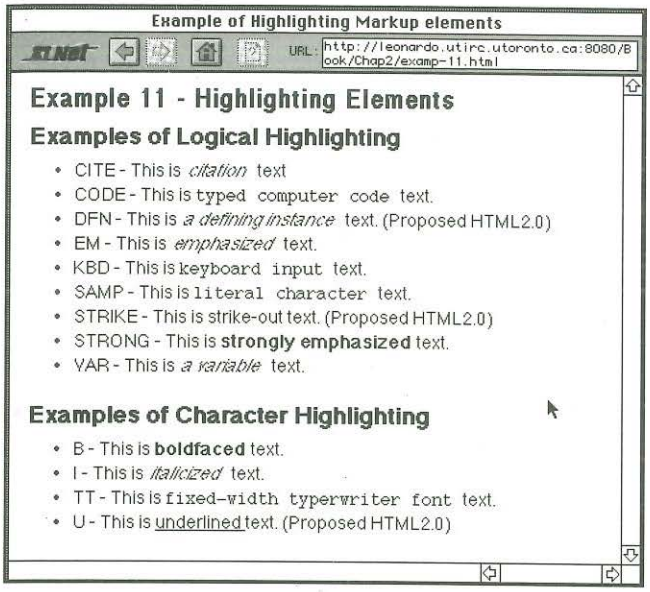


Figure 2.22 Display, by the **MacWeb** browser, of the document shown in Figure 2.21.

CITE Element: Citations

Usage:	`<CITE> ... </CITE>`
Can Contain:	characters, character highlighting, A, BR, IMG
Can Be Inside:	A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI
Attributes:	none

The CITE element marks a small citation—for example, a book or other document reference. Typically, this block of text will be rendered in italics, subject to the capabilities of the browser.

CODE Element: Typed Code

Usage: `<CODE> ... </CODE>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The CODE element marks a selection of typed computer code—for example, a single line of code from a program. Large selections of code should be displayed using a PRE element, which properly reproduces space characters and line breaks. CODE element contents are rendered in a fixed-width typewriter font.

DFN Element: Defining Instance (Proposed: HTML2.0)

Usage: `<DFN> ... </DFN>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The DFN element marks the defining instance of a term. This is useful for generating indexes of definitions, or for searching through documents for defining instances. A browser will typically render this in italics or bold italics. Note that this element is not understood by all browsers.

EM Element: Emphasis

Usage: `<EM> ... </EM>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The EM element marks a block of text for emphasis. Typically, the marked block of text is rendered in italics, subject to the capabilities of the browser. For example, lynx represents EM emphasized text by an underline.

KBD Element: Keyboard Input

Usage: `<KBD> ... </KBD>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The KBD element marks a block of text as keyboard input. Typically, this is displayed with a fixed-width typewriter font.

SAMP Element: Literal Characters

Usage: `<SAMP> ... </SAMP>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The SAMP element marks a block of text as a sequence of literal or *sample* characters. Typically, this is rendered in a fixed-width typewriter font.

STRIKE Element: Struck-Out Text (Proposed: HTML2.0)

Usage: `<STRIKE> ... </STRIKE>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The STRIKE element marks a block of text to be *struck out*. An example would be a legal document, where a sentence or section is struck out but should be left present in the text for reference purposes. Typically, this block of text is rendered in a normal font but with a line drawn horizontally through the text. Other renderings are necessary on text-only browsers, such as lynx. The STRIKE element is not understood by most browsers.

STRONG Element: Strong Emphasis

Usage: `<STRONG> ... </STRONG>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The STRONG element marks a block of text for strong emphasis. Typically, this is rendered in boldface, although text-only browsers, such as lynx, use an underline (with lynx, EM and STRONG emphasis are displayed in the same way).

VAR Element: A Variable

Usage: `<VAR> ... </VAR>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The VAR element marks a variable name. This is typically rendered in italics or bold italics.

PHYSICAL HIGHLIGHTING

Physical highlighting elements request physically desired renderings. Physical styles are useful when translating from another document to HTML, where the only information available may be physical markup instructions. Use logical highlighting in new documents whenever possible.

Physical highlighting elements can be nested and, unlike the case with most logical highlighting elements, these nestings make sense. Therefore, requesting that a block of text be rendered in Underlined-Boldface-Italics is

entirely reasonable. However, be aware that some browsers (for example, **Mosaic For X-Windows**) presently mishandle these nestings.

The physical style elements, and their renderings, are:

B	Boldface (where possible)
I	Italics (may be rendered as slanted in some cases)
TT	Fixed-width typewriter font
U	Underlined (proposed: HTML2.0)

Examples of the renderings of these styles are shown in Figure 2.22.

HTML+ proposes the following additional physical highlighting elements:

S	Strike-through. This does not have the logical meaning associated with the **STRIKE** logical highlighting element.
SUB	Subscript
SUP	Superscript

B ELEMENT: BOLDFACE

Usage:	`<B> ... </B>`
Can Contain:	**characters, character highlighting, A, BR, IMG**
Can Be Inside:	**A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI**
Attributes:	none

The B element marks a section to be rendered in boldface. If this is impossible, the browser can render this in some other way (**lynx** uses an underline).

I Element: Italics

Usage: `<I> ... </I>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The I element marks a section to be rendered in italics. If this is impossible, the browser can render this in some other way (lynx uses an underline).

TT Element: Fixed-Width Typewriter Font

Usage: `<TT> ... </TT>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The TT element marks a section to be rendered with a fixed-width typewriter font.

U Element: Underline (Proposed: HTML2.0)

Usage: `<U> ... </U>`

Can Contain: characters, character highlighting, A, BR, IMG

Can Be Inside: A, character highlighting, ADDRESS, Hn, P, PRE, DD, DT, LI

Attributes: none

The U element marks a section to be rendered with an underline. This element is not understood by all browsers.

HTML+ ELEMENTS

The following elements are part of the HTML+ proposals, and are not understood by most WWW browsers. However, there is a particular push to implement the TABLE element so that this will soon be available on aggressively designed browsers. Other elements that are implemented on some browsers are the FOOTNOTE and MARGIN elements (implemented on emacs-w3), and the SUP and SUB (superscript/subscript) elements implemented on Mosaic for X-Windows Version 2.5.

TABLE Element: Tables (Proposed HTML+)

Usage: `<TABLE> ... </TABLE>`

Can Contain: CAPTION, TH, TD, TR

Can Be Inside: not yet specified; probably:

BLOCKQUOTE, BODY, FORM, DD, LI

Attributes: BORDER

The TABLE element contains tabular data; the structure and contents of the table are defined by additional elements that can appear only inside the TABLE. The TABLE element has one optional attribute:

BORDER (optional)—The border attribute tells the browser to draw a box around each cell in the table, if possible.

The TABLE element can contain four table description elements:

CAPTION	Gives the caption or title of the table.
TH	Table Header: Gives the headings for a particular row or column.
TD	Table Data: Gives the data contents of a table cell.
TR	Table Row: Defines the end of a row and the start of a new row.

The design of HTML TABLEs will be familiar to those who have used the tabular environment in LaTeX. Tables are defined as *cells,* where a cell is an item within the table. Cells can be of various sizes, specified by the number of table rows or columns that are spanned by the cell. It is the author's responsibility to ensure that the number of cells in a given row (or column) add up to the correct table size. An example of a TABLE is shown in Figures 2.23 and 2.24.

CAPTION ELEMENT: TABLE CAPTION (PROPOSED HTML+)

Usage:	`<CAPTION> ... </CAPTION>`
Can Contain:	not yet specified; probably: **characters, character highlighting, A, BR, IMG**
Can Be Inside:	**TABLE**
Attributes:	none

The CAPTION element specifies the table caption or title. Each TABLE can have at most one CAPTION. The CAPTION can contain text, character

highlighting elements, and the IMG element, along with hypertext anchors.

Example:

```
<CAPTION> Profits In each Year: 1983-1993 </CAPTION>
```

TH Element: Table Heading (Proposed HTML+)

Usage: <TH> ... (</TH>)

Can Contain: not yet specified; probably: characters, character highlighting, A, IMG, BR, TABLE

Can Be Inside: TABLE

Attributes: ALIGN, COLSPAN, ROWSPAN

The TH element specifies a Table Header cell—the cell that defines the contents of a row or column. The TH element is not empty, but the end tag </TH> is optional, as its existence is implied by the next <TH>, <TR>, or <TD> tag. The TH element can be present with no contents, which simply means that the cell is blank. TABLES can have more than one TH for a given row or column: It is up to you, the TABLE designer, to ensure that table headings are placed appropriately. The TH element has three optional attributes that define the centering of the contents of the header, and that define how many columns or rows of the entire table are to be spanned by the *cell*.

The attributes of the TH element are:

ALIGN = "*left*", "*right*", "*center*" (optional)—ALIGN specifies the alignment of the table cell contents within the cell: ALIGN="*left*" aligns the contents flush-left within the cell; ALIGN="*right*" aligns the contents flush-right within the cell; ALIGN="*center*" centers the contents within the cell.

ROWSPAN = "*n*" (optional)—ROWSPAN specifies how many table rows are spanned by the cell—the default value is 1. Counting of rows starts from the top of the table. It is the author's responsibility to ensure that the cells in each column sum to the correct number of rows.

COLSPAN = "*n*" (optional)—COLSPAN specifies how many table columns are spanned by the cell—the default value is 1. Counting of columns starts from the left side of the table. It is the author's responsibility to ensure that the cells in each row sum to the correct number of columns.

Example:

```
<TH ALIGN=LEFT> Housing Prices: </TH>
```

This example specifies a table heading cell that spans one row and one column and that contains the heading "*Housing Prices:*." TH cells need not contain simple text strings. They can also contain other TABLEs, lists, images, hypertext anchors, and so on.

```
<html>

<head>
<title> HTML+ Features>: TABLEs, SUB and SUP</title>
</head>

<body>

<h1> Example 12 HTML+ Features</h1>

<p> HTML+ proposed several new elements. among them are
<ul>
<li> superscript element: hello<sup>2</sup>
<li> subscript element: sub<sub>script</sub>.
```

```
</ul>
<p>
Also defined are TABLE elements that allow you to construct
formatted tables.  Here is an example:
<p>
<table border>
<caption>Table Caption </caption>
<th rowspan=2>  Item                          </th>
<th colspan=2>  Breakdown Costs               </th>
<tr>
<th align=left> Pre- </th>
<th align=left> Post- </th>
<tr>
<td> Carbuncles                               </td>
<td align=left>     0.00                      </td>
<td align=left>  $12,000                      </td>
<tr>
<td>Klystrons (a)                             </td>
<td rowspan=2 align=left>$13,000              </td>
<td align=left>$8,000                         </td>
<tr>
<td>Klystrons (b)                             </td>
<td align=left><em> $8,000</em></td>
<tr>
<td>Sticks                                    </td>
<td align=left> 23&#162;                      </td>
<td align=left> 13&#162;                      </td>
</table>

</body>
</html>
```

Figure 2.23 HTML example document illustrating the **TABLE**, **SUP**, and **SUB** elements. Figure 2.24 shows this document as displayed by the **Mosaic for X-Windows** Version 2.5 browser.

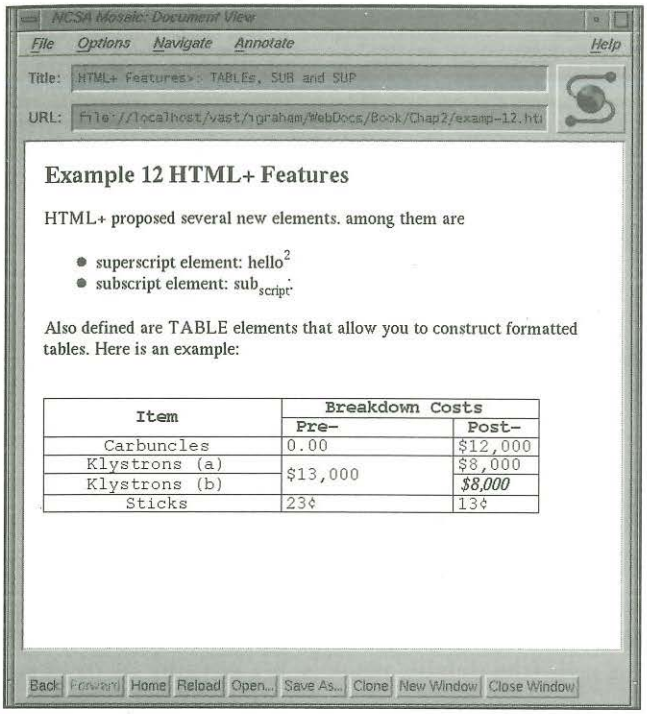


Figure 2.24 Display, by the **Mosaic for X-Windows** Version 2.5 browser, of the document shown in Figure 2.23.

TD Element: Table Data (Proposed HTML+)

Usage: `<TD> ... ( </TD> )`

Can Contain: Not yet specified; probably: characters, character highlighting, A, BR, IMG, TABLE

Can Be Inside: TABLE

Attributes: ALIGN, COLSPAN, ROWSPAN

The TD element specifies a Table Data cell, which is a cell that contains the tabular data. The **TD** element is not empty, but the end tag `</TD>` is optional, as its existence is implied by the next `<TH>`, `<TR>`, or `<TD>` tag. The **TD** element can have empty contents, which simply means that the cell is blank. The **TD** element has three optional attributes that define the centering of the contents of the cell and how many columns or rows of the entire table are to be spanned by the cell. These have the same meanings as with the **TH** element.

Attributes:

ALIGN = "*left*", "*right*", "*center*" (optional)—**ALIGN** specifies the alignment of the table cell contents within the cell: **ALIGN**="*left*" aligns the content flush-left within the cell; **ALIGN**="*right*" aligns the contents flush-right within the cell; **ALIGN**="*center*" centers the contents within the cell.

ROWSPAN = "*n*" (optional)—The **ROWSPAN** attribute specifies how many rows are spanned by this cell—the default value is 1. Counting of rows starts from the top of the table. It is the author's responsibility to ensure that the cells in each column sum to the correct number of rows.

COLSPAN = "*n*" (optional)—The **COLSPAN** attribute specifies how many columns are spanned by this cell—the default value is 1. Counting of columns starts from the left-hand side of the table. It is the author's responsibility to ensure that the cells in each row sum to the correct number of columns.

Example:

```
<TD center colspan=1, rowspan=2> 23.22 </TD>
```

This example specifies a table data cell that spans one column and two rows, and that contains the data value 23.22. **TD** cells need not contain numbers, or even simple text strings; they can also contain other **TABLE**s, lists, images, hypertext anchors, and so on.

TR Element: End of Table Row (Proposed HTML+)

Usage: <TR>

Can Contain: empty

Can Be Inside: TABLE

Attributes: none

The TR element indicates the end of a table row. Thus, a row in a table might be coded:

```
<TH> Heading <TD> data 1 <TD> data2 <TD> data3 <TD> data4 <TR>
```

which indicates a row containing five columns; the first column containing a table heading, and the rest containing table data. Every row must be terminated by a <TR>, except for the final row, where the final <TR> is implied by the </TABLE> tag ending the table.

ABSTRACT Element: Abstracts (Proposed HTML+)

Usage: <ABSTRACT> ... </ABSTRACT>

The ABSTRACT element is used to give an abstract of a document. This would be part of the BODY and hence would be displayed, probably with an indent and a noticeably different font.

FIG Element: Floating Figures (Proposed HTML+)

Usage: <FIG> ... </FIG>

The FIG element is designed to be a more sophisticated way of including images within documents. The FIG element would indicate the desired image as an attribute to the FIG (e.g., `<FIG SRC="image.gif">`, just as with the IMG element), but would also allow for an image CAPTION and for typed text associated with the image. The resulting FIG element is then treated as a paragraph, with the image and associated CAPTION floating within the paragraph for most appropriate formatting, and with the text allowed to flow around the image.

FIG images can also be ISMAP active images, as with the IMG elements. Another proposed extension is to make FIG images active by including geometry information about the active region with the document itself. This would allow you to click on a region and access a document without first having to access a server to parse the image map. The FIG element is not currently implemented.

FOOTNOTE Element (Proposed HTML+)

Usage: `<FOOTNOTE> ... </FOOTNOTE>`

The FOOTNOTE element would be used to include footnote annotations within a document. This may appear in the document as a special footnote icon or symbol indicating the presence of a footnote; it does not have to appear at the bottom of the document page. Clicking on the button should *pop up* the footnote, displaying it next to or on top of the current document. FOOTNOTEs should be small and could contain images and text highlighting elements.

The emacs-w3 browser supports the FOOTNOTE element.

LIT Element: Literal Text (Proposed HTML+)

Usage: `<LIT> ... </LIT>`

In HTML2.0, preformatted text can be displayed only using the **PRE** environment. This forces the text into a fixed-width typewriter font. This preformatted mode is certainly not ideal for such text as poetry or other prose, for which the line breaks and indents are important, and where you want easier to read variable-width fonts.

The **LIT** element is proposed for presenting such information. The **LIT** element would preserve line breaks in the typed text and preserve the special meanings of **TAB** characters (presently, **TAB**s are treated as simple white space and are ignored). The actual indent to associate with a **TAB** would be specified as an attribute of **LIT** (for example, `<LIT AT=10 AT=20 > .. </LIT>`).

MARGIN Element: Margin Notes (Proposed HTML+)

Usage: `<MARGIN> ... </MARGIN>`

The **MARGIN** element would be used to include a margin note for a particular paragraph. This might appear in the document as a special icon note next to the **MARGIN**ed paragraph. Clicking on the icon would pop up the margin note in a small overlay window, and would retain the current document in the main viewing window. **MARGIN** notes should be small, but could contain images and text highlighting elements. The **emacs-w3** browser supports the **MARGIN** element.

MATH Element: Mathematical Expressions (Proposed HTML+)

Usage `<MATH> ... </MATH>`

The MATH element is proposed to allow for the construction (and correct display on graphically capable terminals) of complex mathematical expressions, such as equations, integrals, and so on. Unfortunately, this is not currently implemented.

The best way currently to present equations in HTML documents is to convert the equations or other mathematical symbols into small image files and include them inline with the text. The program **latex2html** does this automatically when converting LaTeX documents into HTML. **Latex2html** is discussed in more detail in Chapter 6.

NOTE Element: Exclamatory Notes (Proposed HTML+)

Usage: `<NOTE> ... </NOTE>`

The NOTE element would be used when you want to display an exclamatory note. This could be displayed highlighted from the regular text, indented, and perhaps with a special notifying icon. The NOTE element can take two attributes: the SRC="*URL*" attribute, which could specify an optional notifying icon to be used to mark the NOTE, while ROLE= "*tip*", "*note*", "*warning*", or "*error*" could be used to specify the nature of the NOTE.

ONLINE/PRINTED ELEMENTS: (PROPOSED HTML+)

Usage: `<ONLINE> ... </ONLINE>`

`<PRINTED ... </PRINTED>`

The ONLINE and PRINTED elements would be used to give two different representations for the same piece of the document: one to be used with the interactive electronic format (ONLINE), and the other for use with a printed version (PRINTED). This may be useful for giving a printed text alternative to a hypertext link so that the printed version would give more details about the link or the related reference.

REFERENCES

An introduction to SGML:

`http://etext.virginia.edu/bin/tei-tocs?div=DIV1&id=SG`

The HTML 2.0 draft specification:

`http://www.hal.com/products/sw/olias/Build-html/CGQL0ZBEfmg24aK.html`

The HTML2.0 Document Type Definition (DTD):

`http://www.hal.com/%7Econnolly/html-spec/html.decl`

`http://www.hal.com/%7Econnolly/html-spec/html.dtd`

HTML references:

`http://akebono.stanford.edu/yahoo/Computers/World_Wide_Web/HTML/`

HTML+ references:

`http://akebono.stanford.edu/yahoo/Computers/World_Wide_Web/HTML/HTML_/`

CHAPTER

3

UNIFORM RESOURCE LOCATORS (URLS)

This chapter explains the structure and syntax of the *Uniform Resource Locator*, which is the addressing scheme used in HTML documents to indicate the target of hypertext links. The URL scheme allows WWW browser clients to send search data to servers for further processing. This mechanism is explained here, aand also in Chapter 4.

Uniform Resource Locators, or *URLs*, are a scheme for specifying Internet resources using a single line of printable ASCII characters. This scheme encompasses all major Internet protocols, including FTP, Gopher, HTTP, and WAIS. URLs are one of the foundation tools of the World Wide Web and are used within HTML documents to reference the targets of a hypertext link. However, URLs are not restricted to the World Wide Web and can be used to communicate information about Internet resources in e-mail letters, handwritten notes, or even books.

161

A URL contains the following information:

- The *protocol* to use when accessing the server (e.g., HTTP, Gopher, WAIS).
- The Internet *Domain Name* of the site on which the server is running, along with any required username and password information.
- The *Port Number* of the server; if this is omitted, the browser assumes a commonly understood default value of the indicated protocol.
- The *location* of the resource in the hierarchical (often, directory) structure of the server.

Here is a typical example; in this case, for the HTTP protocol:

```
http://www.cern.ch/hypertext/WWW/RDBgate/Implementation.html
```

This references the file ***Implementation.html*** in the directory ***/WWW/RDBgate,*** accessible at the server ***www.cern.ch*** using the HTTP protocol.

ALLOWED CHARACTERS IN URLS

Every URL must be written using printable ASCII characters (the bottom half of the ISO Latin-1 character set, as discussed in Appendix A, and excluding the ASCII control characters), and *cannot* be written using the full ISO Latin-1 character set. This restriction ensures that URLs can be sent by electronic mail; many electronic mail programs mishandle characters from the upper half of the ISO Latin-1 character set. In a URL, any ISO Latin-1 character can be represented using a *character encoding* scheme. This is analogous to the character entities used with HTML. However, the schemes are distinctly different: You must *never* use HTML characters or entity references in a URL.

The mechanism is simple: Any ISO Latin-1 character can be represented by the encoding:

```
%xx
```

where the percent sign is a special character indicating the start of the encoding and where *xx* is the *hexadecimal code* for the desired ISO Latin-1 character (the *x* represents a hexadecimal digit in the range `[0-9,A-F]`). Table A.1 (Appendix A) lists all the ISO Latin-1 characters alongside their hexadecimal codes. As an example, the encoding for the character *é* (the letter *e* with an acute accent) is `%E9`.

DISALLOWED CHARACTERS

Several ASCII characters are disallowed in URLs and can be present only in encoded form, because these characters often have special meanings in a non-URL context and may be misinterpreted. For example, HTML documents use the double quotation mark (") to delimit a URL in a hypertext anchor so that a quotation mark inside the URL would end the URL prematurely. Therefore, the double quote is disallowed. The space character is also disallowed, since many programs will consider the space as a break between two separate strings. Space characters often appear in Macintosh file or folder names. For example, the filename Network Info (where there is a single space between the words Network and Info) must be encoded in a URL as:

```
Network%20Info
```

The disallowed characters are summarized in Table 3.1.

Table 3.1 Disallowed ASCII Characters in URLs

Character	Hex	Character	Hex
TAB	09	SPACE	20
"	22	<	3C
>	3E	[	5B
\	5C	]	5D
^	5E	`	60
{	7B	|	7C
}	7D	~	7E

SPECIAL CHARACTERS

In a URL, several ASCII characters have special meanings, such as the percent character that denotes character encodings. The forward slash character (/) is also special and denotes a change in hierarchy, such as a directory change. These special characters must be encoded if you want them to appear as regular uninterpreted characters. Thus, to include the string:

```
ian%euler
```

in a URL, you must encode it as:

```
ian%25euler
```

where `%25` is the encoding for the percent character. If you do not do this, a program parsing the URL will try and interpret `%eu` as a character encoding. Conversely, you must not encode a special character if you require its special meaning. For example, the string:

```
dir/subdir
```

indicates that `subdir` is a subdirectory of `dir`, while:

```
dir%2Fsubdir
```

is just the character string `dir/subdir` (`%2F` is the encoding for the slash).

The general rule is: Encode any character that might be special if you do not want to use its special meaning. The more common special characters are listed in the following:

- The percent sign (%). This is the escape character for character encodings.
- The slash (/). This is used to indicate hierarchical structures, such as directories.
- The hash (#). This is used to separate the URL of a resource from the *fragment identifier* for that resource. A fragment identifier references a particular location within a resource (special in HTTP URLs only).

- The question mark (?). This indicates a *query string*; everything after the question mark is query information to be passed to the server (special in Gopher, WAIS, and HTTP URLs only).

EXAMPLE OF A UNIFORM RESOURCE LOCATOR

Figure 3.1 shows a typical URL (in this case, for the HTTP protocol) with the different parts and the associated meanings. The different parts are discussed in subsequent sections.

1. PROTOCOL

The first string in the URL specifies the Internet *protocol* to use in accessing the resource; this example requests the HTTP protocol. The protocol is indicated by the name appearing before the first colon in the URL. URL schemes are defined for most Internet protocols, including FTP, Gopher, HTTP, telnet, and WAIS. The details of the different schemes are presented later on.

2. ADDRESS AND PORT NUMBER

The second part of this URL is the Internet address of the server; this information lies between the double forward slash (//) and a terminating forward slash (/). This example gives the *domain name* of the server and the *port number* to contact. Omitting the port number implies the default value for the given protocol. Numeric IP addresses can be used instead of domain names; for example:

```
//132.206.9.22:1234/
```

You can include username and password information if this is needed to access a resource. The format is:

```
//username:password@www.address.edu:1234/
```

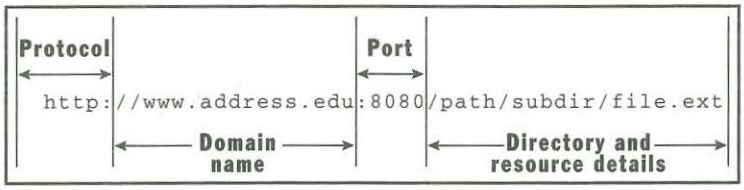


Figure 3.1 Example of a URL.

Note that the password can be read by anyone who reads the URL, so this is not a secure way to allow access to a resource.

3. RESOURCE LOCATION

The forward slash after the host and port number field indicates the beginning of the *path* information required to locate the resource on the server. This field varies considerably, depending on the service being accessed. Often, this resembles a directory path leading down to a file, as in Figure 3.1. In this context, the forward slash character (`/`) is used in place of *all* system-dependent symbols, such as the backslash (`\`) on DOS or Windows computers; the colon (`:`) on Macintoshes; and the `[dir.subdir.subsubdir]` expressions used by VAX/VMS systems.

Passing Search Strings

The URL syntax allows you to pass query strings to the designated server, in situations (typically, Gopher, HTTP, or WAIS) that support such service. This is accomplished by appending the query strings to the URL, separated from the URL by a question mark. Two examples are:

```
gopher://gopher.somewhere.edu/77/searches.phone?bob+steve
http://www.somewhere.edu/cgi-bin/srch-data?archie+database
```

Special Cases

In some cases, Internet domain names are not part of a URL. This is the case for protocols that are not dependent on a particular server, such as sending electronic mail or accessing USENET newsgroup articles.

SOME SIMPLE URL EXAMPLES

```
http://info.cern.ch/hypertext/WWW/Addressing/URL/Overview.html
```

This references the file *Overview.html* in the directory */hypertext/WWW/Addressing/URL/* from the server *info.cern.ch* using the HTTP protocol and the default port number (80 for HTTP).

```
gopher://gumby.brain.headache.edu:151/7fonebook.txt
```

This references the searchable index *fonebook.txt* from the Gopher server at *gumby.brain.headache.edu* running on port number `151`.

```
news:alt.rec.motorcycle
```

This references the newsgroup *alt.rec.motorcycle*.

```
mailto:ross@physics.mcg.ca
```

This indicates sending an electronic mail message to the indicated e-mail address.

PARTIAL URLS

Within a given HTML document, you do not need to specify the entire URL to locate a second, neighboring document. This is because being *in* the document already implies information about the current URL, so that you can reference neighboring documents or resources using a *partial* URL that gives the location *relative* to the current document.

Suppose you originally access the document *file.html* using the full URL:

```
http://www.stuff.edu/main/docs/file.html
```

Within this document there is a hypertext reference containing a *partial* URL:

```
<A HREF="stuff.html"> anchor text </A>
```

Where is this file? From inside *file.html*, any information not present in a URL reference is considered the *same* as that used to access the current document. Thus, the partial URL `stuff.html` is transformed into a full

URL by appropriating the missing information from the URL used to access ***file.html***. The completed URL is then:

```
http://www.stuff.edu/main/docs/stuff.html
```

which indicates, as expected, that ***stuff.html*** is on the same server and in the same directory as ***file.html***. Other equivalent partial URLs are then:

```
/main/docs/stuff.html
//www.stuff.edu/main/docs.html
```

The former appropriates http://www.stuff.edu from the current URL to complete the reference, while the latter appropriates only the http: part from the *base* URL of the current document.

You can also use partial URLs to reference files in other directories; for example, from the example ***file.html*** the relative URL:

```
../../main.html
```

indicates the file ***main.html*** in the root HTTP directory, namely:

```
http://www.stuff.edu/main.html
```

Partial URLs are very useful when constructing large collections of documents that will be kept together. Of course relative URLs become invalid if a document is moved to a new directory or a new Internet site. This problem can be mitigated using the **BASE** element of the HTML, which is used to record the correct ***BASE*** URL of a document. If the document is moved, all relative URLs are determined relative to the URL recorded by the **BASE** element.

URL SPECIFICATIONS

We now look at the details of the different URL forms. The protocols most commonly referenced by URLs are:

```
ftp:
gopher:
http:
```

```
mailto:
news:
telnet:/rlogin:/tn3270:
wais:
file: (local file access)
```

FTP URLS

Ftp URLs designate files and directories accessible using the FTP protocol. In the absence of any username and password information, anonymous FTP access is assumed. This connects you to the designated server as user *anonymous* and uses your Internet mail address as the password. Here is a typical example, referencing the file ***splunge.txt*** located in the directory ***path*** at the Internet site ***internet.address.edu***:

```
ftp://internet.address.edu/path/splunge.txt
```

You can specify an FTP access of a directory using a URL such as:

```
ftp://internet.address.edu/path/
```

In this case, FTP servers provide a list of directory contents. WWW browsers display this information as a menu, allowing the user to navigate through the filesystem or select particular files for downloading.

Non-Anonymous FTP Access

You can reference non-anonymous FTP resources by specifying, as part of the URL, the username and password of the account you wish to access. For example,

```
ftp://joe_bozo:bl123@internet.address.edu/dir1/Dir2/file.gz
```

allows a user to access files on machine ***internet.address.edu*** belonging to user `joe_bozo` with password `bl123`. This is, however, not a secure way of giving access to this file, since anyone who reads the URL knows `joe_bozo`'s password. If you have information you want to make available via FTP, set up a proper anonymous FTP service.

FTP Directory Paths

Directory paths in FTP accesses are defined relative to a home directory. This home directory is different, depending on how you accomplish the FTP connection. If someone connects to a machine via anonymous FTP, they are placed in the anonymous FTP home directory and have restricted access to the server filesystem. This is a security feature that allows you to make files available to the public without exposing your system to unauthorized users. On the other hand, if someone connects to the same machine as a registered user, their home directory will be that of the user, entirely different from the anonymous FTP directory.

All of this means that the path you use to access a file as a regular user can be quite different from the path used to access a file by anonymous FTP. You must keep this in mind when archiving data into anonymous FTP archives and when creating URLs that point to these data.

Modes for File Transfers

The FTP protocol allows several modes for transferring files. The most important is *image* or *binary* mode, which makes a byte-by-byte copy of the file. This is the mode to use when transferring programs, compressed data, or image files. Also common is *ASCII* mode, which should be used to transfer plain, printable text files. This mode is useful because it *corrects* for the fact that PCs, Macintoshes, and UNIX computers use different characters to mark the end of a line of text. In particular, Macintoshes use the carriage-return character CR; UNIX computers use the line-feed character LF; and DOS/Windows computers use both CR and LF (often written CRLF). In ASCII mode, FTP automatically converts between these three end-of-line markers to ensure that the received file has the new line codes appropriate to the local system. You cannot use this mode to transfer programs, however, since programs and data files will contain bytes with the same codes as CR or LF characters, and, under ASCII mode, these codes will be converted into the new line codes appropriate to the local system, which destroys the data content of binary data files.

The FTP protocol has no knowledge of the data contents of a file and must be told what mode to use in a file transfer. Thus, your WWW browser must have some way of determining the data type of a file being accessed via an **ftp** URL. In general, the browser *guesses* the type from the suffix of the filename. All WWW clients maintain a local database that maps filename extensions onto data types. For example, **Mosaic for X-Windows** maintains this database in a file called ***mime.types***. WWW browsers use this information to guess the contents of a file and to select the appropriate transfer mode.

Troubleshooting Ftp URLs

At times, an **ftp** URL request may fail, either because the network is down or because the machine you are trying to access is overloaded with FTP accesses and refuses your request. Alternatively, the server may be configured to restrict access to only certain Internet sites. WWW browsers are notoriously terse in handling FTP connections, and say very little if a connection has failed. You can check for the cause of the problem by using an FTP connection outside of your WWW application. A stand-alone FTP program provides much more commentary on the state of your connection and will often give you a message explaining why the connection cannot be made.

GOPHER URLS

Gopher servers can be accessed via URLs in a manner that looks superficially similar to that of FTP or HTTP servers, but is, in fact, quite different. This is because Gopher resources are accessed using a combination of *resource identifier* codes and *selector strings*, so a **Gopher** URL is quite different from an **ftp** or **http** URL. Resource identifiers are single-digit codes that specify the *type* of the Gopher resource; for example, that it is a text file, a directory, or a searchable index. The Gopher selector string is just a name associated with this resource. This can be a directory or filename, but can also be a redirection to a database search procedure or a telnet session.

Sometimes, the selector string has, as its first character, a duplicate of the single-character resource type identifiers. Table 3.2 summarizes the Gopher resource identifier codes.

Table 3.2 Gopher Resource Identifier Codes

Code	File/Resource Type
0	Text file
1	Directory
2	CSO Name/phone book server
3	error
4	Macintosh binhexed (*.hqx) file
5	DOS binary file of some type
6	UNIX uuencoded file
7	Full text index search
8	Telnet session
9	Binary file

General Form of a Gopher URL

The general form for a Gopher URL is:

```
gopher://domain.name.edu/Tselector_string
```

where `T` is the Gopher type code and `selector_string` is the Gopher selector string. The root information of a Gopher server can be obtained by leaving out all type and selector string information. Thus, the root information of the preceding Gopher server is available through the URL:

```
gopher://domain.name.edu/
```

Hierarchical relationships are possible. For example:

```
gopher://domain.name.edu/1stuff
```

would retrieve the Gopher contents of the indicated directory, while the URL

```
gopher://domain.name.edu/7stuff/index
```

means access to the ***index*** search in the directory ***stuff***.

Passing Search Information

Search information is sent to the Gopher server by appending the search strings to the URL, separated from the URL by a question mark. Thus, to pass the strings `bob`, `carol`, and `ted` to the Gopher search index noted above, the URL is:

```
gopher://domain.name.edu/7stuff/index?bob+carol+ted
```

Note that the URL syntax for Gopher queries uses a plus (+) sign to separate different search strings. Therefore, if you want to send a plus sign as part of a string, it must be encoded. The encoding for a plus sign is `%2B`.

Client Construction of Query Strings

Inserting plus sign separators and converting plus signs in query strings into encoded values is done by the WWW or Gopher client. When a user accesses a Gopher search from a WWW browser, he or she is prompted for search strings. These are generally entered in a text box, using space characters or tabs to separate the different strings. When you submit the search, the strings are appended, with appropriate encodings, to the URL. The client software is responsible for replacing space characters by plus signs and for encoding characters in your search string that might be incorrectly interpreted.

Gopher URL Examples

`gopher://gopher.utirc.utoronto.ca/`

Retrieve the home menu from the Gopher server running on the machine *gopher.utirc.utoronto.ca* running on the default port number (70).

`gopher://gopher.utirc.utoronto.ca/1adaptive.technology`

Retrieve the adaptive technology menu from the indicated Gopher server running on the default port number. The selector string for this directory is 1adaptive.technology.

`gopher://gumby.brain.headache.edu:151/7fonebook.txt?bob+carol+ted+alice`

Access the searchable index fonebook.txt from the named Gopher server, running on port number 151, and pass to it the four indicated search strings. The Gopher selector string is fonebook.txt.

The Gopher protocol is currently being enhanced, and more sophisticated URLs will be needed to access these newer Gopher features. For more information, please consult the References at the end of this chapter.

HTTP URLS

Http URLs designate files, directories, or server-side programs accessible using the HTTP protocol. **Http** URLs follow the general form indicated in Figure 3.1. Here are some examples:

`http://www.site.edu:3232/cgi-bin/srch-example`

References the program ***srch-example*** at the site ***www.site.edu***, accessible through the HTTP server running on port 3232.

`http://www.utirc.utoronto.ca/`

Access the root directory of the indicated HTTP server. The server can be configured to deliver a standard HTML document, a listing of the directory contents, or an error message.

An **http** URL must always point to a file (text or program) or a directory. A directory is indicated by terminating the directory name with a forward slash, such as:

```
http://www.site.utoronto.ca/HTMLdocs/
```

The following reference to this directory is an error and implies that you are referring to a file, and not a directory:

```
http://www.site.utoronto.ca/HTMLdocs
```

Passing Parameters to the Server in a URL

Http URLs can pass strings to be passed to the server. These strings are appended to the URL separated by a question mark, as with Gopher query strings. Two common examples follow:

```
http://some.site.edu/cgi-bin/foo?arg1+arg2+arg3
```

This is equivalent to the Gopher mechanism discussed in the previous section, and passes the three strings `arg1`, `arg2`, and `arg3` to the server.

```
http://some.site.edu/cgi-bin/imagemap?ix,iy
```

This mechanism is used by active images. For example, this URL is implied by the following HTML markup:

```
<A HREF=http://some.site.edu/cgi-bin/imagemap>
        <IMG SRC="image.gif" ISMAP>
</A>
```

The **ISMAP** directive declares this image to be an active image map, while the surrounding hypertext anchor indicates the URL to which the image coordinates should be sent. When the user clicks the mouse over the image, the coordinates of the mouse pointer relative to the upper left-hand corner of the image (`ix` and `iy`) are sent by appending them to the indicated URL. The ***imagemap*** program must know how to interpret this information. This is discussed in more detail in Chapter 5.

In these examples, the information is passed to a resource in the ***/cgi-bin*** directory. In general, HTTP servers do not handle query data directly and instead pass it on to other *gateway* programs for further processing. The directory name ***/cgi-bin*** is a symbolic name used by several servers to reference a directory containing gateway programs.

Fragment Identifiers

A single HTML document can use the **NAME** attribute of the anchor element to specify distinct targets of hypertext references. For example:

```
<A NAME="raw"> hypertext link target </A>
```

marks the string as a possible target of a hypertext link. In URLs, these named locations are called *fragments* and can be referenced in a URL by appending the *fragment identifier* to the document URL, separated from it by the hash (`#`) sign. For example, suppose the preceding anchor is found in the document:

```
http://site.world.edu/dir/data.html
```

The string `"hypertext link target"` is then explicitly referenced by the URL:

```
http://site.world.edu/dir/data.html#raw
```

You can also use fragment identifiers from within a single document by using a partial URL that references only the fragment identifier, prepended by the hash sign. Thus, within the document *data.html*, the string `"hypertext link target"` is referenced with the URL `#raw`. For example:

```
<A HREF="#raw"> raw data </a>
```

Server Issues: Special Program Directories

URL resource specification strings beginning with the strings `/cgi-bin` or `/htbin` are often special and refer to directories containing programs and scripts (this is an HTTP server feature and these names can be changed in the server configuration files). When a server is contacted via a URL referencing a gateway program, the server launches the program and passes data from the client (if any) to this program for further processing. This is discussed in more detail in Chapter 4. However, here are two simple examples, with brief explanations:

`http://some.site.edu/cgi-bin/srch-example`

The server executes the program ***srch-example*** found in the ***cgi-bin*** directory. Any output from ***srch-example*** is sent back to the client.

`http://www.site.edu/cgi-bin/srch-example/path/other?srch_string`

The server again executes the program ***srch-example*** found in the ***cgi-bin*** directory. The extra *path information* path/other is passed as a parameter to ***srch-example***, as is the query information srch_string.

Server Issues: Personal HTML directories

Users who have accounts on a machine running the NCSA HTTP server can have world-accessible HTML documents in their own home directories, distinct from those files in the server hierarchy. With the NCSA server, these *personal* HTTP document directories are indicated in a URL by a tilde (~) character prepended in front of the path information (the first item in the path hierarchy following the tilde must be the account name of the user). The tilde tells the server that this is not a regular directory, but a

redirection to a personal document archive of the user with the indicated account. For example, if the user `iang` has a personal document directory, this could be accessed using the URL:

```
http://site.world.edu/~iang/
```

MAILTO URLS

The **mailto** URL indicates that mail is to be sent to the designated electronic mail address. The format for **mailto** is straightforward:

```
mailto:mail_address
```

where `mail_address` is the address to which the message should be sent. Typically, this is of the form `name@host`, where `name` is the username of the person, and `host` is the name of the machine. A browser will allow the user to type in the mail message.

Some mail addresses contain the percent character. This character must be encoded, since it is a special character (marking the beginning of a character encoding string). As an example, the e-mail address:

```
jello%ian@utirc.utor.ca
```

must be converted into the **mailto** URL:

```
mailto:jello%25ian@utirc.utor.ca
```

Several browsers, such as **Mosaic**, **WinWeb**, and **MacWeb**, do not support **mailto** URLs.

NEWS URLS

News URLs reference USENET newsgroups or individual USENET news articles. Newsgroups are specified via the URL:

```
news:news.group
```

where `news.group` is the name of a particular newsgroup. Note that this does not specify an NNTP server from which news can be accessed. This must be specified elsewhere and in a browser-specific manner. For example, **MacWeb** users can configure this with a pull-down menu, while UNIX

Mosaic for X-Windows users must set the NNTPSERVER environment variable to the name of the desired server.

TELNET/TN3270/RLOGIN URLS

You can use a **telnet** URL to indicate a telnet link to a remote machine. An example is:

```
telnet://flober.rodent.edu
```

If you are connecting to a machine that requires IBM 3270-terminal emulation, you can substitute `tn3270` for `telnet`, provided your own machine has a tn3270 client program and that your WWW client is configured to know about this program. Note that you cannot include username and password information in a telnet URL.

Rlogin connections can be indicated using the **rlogin** URL. The specification is:

```
rlogin://username@flober.rodent.edu
```

where `username` is the account name on the remote machine. You will be prompted for a password in the resulting window unless one is not required. Many browsers emulate **rlogin** URLs using a telnet connection.

WAIS URLS

WAIS servers can be accessed via URLs in a manner similar to HTTP servers. The major difference is in the file specification: Here, we must pass the correct search instructions to the WAIS server, in addition to information about what is to be searched.

The standard form for accessing a WAIS server is:

```
wais://wais.server.edu/database?search
```

where `wais.server.edu` is the Internet domain name of the host running the WAIS server (including a port number, if required); `database` is the name of the WAIS database to be searched; and `search` is a list of search instructions to pass to the database. Another form is:

```
wais://wais.server.edu/database
```

which designates a particular searchable database. A browser will understand that this URL references a searchable database and will prompt for query string input.

FILE URLS

File URLs are specific to a local system and should not be used in documents to be accessed over the Internet. A **file** URL represents access of files from computers on a network, particularly from the local computer, but does not specify a protocol for accessing these files. The general form for a **file** URL is:

```
file://hostname/path/file
```

where `hostname` is the domain name for the system, and `path/file` is the locator of the file. The **file** URL is most commonly used to represent local file access. All browsers allow local file access, often with a pull-down menu, and represent the location of the selected file using a **file** URL. The domain name for local file access can be either the special string `localhost` or an empty field. For example, if you are accessing the local file ***/big/web/docs.html***, the **file** URL could be either of the following:

```
file://localhost/big/web/docs.html
file:///big/web/docs.html
```

Note that the path is the absolute path to the file from the root of the file system.

REFERENCES

IETF working draft specification for URLs:

```
ftp://ds.internic.net/internet-drafts/draft-ietf-uri-url-08.txt
```

(If this file is not available you can check the status of the IETF draft document from the document:)

```
ftp://ds.internic.net/internet-drafts/1id-abstracts.txt
```

Hypertext URL documentation:

```
http://info.cern.ch/hypertext/WWW/Addressing/URL/Overview.html
http://info.cern.ch/hypertext/WWW/Addressing/Addressing.html
```

CHAPTER

4

THE HTTP PROTOCOL AND THE COMMON GATEWAY INTERFACE

To develop interactive HTML documents, you need to understand the interaction between the WWW clients and an HTTP server. This interaction involves two distinct but closely related issues. The first is the HTTP protocol used for communication with HTTP servers. This protocol has several specific communication *methods* (for example, GET, POST, or HEAD) that allow clients to request data from the server and send information to the server.

The second issue is the way HTTP servers handle a client's requests. If the request is for a file, the server simply locates the file and sends it, or sends an appropriate error message if the file is unavailable. Of interest here is the situation when the client wants to send information to the server for more complicated processing. In general, servers do not do this processing themselves and instead *hand off* the work to other programs called

gateway programs. The *Common Gateway Interface* (*CGI*) specification defines the mechanisms by which HTTP servers communicate with gateway programs. You therefore need to understand both the HTTP protocol and the CGI specification to write server-side gateway programs and client HTML documents that use these programs.

Gateway programs are referenced using URLs, as with files. When a client accesses a URL pointing to a gateway program, the server activates the program and uses the CGI mechanisms to pass to the program data sent by the client (if any). The gateway program acts on the data and returns its response back to the server, again using the CGI mechanisms. The server then forwards the data to the client that initiated the request using the HTTP protocol, completing the transaction.

This chapter begins with a description of the HTTP protocol, followed by an examination of the Common Gateway Interface specification, including examples of gateway programs that receive data from clients via HTML FORM and ISINDEX search interfaces. The chapter concludes with references to the IETF working draft documents on the HTTP protocol and the CGI specification, should you be interested in further details.

THE HTTP PROTOCOL

HTTP is an Internet client-server protocol designed for the rapid and efficient delivery of hypertext materials. HTTP is a stateless protocol, which means that a client can make several requests of a server but that each request is treated independently, with the server having no recollection of previous connections. This statelessness is an important feature if the server is to be fast.

All HTTP communication uses 8-bit characters, which ensures safe transmission of all forms of data, including HTML documents containing 8-bit ISO Latin-1 characters.

An HTTP connection has four stages:

1. Open the connection—The client contacts the server at the Internet address and port number specified in the URL (the default port is 80).
2. The request—The client sends a message to the server, requesting service. The request consists of HTTP *request headers* that define the *method* requested for the transaction and provide information about the capabilities of the client, followed by the data being sent to the server (if any). Typical HTTP methods are *GET*, for getting an object from a server, or *POST*, for posting data to an object on the server.
3. The response—The server sends a response to the client. This consists of *response headers* describing the state of the transaction (for example, the status of the response—successful or not—and the type of data being sent), followed by the actual data.
4. Close the connection—The connection is closed.

This procedure means that a connection can download only a single document or process a single transaction, while the stateless nature of the transaction means that each connection knows nothing about previous connections. The implications of this are illustrated in the following example HTTP transactions.

Single Transaction per Connection

Suppose HTTP is being used to access an HTML document containing ten inline images. Composing the entire document then requires 11 distinct connections to the HTTP server: one to retrieve the HTML document itself, and ten more to retrieve the ten image files.

Statelessness of the Connection

Suppose a user retrieves a fill-in HTML FORM from a server and enters his or her username and password information to access a restricted server-side resource. When the user submits the FORM data to the server, this

username/password information is sent to the server gateway program as part of the data contents (the URL of the gateway program is specified as an attribute to the FORM element, as discussed in Chapter 2).

This CGI program returns the results as an HTML document, along with a second FORM allowing further requests. But, this FORM doesn't contain a place for passwords. Since the server is stateless and has no memory of your first connection, how does it know who the user is the second time around? It does not. To keep track of this information, the CGI program must place the username/password information in an `<INPUT TYPE ="hidden">` element, within the returned FORM, as a record of the *state* of the client-server transaction. This information is then passed back and forth between the client and server, preserving knowledge of the username and password for each transaction.

EXAMPLE HTTP CLIENT-SERVER SESSIONS

The easiest way to understand the HTTP protocol is through simple examples. The following section documents five examples covering the most important HTTP methods. Each presentation shows both the data sent from the client to the HTTP server and the data returned from the server to the client. First a few words about how this *eavesdropping* was accomplished.

Monitoring Client-Server Interaction

It is not difficult to monitor the interaction between HTTP servers and World Wide Web clients, because the communication is entirely in character data sent to a particular port, so that all you need is the ability to *listen* at a port or talk to a port. You can listen at a port using the program **listen**, given in Appendix D, while you can talk to a port using the telnet program.

You use **listen** to find out what a Web client sends to a server. To obtain this information, you run the **listen** program on a computer; for example,

leonardo.subnet.ca. When you start **listen**, it prints out the port number it is listening at, for example:

```
listening at port 1743
```

and then falls silent, waiting to print any data that arrives at that port. You now configure a WWW client to send HTTP requests to this port. You do this by *pointing* the browser to port 1743 on ***leonardo.subnet.ca***, using a URL, such as:

```
http://leonardo.subnet.ca:1743/Tests/file.html
```

The client dutifully sends the HTTP request headers to port number 1743 on ***leonardo.subnet.ca***, where **listen** receives the data and prints it to the screen.

Determining what the server sends in response to the client takes a bit more work. In this case, you can use the telnet program to connect to the server and then must enter by hand the HTTP request headers that were sent by the client (and that you intercepted using the **listen** program, as just described). Suppose, for example, that the server is running at port 80 on ***leonardo.subnet.ca***. To connect to this server, simply make a telnet connection to this port. On a UNIX system, type (the PC or Macintosh versions are slightly different):

```
leonardo:> telnet leonardo.subnet.ca 80
Trying 128.100.121.33...
Connected to leonardo.subnet.ca
Escape character is '^]'
```

The string you type is in boldface. Telnet gives three lines of information to let you know what it is doing and then falls silent. Whatever you type is sent to the server running on ***leonardo.subnet.ca***, so you now type in the required request headers. Whatever the server sends in response is sent to the telnet program and printed on the screen.

These tools are used in the following five examples to determine the information passed between the client and server in typical HTTP transactions.

These five examples look at:

1. A GET method request for a file
2. A GET method request to a gateway program, with a query string appended to the URL
3. An HTML FORM accessing a server gateway program using the GET method
4. An HTML FORM accessing a server gateway program using the POST method
5. An HTTP HEAD method request of a file

These examples build on an understanding of URLs and of the way data are gathered by an HTML form. These topics were covered in Chapters 2 and 3.

EXAMPLE 1: A SIMPLE GET METHOD REQUEST

This request accesses the server and requests that a document be sent to the client. This could be initiated by clicking on a hypertext anchor pointing to the file. For example:

```
<A HREF="http://some.server.edu/Tests/file.html">
   anchor text
</A>
```

The transaction can be broken into two parts: the passing of the request to the server, and the response information sent by the server back to the client.

Passing the Request ■ Figure 4.1 shows the actual data sent by a Mosaic for X-Windows Version 2.4 client to the server. Other clients send qualitatively the same information. The dots indicate Accept headers that were omitted to save space.

```
GET /Tests/file.html HTTP/1.0
Accept: text/plain
Accept: application/x-html
Accept: application/html
Accept: text/x-html
Accept: text/html
Accept: audio/*
.
.
.
Accept: text/x-setext
Accept: */*
User-Agent: NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified
        [a blank line, containing only CRLF ]
```

Figure 4.1 Data sent from a client to an HTTP server during a simple GET request. *Comments are in italics.*

This request message consists of a *request header* containing several *request header fields*. Each field is a text line, terminated by a carriage-return linefeed character pair (CRLF). The blank line (containing only a CRLF pair) at the end indicates the end of the headers and the beginning of any data being sent from the client to the server. This example transaction does not send data to the server, so the blank line is the end of the request.

The request message contains two parts: the *method* field, which is the first line of the request and which specifies the HTTP method to be used and the location of the desired resource on the server, followed by HTTP *request* fields, which pass information to the server about the capabilities of the client.

The method field contains three text fields, separated by white space (white space is any combination of space or tab characters). In this example, the method header is:

```
GET /path/file.html HTTP/1.0
```

The general form of this field is:

```
HTTP_method  identifier  HTTP_version
```

These three strings contain:

`HTTP_Method` The HTTP method specification—GET in the preceding example. The method specifies what is to be done to the object specified by the URL. Some other common methods are HEAD, which requests header information about an object, and POST, which is used to send information to the object.

`identifier` The identifier of the resource. Here, this is the URL stripped of the protocol and Internet domain name strings. If this were a message to a *proxy server*, this would be the entire URL. Proxy access is discussed later in this chapter.

`HTTP_Version` The HTTP protocol version used by the client, currently HTTP/1.0.

The example shows several additional request headers. The Accept field passes to the server a list of data representation schemes acceptable to the client. These are given as MIME content-types and simply tell the server what type of data the client can handle. MIME types are discussed in more detail in Appendix B. The meanings are relatively straightforward. For example, the line `Accept: text/plain` means that the client can accept plain text files, while the line `Accept: audio/*` means that the client can accept any form of audio data.

There are several other possible request fields. The only other one used in the preceding example is the `User-agent` header:

```
User-Agent: NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified
```

This provides information about the client making the request. The string `libwww/2.12` refers to the CERN common-code library for WWW applications, which is a software library used in developing WWW

software. Other possible request headers are summarized at the end of this section.

Server Response ■ When the server receives the request, it tries to apply the designated method to the specified object (file or program) and passes the results of this effort back to the client. The returned data are preceded by a response header made up of *response header fields*, which communicate information about the state of the transaction back to the client. As with the request header fields sent from the client to the server, these are single lines of text terminated by a CRLF. The end of the response header is indicated by a single blank line containing only a CRLF. The data of the response comes after the blank line.

Figure 4.2 shows the data sent from the server to the client in response to the request of Figure 4.1.

The first seven lines are the response headers. The end of the response headers is indicated by the single blank line. The data response of the request (in this case, the requested HTML document) follows this blank line.

The first line in the response header is a status line that lets the client know what protocol the server uses and whether the request was successfully completed. In our example, this line is:

```
HTTP/1.0 200 OK
```

The general format for this line is:

```
http_version status_code explanation
```

The `HTTP_version` field gives the protocol version used by the server, currently HTTP/1.0. The `status_code` is a number between 200 and 599 that gives the status of the connection, while the `explanation` field is a text string that provides more descriptive information about the status. Explanation strings may vary from server to server, whereas status codes are explicitly defined by the HTTP specifications. Status codes between 200

and 299 indicate successful transactions. Numbers 300 to 399 indicate redirection, which means that the object specified by the URL has moved and that the server is also sending the client the new URL of the object (the server sends the new URL in a `Location` header). Numbers 400 to 599 are error messages. When you encounter an error, the server usually sends a small HTML document to explain the error. Figure 4.3 gives more details about the different status codes.

```
HTTP/1.0 200 OK
Date: Friday, 23-Sep-94 16:04:09 GMT
Server: NCSA/1.3
MIME-version: 1.0
Content-type: text/html
Last-modified: Friday, 23-Sep-94 16:03:27 GMT
Content-length: 145
     [a blank line, containing only CRLF ]
<html>
<head>
<title> Test HTML file </title>
</head>

<body>

<h1> This is a test file</h1>
<p> So what did you expect, art?

</body>
</html>
```

Figure 4.2 Data returned from the server to the client subsequent to the GET request of Figure 4.1. *Comments are in italics.*

The remaining response headers contain information about the server and about the response being sent. The example in Figure 4.2 returned six lines of response header information. The first three, the `Date`, `Server`, and `MIME-version` fields, describe the server and the details of the transaction, while the `Content-type`, `Last-modified`, and `Content-length` fields pass information specific to the document being returned. The formats and meanings of these header fields are:

`Date: date_time`—Contains the time and date when the current object was assembled for transmission, in the format `Friday, 23-Sep-94 16:04:09 GMT`. Note that the time must be Greenwich Mean Time (GMT) to ensure that all servers share a common time zone.

`MIME-version: version_number`—Gives the MIME protocol version. The current version is 1.0.

`Server: name/version`—This gives the name and version of the server, with a slash character separating the server name from the version number; for example, `NCSA/1.3` or `CERN/3.0pre6`.

`Content-length: length`—Gives the length in bytes of the data portion of the message (in this case, `145`). In some cases, the length is unknown (for example, if it is output from a gateway program), in which case, this field is absent.

`Content-type: type/subtype`—Gives the MIME `Content-type` of the data being sent. This tells the client what type of data is being sent by the server. In this example, the MIME type is `text/html`, which informs the browser that the data is an HTML document.

`Last-modified: date_time`—Gives the date and time that the document was last modified, in the format `Friday, 23-Sep-94 16:03:27 GMT`. As with the `Date` field, the date must be given in Greenwich Mean Time.

How are these headers generated? If the request is for a file, the HTTP server constructs these headers itself. If the request is to a CGI gateway

program, then the gateway program must provide headers that specify details about the returned data, such as the `Content-type`, that cannot be otherwise determined by the server. The server parses these headers and adds some of its own to construct a complete collection of server response headers. Alternatively, the gateway program can return all the response headers and bypass the server. This *nonparsed header* option is described in the discussion of the CGI specification.

A complete list of commonly used HTTP response headers is given at the conclusion of the HTTP protocol section.

A) Successful Transactions

200 The request was fulfilled.

201 The request was a POST method and was completed successfully.

202 The request has been accepted for processing, but the results of this processing are unknown. This would be returned, for example, if the client deposited data for batch processing at a later date.

203 The request was fulfilled but has returned partial information.

B) Redirection Transactions

301 The data requested has been permanently moved to a new URL. If this status is returned, the server should also send the client the URL of the new location via the header

```
Location: URL comments
```

where `URL` is the new document URL. Browsers that understand the `Location` field will automatically connect to the new URL.

302 The data was found but it actually resides at a different URL. If this status is returned, the server should also send the client the correct URL via the header

```
Location: URL comments
```

	Browsers that understand the `Location` field will automatically connect to the new URL. You will get a 302 Redirection if a URL pointing to a directory is missing the trailing slash character.
304	A GET request was sent containing the `If-Modified-Since` field. However, the server found that the document had not been modified since the date specified in this field. Consequently, the server responds with this code and does not resend the document.

C) Error Messages

400	The request syntax was wrong.
401	The request required an `Authorization:` field, and the client did not specify one. The server also returns a list of the allowed authorization schemes using `WWW-Authenticate` response headers. This mechanism is used by a client and server to negotiate data encryption and user authentication schemes.
402	The requested operation costs money, and the client did not specify a valid `Chargeto` field in the request header.
403	You have requested a resource that is forbidden.
404	The server cannot find the URL you requested.
500	The server has encountered an internal error and cannot continue with the request.
501	The request you have made is legal, but the server does not support this method.

Figure 4.3 HTTP status codes.

LESSONS FROM EXAMPLE 1

1. When a client contacts an HTTP server, it sends the server a collection of *request headers*. These headers include the *method field*, which specifies the HTTP method being requested by the client and the *location* on the server of the resource being requested, followed by other request header fields that pass information about the capabilities of

the client. The collection of request headers is terminated by a single blank line.

2. The server responds with a message consisting of a *response header* followed by the requested data. The response header fields communicate information about the state of the transaction, including a MIME *content-type* header that explicitly tells the client the type of data being sent. The collection of response headers is terminated by a single blank line. The data being sent to the client follows after the blank line.

Example 2: A GET Method Request with Search Strings

The second example is a GET method, but with query information appended to the URL. As discussed in Chapter 2, query information is appended to the URL, separated by a question mark. This is done automatically by World Wide Web browsers when you submit ISINDEX queries or HTML FORMs data using the GET method.

A typical URL might be:

```
http://www.stuff.ca/cgi-bin/srch-example?item1+item2+item3+item4
```

which passes four items to the program ***srch-example***.

Figure 4.4 shows the request headers sent by the **MacWeb** browser when it accesses this URL. Again, some accept headers are omitted to save space.

This GET method is also known as the HTTP *TEXTSEARCH* method, even though it is constructed using GET and a modified URL. This request header is essentially the same as in Figure 4.1, the major difference being the query strings added to the locator string. The server's interpretation of this string is described in the second half of this chapter.

```
GET /cgi-bin/srch-example?item1+item2+item3+item4 HTTP/1.0
Accept: video/quicktime
Accept: video/mpeg
Accept: image/x-xbitmap
Accept: image/pict
Accept: image/jpeg
.
.
.
Accept: text/plain
Accept: text/html
User-Agent: MacWeb/1.00ALPHA2 libwww/2.13
   [a blank line, containing only CRLF ]
```

Figure 4.4 Data sent from a **MacWeb** client to an HTTP server during a GET request, with query strings appended to the URL. *Comments are in italics.*

Lesson from Example 2

Query data appended to a URL during a GET request to a server is passed as part of the locator string in the HTTP *method field* of the request headers. All other request headers are the same as those for a standard GET request, as described in Example 1. A GET request with an appended query string is also called the HTTP TEXTSEARCH method.

Example 3: Submitting a FORM Using the GET Method

This example examines how an HTML FORM submits the form data contents to a server, using the HTTP GET method. The example HTML FORM is shown in Figure 4.5. The actual rendering of this form by a WWW browser is shown in Figure 4.6.

```
<FORM ACTION="http://leonardo.utirc.utoronto.ca:8080/cgi-bin/form1" METHOD=GET>
<p> Search string: <INPUT TYPE="text" NAME="srch" VALUE="dogfish">
<p> Search Type:
  <SELECT NAME="srch_type">
    <OPTION> Insensitive Substring
    <OPTION SELECTED> Exact Match
    <OPTION> Sensitive Substring
    <OPTION> Regular Expression
  </SELECT>
<p> Search databases in:
  <INPUT TYPE="checkbox" NAME="srvr" VALUE="Canada" CHECKED> Canada
  <INPUT TYPE="checkbox" NAME="srvr" VALUE="Russia"  > Russia
  <INPUT TYPE="checkbox" NAME="srvr" VALUE="Sweden" CHECKED> Sweden
  <INPUT TYPE="checkbox" NAME="srvr" VALUE="U.S.A."  > U.S.A.
  <em>(multiple items can be selected.)</em>
<P> <INPUT TYPE="submit"> <INPUT TYPE=reset>.
</FORM>
```

Figure 4.5 Example HTML FORM that uses the GET method to submit data to a server.

The FORM element is discussed in detail in Chapter 2. This FORM defines the three variable names, *srch*, *srch_type*, and *srvr*. These have been assigned values *srch=dogfish*, *srch_type=Exact Match*, *srvr=Canada*, and *srvr=Sweden*. **Mosaic for X-Windows** Version 2.4 is again used, as in Example 1. Figure 4.7 shows the data sent by this FORM to the server. The dots indicate accept headers that were deleted to save space.

As discussed in Chapter 2, submitting a FORM sends the data to the server as a collection of *name/value* pairs. In practice, this is accomplished by sending the data as a collection of strings, `name=value`, with the ampersand character (&) separating the strings. In addition, blank characters are encoded into plus signs. With the GET method, this string is appended to



Figure 4.6 Mosaic for X-Windows browser rendering of the FORM example in Figure 4.5.

the URL and separated from it by a question mark. When the HTTP server receives these data, it forwards the entire *query string* to the gateway program ***form1***. These details are discussed in Example 9 in the CGI section of this chapter.

LESSON FROM EXAMPLE 3

When an HTML FORM submits data to an HTTP server using the GET method, the FORM data are appended to the URL as a query string. Consequently, the FORM data are sent to the server in the query string part of the locator string in the request header method field. The FORM data are encoded according to the URL syntax.

```
GET /cgi-bin/form1?srch=dogfish&srch_type=Exact+Match&srvr=Canada&srvr=Sweden HTTP/1.0
Accept: text/plain
Accept: application/x-html
Accept: application/html
Accept: text/x-html
Accept: text/html
Accept: audio/*
.
.
.
Accept: text/x-setext
Accept: */*
User-Agent: NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified
    [a blank line, containing only CRLF ]
```

Figure 4.7 Data sent from a client to an HTTP server during a FORM-based GET request.

EXAMPLE 4: SUBMITTING A FORM USING THE POST METHOD

This example again uses the FORM shown in Figure 4.6 but with one subtle change: It uses the HTTP POST method, not GET. Therefore, this example sends the same data to the server but by a different method.

As in Example 3, this form has variable names *srch*, *srch_type*, and *srvr*. These have been assigned values *srch=dogfish*, *srch_type=Exact Match*, *srvr=Canada*, and *srvr=Sweden*. Figure 4.8 shows the data sent to the server, again from the **Mosaic for X-Windows** browser.

The POST method sends data in a message *body*, not in the URL. The difference is indicated in three places. The first is in the method header, which now specifies the POST method and which has no data appended to the URL. The second is in the two new request header fields: `Content-type` and `Content-length`. Last, it is indicated by the line of data following the headers. This line is the data being sent to the server via the POST method.

```
POST /cgi-bin/form1 HTTP/1.0
Accept: text/plain
Accept: application/x-html
Accept: application/html
Accept: text/x-html
Accept: text/html
Accept: audio/*
 .
 .
 .
Accept: text/x-setext
Accept: */*
User-Agent: NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified
Content-type: application/x-www-form-urlencoded
Content-length: 58

srch=dogfish&srch_type=Exact+Match&srvr=Canada&srvr=Sweden
```

Figure 4.8 Data sent from a client to an HTTP server during a FORM-based POST request.

The two new headers tell the server that there will be data following the request headers. The content-length header gives the length in bytes of the message, while the content-type header specifies the MIME-type of the message being sent to the server. There is currently only one valid content-type for FORM data being sent to a server, namely:

```
Content-type: application/x-www-form-urlencoded
```

This header indicates that the data is from a FORM and that it is encoded in the same manner as when appended to a URL. You can compare the message body at the bottom of Figure 4.8 with the data appended to the URL in Figure 4.7 to verify this equivalence. The details of how the data are passed from the server to the CGI program are described in the second half of this chapter.

Lesson from Example 4

When an HTML FORM submits data to an HTTP server using the POST method, the FORM data are sent to the server as a *message body* that follows the request headers. This message body is encoded in the same manner as when appended to the URL. Additional request headers tell the server the content-type of the arriving message (`application/x-www-form-urlencoded`) and the length of the message. No data are appended to the URL.

Example 5: Accessing a Document with the HEAD Method

The HEAD method is often used by programs that automatically search for documents on WWW servers. This method requests that the server send the response headers relevant to the requested URL, but not the referenced object. This is a quick way of seeing if a document or gateway program is actually present and of obtaining some general information about it, such as its MIME content-type, without downloading an entire document.

For example, suppose you access the document in Example 1, but using the HEAD method. This request is written:

```
HEAD /Tests/file.html HTTP/1.0
User-Agent: HEAD Test Agent
From: name@domain.name.edu
```

You do not need all the accept headers, since you are not requesting a complete document. The new `From` field gives the server the electronic mail address of the person making the HEAD request. The response to this request (in this case, the NCSA Version 1.3 HTTP server) is:

```
HTTP/1.0 200 OK
Date: Saturday, 24-Sep-94 23:10:55 GMT
Server: NCSA/1.3
MIME-version: 1.0
```

```
Content-type: text/html
Last-modified: Sunday, 18-Sep-94 19:44:28 GMT
Content-length: 0
```

This indicates that the document is there (the status code 200 implies a valid URL), and includes information about the document type and the date it was last modified. In principle, the HTML **META** allows the server to parse the document and include additional information about the document in these header fields, but few servers currently implement this feature.

LESSON FROM EXAMPLE 5

A HEAD request retrieves the response headers for the indicated URL only—the document itself is not retrieved. If the HEAD request is for an HTML document, some servers parse the *document* **HEAD** for information to include in the response headers—this server feature is not widely implemented.

The preceding discussion and examples provided an overview of the HTTP protocol. You can, if you wish, skip to the CGI specifications section of this chapter. The next section looks briefly at more complex features of the HTTP protocol, such as user authentication, format negotiation, and other less used methods, and gives a summary of commonly used request headers, response headers, and HTTP methods.

USER AUTHENTICATION, DATA ENCRYPTION, AND ACCESS CONTROL

So far, we have not discussed controlling access to an HTTP server, or the encryption of data for transmission between client and server. There are several mechanisms available for doing these things, but as yet no well-established standard. This section briefly discusses the different options open to a server administrator, and refers the interested reader to more advanced documentation on this topic.

For low-level security, you can configure the server to allow access only to machines in an authorized Internet domain. For example, several servers can be configured to allow access only to machines with Internet domain names ending with a string such as .domain.site.edu. This feature is available on several servers. Often, access can be restricted on a per-file or per-directory basis. This is not very secure, however, since a clever cracker can *spoof* a domain name and access the data. It is also not very specific, since anyone can access the restricted material, provided they can access one of the allowed machines.

A somewhat finer control is possible when accessing server-side CGI programs. For example, you can connect an HTML FORM to a server-side program and have the FORM request identification and password information for authentication on the server. This is still not very secure, since the information being sent is not encrypted so that anyone intercepting the messages can read the username and password information.

Truly secure communication requires negotiated encryption of the data being sent from client to server and from server to client. Negotiated encryption means that the server and client must negotiate the scheme to be used and must also exchange information allowing each of them to send encrypted information that can be decoded by the receiver. This requires that both client and server contain encryption/decryption software and that they both understand the same encryption schemes. Currently, the NCSA and CERN HTTP servers have these capabilities, as do certain client programs, such as **Mosaic for X-Windows**. The references at the end of this chapter describe the details of the CERN and NCSA authentication/encryption implementations. Authentication is an important feature for most commercial providers of WWW software products, so, if encryption is important, you may want to consider commercial providers of server and browser software.

Proxy Servers and Server Caching

Proxy servers are used by local area networks (LANs) that want to protect themselves from unauthorized entry over the Internet. This can be done by installing *firewall* software on the gateway machine that links the local network to the outside world. A firewall keeps TCP/IP packets from entering the local network from the outside world and thereby protects the LAN from the dangerous world outside.

Unfortunately, a firewall means that users inside the LAN cannot access WWW resources outside, since communication is blocked by the firewall. The solution is to install a *proxy server* on the firewall, and to configure your WWW browsers to refer to this proxy server instead of the real servers outside your LAN. A proxy server has access to both the inside and outside worlds, and can safely pass information back and forth across the firewall. When your browser wants to request information from outside the firewall, it sends this request to the proxy server on the firewall. The proxy server then *proxies* the request—that is, it completes the request on your behalf and forwards to you the result.

This two-stage process can be slow. To speed up service, proxy servers can *cache* the retrieved files. Thus, the first time you request a document, the server goes to the outside world and fetches it, but subsequently retains a copy on its own local disk. If you make another request for this same file, the proxy server gives you the locally cached copy of the document and does not refetch it. This saves time and can also significantly reduce the load on the network connection.

This can be a problem, however, if the file is one that changes with time. This can be the case for pages containing periodically updated data or for documents created dynamically by CGI programs. This problem can be solved using the `Expires` server response header, which is used to inform a proxy server how long it can keep a cached copy of a downloaded document. When the proxy server retrieves a document from the primary server

outside the firewall, the primary server can place in the server response headers the line:

```
Expires: date
```

This tells the proxy server how long it can keep the document before it must be updated from the original server.

CERN HTTP Proxy Server

There are two ways of implementing proxy servers on a LAN. The CERN HTTP proxy server was explicitly designed to be an HTTP server, and also to support proxy access outside a firewall-protected network. The CERN HTTP server also implements document caching, and is the best choice for implementing HTTP service in a secure environment. More details about the CERN HTTP server are given in Chapter 8.

SOCKS Proxy Support

The SOCKS package is another proxying package, but was not designed specifically for HTTP service and does not have caching capabilities. However, this is an ideal choice if you do not also need an HTTP server, and is significantly easier to implement than the full CERN HTTP server. Additional information about SOCKS can be found at:

```
ftp://ftp.nec.com/pub/security/socks.cstc
```

A proxy server is useful only if your browser can be configured to use it. Chapter 7, which discusses the different WWW browsers, notes browsers that support proxy service. Additional information about proxy support is found in the References at the end of this chapter.

HTTP METHODS AND HEADERS REFERENCE

The following three sections summarize the currently implemented HTTP methods, request headers, and response headers. A more detailed list can be found in the References at the end of this chapter.

HTTP METHODS

The most common methods are the GET/TEXTSEARCH, HEAD, and POST methods just discussed. The following is a slightly more complete list of these methods:

GET	Retrieve the indicated URL.
HEAD	Retrieve the HTTP header information for the indicated URL.
TEXTSEARCH	Access the indicated URL and query the URL. This is done using a GET method and a URL that contains appended query data.
LINK	This method links an existing object to another object. For an HTML document, this would imply editing the document and adding LINK information to the document HEAD.
UNLINK	Remove the link information inserted, for example, by a LINK method.
POST	Send the data to the indicated URL. The URL must already exist.
PUT	Place the data being sent by the client in the indicated URL, replacing the old contents. The URL must already exist. The CERN HTTP server has an implementation of PUT.

HTTP REQUEST HEADERS

`Accept: type/subtype`—This field contains a list of MIME content-types that are acceptable to the client, separated by semicolons.

`Accept-Encoding: encoding_type`—Lists the data encoding types acceptable to the client. For example, `Accept-Encoding: x-compress` tells the server that the client can accept compressed data in the compress format.

The only current encoding types are `x-compress` and `x-gzip`. If a server sends a document in compressed format, it must include the `Content-Encoding` response header.

`Authorization: scheme data`—This passes user authentication and encryption scheme information to the server.

`Content-length: length`—Gives the length (in bytes) of the message being sent to the server.

`Content-type: type/subtype`—This gives the MIME content-type of the message being sent to the server.

`From: mail_address`—This contains the address, in Internet mail format, of the user accessing the server.

`If-Modified-Since: date`—This request header is used with the GET method to make the GET request conditional: If the requested document has not changed since the indicated time and date, the document is not sent. The date must be in Greenwich Mean Time and of the format `Friday, 23-Sep-94 18:28:33 GMT`. If the document is not sent, the server should send the response header message 304 (not modified).

`Pragma: directives_for_server`—Pragma directives are designed for passing special-purpose information to servers. Currently, there is only one server directive, `Pragma: no-cache`, which instructs a proxy server to always fetch the document from the actual server, and to never use a locally cached copy.

`Referer: URL`—This gives the URL of the document from which the request originated. This can be a partial URL, in which case, it is interpreted relative to the URL of the document being requested. If a document contains an HTML **BASE** element, then the contents of this should be sent instead.

`User-Agent: program/version comments`—Provides information about the client software making the request.

HTTP RESPONSE HEADERS

The following is a list of currently implemented HTTP response headers. A more detailed list can be found in the References listed at the end of this chapter.

`Content-Encoding: encoding_type`—Specifies the encoding type mechanism used in generating the results. The only currently valid types are `x-compress` and `x-gzip`. You can have only one type per header. This allows compressed files to be uncompressed on-the-fly by the client.

`Content-length: xxx`—Gives the length (in bytes) of the message being sent to the client.

`Content-Transfer-Encoding: encoding_type`—This is a MIME protocol header and gives the encoding mechanism used for the MIME messages. The default is 8-bit character encoding.

`Content-type: type/subtype`—This gives the MIME content-type of the message being sent to the client.

`Expires: date_time`—Gives the time and date after which the information being sent should be considered invalid. This can allow clients to automatically refresh data that should be periodically updated. Proxy servers use this field to determine when a cached copy of a document should be refreshed.

`Title: title`—The title of the document. This should be identical to the contents of the document's **TITLE** element.

`WWW-Authenticate: scheme scheme_message`—This passes information to the client stating the encryption and authorization schemes the server wants to use. This is used only for directories, files, or CGI programs that require user authentication. `Scheme` gives the name of the authorization scheme, while `scheme_message` gives related data.

`WWW-Link:`—This should contain the information from the **LINK** elements of an HTML document. This allows the HTTP header to contain information

about an HTML document, and would allow clients using HEAD methods to access information about the document useful for cataloging and indexing purposes.

THE COMMON GATEWAY INTERFACE

The Common Gateway Interface (CGI) is the specified standard for communication between HTTP servers and server-side gateway programs. When you access a gateway program, the server activates the program and passes it any ISINDEX query, FORM, or other data that was sent from the client. When the gateway program has finished processing the data, it sends the result back to the server and, from there, back to the client. The CGI specifications define how these data are passed from the server to the gateway program, and vice versa.

Gateway programs can be compiled programs written in languages such as C, C++, or Pascal, or they can be executable scripts written in scripting languages such as perl, tcl, and the various shell programs. In fact, most gateway programs are scripts, since these are easy to write and modify and are inherently transportable from machine to machine. In addition, execution speed is often not an important factor when writing a gateway program, since the slowest part is usually the resource the gateway connects to, and not the gateway program itself.

SENDING DATA FROM THE CLIENT TO THE SERVER

There are three ways data is sent from the client to the server. The first is via a URL query string, such as:

```
http://some.site.edu/cgi-bin/ex_prog?query_info
```

This passes the query string `query_info` to the server. The server, in turn, launches the gateway program ***ex_prog*** and passes it the query string.

The second way is as *extra path* information, which is placed in the URL by adding directory-like information to the URL just after the name of the gateway program. For example:

```
http://some.site.edu/cgi-bin/ex_prog/dir/file?query_info
```

The string `/dir/file` is interpreted by the server as extra path information, while the string `query_info` is once again interpreted as a query string. When the server launches the gateway program ***ex_prog***, it passes to ***ex_prog*** both the query string `query_info` and the extra path string `/dir/file`.

Third, data can be sent to the server in a message body. This is possible with the HTTP POST method and is often used with HTML FORMs. When a server receives a POST method message from a FORM, it sends the POSTed data to the designated gateway program.

SENDING DATA TO THE GATEWAY PROGRAM FROM THE SERVER

The CGI specifications define the mechanisms by which data are forwarded to a gateway program. There are three mechanisms:

1. Command-Line Arguments—The server launches the gateway program with command-line arguments.
2. Standard Input—The server passes data to the gateway program such that it is read as input (from standard input) by the gateway program client.
3. Environment Variables—The server puts information in special *environment variables* before starting the gateway program. The gateway program can then access these variables and obtain their contents.

The mechanisms that are used during a particular transaction depend on the HTTP method of the request (GET or POST), and also on the nature of the query string appended to the URL (ISINDEX versus non-ISINDEX queries). Later sections use examples to illustrate typical cases.

RETURNING DATA FROM THE GATEWAY PROGRAM TO THE SERVER

There are two ways in which a CGI program can send information back to the server:

1. Write to standard output—The gateway program passes data back to the server by writing data to standard output. This is the *only* way that gateway programs can return data to a client.

2. The name of the gateway program—Gateway programs with names beginning with the string ***nph-*** are called *nonparsed header* programs and are treated specially by the server. In general, the server parses the output of a gateway program looking for headers that it can use to create the HTTP response headers it will send to the client with the returned data. If a gateway program name begins with ***nph-***, the server sends the gateway program directly to the client and does not add any header information.

These methods are illustrated with several examples. The first of these, Example 6, looks at an HTML ISINDEX document request. Example 7 demonstrates nonparsed header gateway programs, which send data directly back to the client, bypassing any server processing. Example 8 shows how environment variables are passed to the gateway program and explains the contents of these variables. Examples 9 and 10 show how HTML FORMs data—using the GET and POST methods, respectively—are passed to a gateway program.

EXAMPLE 6: ISINDEX SEARCHES

ISINDEX queries is the *only* query method that passes data to a gateway program as command-line arguments. It is a simple technique and a useful starting point for understanding client-server-gateway interactions.

This example accesses the gateway program ***srch-example*** listed in Figure 4.9. This is a Bourne-shell script designed to search a phone-number database using the search program ***grep***. This script searches for names in a phone-listings database and uses the **ISINDEX** element to prompt for the search string. In this example, the search string is just the list of names you want to search for. When this script receives this data, it searches the database for the indicated names and returns the names and phone numbers of any matches. The script is designed both to prompt for search strings and to return the results of the search.

From Chapter 2, you will recall that ISINDEX sends data to the server by appending the query data onto the URL being viewed; for example:

```
http://some.where.edu/cgi-bin/srch_program?query_string
```

When this information reaches the server, the URL is decoded and, if it is an ISINDEX query, the server takes the query strings and passes them to the indicated gateway program as command-line arguments.

How does the server know if a query is an ISINDEX query? The convention is that an ISINDEX query string *does not contain* any unencoded equal signs (=). As noted in Example 3, FORM data is passed as a collection of `name=value` strings, which always contains equal signs. Therefore, the presence of an equal sign indicates that the query does not come from an ISINDEX query, in which case, the server does not convert the query data into command-line arguments.

Step 1: First Access of the URL ■ In the example, we first access the script ***srch-example*** using the URL:

```
http://leonardo.utirc.utoronto.ca:8080/cgi-bin/srch-example
```

Note that there is no query information attached to this URL; this is an important factor in the behavior of the script.

```
01 #!/bin/sh
02 echo Content-TYPE:  text/html
03 echo
04
05 if [ $# = 0 ]              # is the number of arguments == 0 ?
06 then                       # do this part if there are NO arguments
07  echo "<HEAD>"
08  echo "<TITLE>Local Phonebook Search</TITLE>"
09  echo "<ISINDEX>"
10  echo "</HEAD>"
11  echo "<BODY>"
12  echo "<H1>Local Phonebook Search</H1>"
13  echo "Enter your search in the search field.<P>"
14  echo "This is a case-insensitive substring search: thus"
15  echo "searching for 'ian' will find 'Ian' and Adriana'."
16  echo "</BODY>"
17 else                         # this part if there ARE arguments
18  echo "<HEAD>"
19  echo "<TITLE>Result of search for \"$*\".</TITLE>"
20  echo "</HEAD>"
21  echo "<BODY>"
22  echo "<H1>Result of search for \"$*\".</H1>"
23  echo "<PRE>"
24   for i in $*
25   do
26      grep -i $i /vast/igraham/Personnel
27  done
28  echo "</PRE>"
39  echo "</BODY>"
40 fi
```

Figure 4.9 Bourne-shell script CGI gateway program ***srch-example***.

Line 1 tells the computer to interpret this script using the **/bin/sh** program, which is the traditional location and name for the Bourne shell. The second two lines *echo* information to standard output (`echo` is the Bourne-shell command for printing to standard output). Standard output information is sent back to the server and, from there, back to the client. These two lines provide HTTP response header information about the data to come. This is necessary, as the server has no other way of knowing the type of data the gateway program will send. The first line in this script prints the header:

```
Content-type: text/html
```

to tell the client and server that the data to follow is an HTML document. The next line of the script prints a blank line. Per the HTTP protocol, this indicates the end of the header fields and that subsequent output is message data.

Line 5 tests the number of command-line arguments. In this case, there is no query string, so there are no arguments and the first branch of the `if` is executed. This first branch prints to standard output a simple HTML document explaining the nature of the search; this is shown in Figure 4.10. The document also contains the **ISINDEX** element, which instructs the browser to prompt for search information; this results in the query box shown in Figure 4.10. The names *ian* and *bradley* are already typed into this box, separated by a single space. These are the names to be used in the search.

Step 2: Second Access of the URL ■ Submitting this ISINDEX search information accesses the same URL, but appends the names *ian* and *bradley* to the URL as query strings. Thus, in this second phase, the accessed URL is:

```
http://leonardo.utirc.utoronto.ca:8080/cgi-bin/srch-example?ian+bradley
```

where the space between *ian* and *bradley* has been encoded as a plus sign.

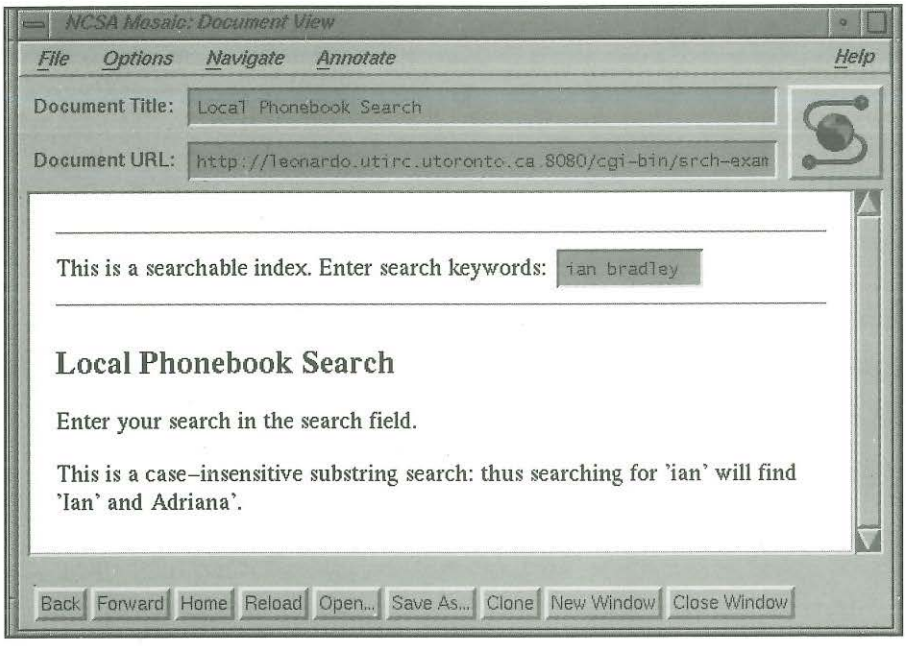


Figure 4.10 Document returned from the script ***srch-example*** when accessed *without* a query string appended to the URL.

When the server receives this URL, it parses the query string and finds that there are no unencoded equal signs, so it knows that this is an ISINDEX query. It therefore takes the query string and breaks it into individual strings, using the plus signs to mark the string separators. This yields two strings: `ian` and `bradley`. The server next launches the gateway program ***srch-example***, using the names `ian` and `bradley` as command-line arguments. If you were to type the command yourself, this would look like:

```
srch-example ian bradley
```

Figure 4.11 shows the results of this second access to the Bourne-shell program and, by following Figure 4.9, you can see how this was generated. As before, the first two lines print the MIME content-type of the message and the blank line separating the HTTP headers from the data. At line 5, the program checks for command-line arguments. This time, there are arguments, so the second branch of the script is executed starting at line 18.

This section prints a different HTML document: this time, including output from the program *grep*. Lines 24 through 27 loop the variable `i` through all the different command-line arguments. The contents of the variable `i` (denoted by `$i`) are given as an argument to the program *grep*, which scans the file */vast/igraham/Personnel* for names matching the pattern given by the argument `i`. *Grep* prints the matches to standard output. The result of the searches is shown in Figure 4.11. Note that there is now no query box. The second branch through the script in Figure 4.9 did not include an ISINDEX element, so no query box is presented.

EXAMPLE 7: HEADERS AND GATEWAY PROGRAMS

In this example, we look at how the HTTP server adds header information to the document returned by the gateway program. Consider first (see Figure 4.12) the actual data sent by the server to the client when it first accessed the *srch-example* (these are the data that produced Figure 4.10).

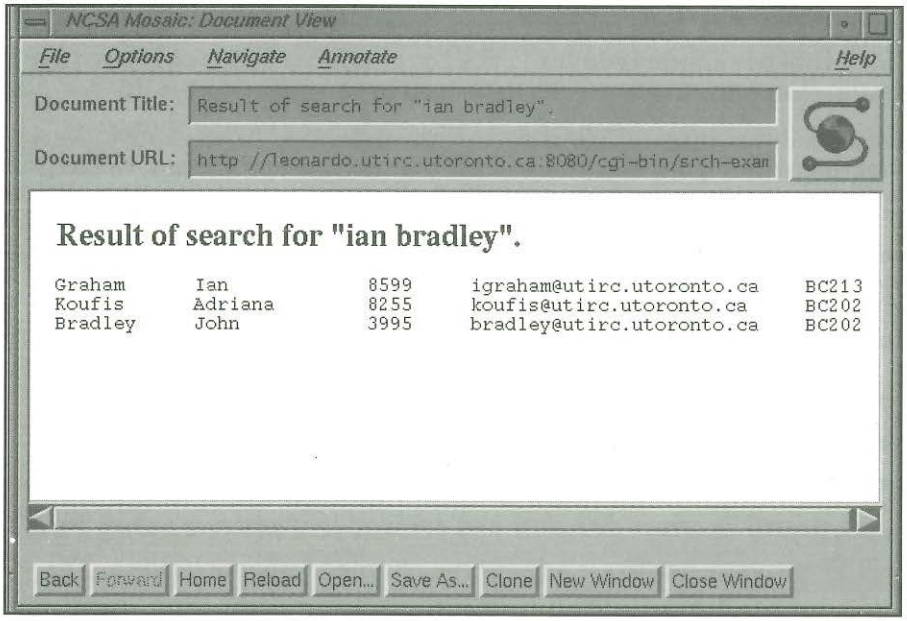


Figure 4.11 Document returned from the script ***srch-example*** when accessed *with* a query string appended to the URL.

```
HTTP/1.0 200 OK
Date: Tuesday, 27-Sep-94 16:43:48 GMT
Server: NCSA/1.3 MIME-version: 1.0
Content-type: text/html

<HEAD>
<TITLE>Local Phonebook Search</TITLE>
<ISINDEX>
</HEAD>
<BODY>
<H1>Local Phonebook Search</H1>
Enter your search in the search field.<P>
This is a case-insensitive substring search:thus
searching for 'ian' will find 'Ian' and Adriana'.
</BODY>
```

Figure 4.12 Data returned to the client upon accessing the URL `http://leonardo.utirc.utoronto.ca:8080/cgi-bin/srch-example`. These are the data used to generate Figure 4.10.

Comparing with Figure 4.9, you will see that the headers are not, in fact, those returned by the script: The content-type headers are typographically different. In fact, the headers returned to the client were generated by the HTTP server, with help from the headers passed from the gateway program.

The server takes the header data returned by the gateway program and parses each of the header fields. Most of these headers are passed through unaltered and are included with headers returned to the client. Some, however, are treated as *server directives* and are used by the server to *modify* the HTTP response headers it adds to the returned response headers. The three currently valid server-directive headers are:

1. `Content-type: type/subtype`—This gives the MIME-type for the data being sent by the gateway program. The server replaces its default Content-type server response header with the type/subtype values given here.

2. `Location: URL`—This tells the server that the gateway program is specifying the URL to which the client should be redirected. The server adds this header to the server response and also changes its header status line from:

   ```
   HTTP/1.0 200 OK
   ```

   to:

   ```
   HTTP/1.0 302 Redirection
   ```

   which tells the client that there is redirection information in the response headers, and that it should use the `Location` field to access the desired document.

3. `Status: code string`—This passes an HTTP status string to the server for use in place of the standard value. For example, if the ***srch-example*** gateway script returned the header:

   ```
   Status: 444 four-fourty-for
   ```

   as a result, the response headers returned to the client would look like:

   ```
   HTTP/1.0 444 four-fourty-for
   Date: Tuesday, 27-Sep-94 16:43:48 GMT
   Server: NCSA/1.3
   MIME-version: 1.0
   Content-type: text/html
   ...
   ```

NONPARSED HEADERS

It is possible to return gateway program output directly to the client without it being parsed by the HTTP server. This is done by appending the string ***nph-***, for *nonparsed headers*, to the name of the script. When the

server sees gateway program names beginning with ***nph-***, it knows to pass the gateway program output directly to the client. For example, Figure 4.13 shows the data returned from the gateway program ***nph-srch-example***; this is an exact duplicate of the program ***srch-example***, the only change being the string ***nph-*** added to the program filename.

Comparing Figure 4.13 with Figure 4.9 shows that the response now contains just the data printed by the gateway program, with nothing added or changed by the server. The advantage of nonparsed header gateway programs is speed, since the server is not required to parse the returned data and generate appropriate headers. In exchange, the gateway program itself must produce all the required headers. Thus, an ***nph-*** script must also print, at a minimum, the following response headers (with values appropriate to the script):

```
HTTP/1.0 200 OK
Date: Tuesday, 27-Sep-94 16:43:48 GMT
Server: NCSA/1.3
MIME-version: 1.0
```

EXAMPLE 8: ENVIRONMENT VARIABLES

The preceding examples would imply that the server passes very little information to the gateway programs. In fact, the server is not so ungenerous. Before launching the gateway program, the server initializes several *environment variables* that are subsequently accessible to the gateway programs. In particular, this mechanism is used to pass extra path information to the gateway program. The names and contents of the environment variables are shown in Figures 4.14 and 4.15. Figure 4.14 shows the gateway script ***srch-example-2***; this is the same ISINDEX script shown in Figure 4.9, modified to print the contents of the environment variables. The HTML document generated upon accessing this script at the URL:

```
http://leonardo.utirc.utoronto.ca:8080/cgi-bin/srch-example-2/dir/file?ian+bradley
```

is shown in Figure 4.15. Accessing this URL passes both query string and extra path information to the referenced gateway program.

```
Content-TYPE: text/html

<HEAD>
 <TITLE>Local Phonebook Search</TITLE>
<ISINDEX>
</HEAD>
<BODY>
<H1>Local Phonebook Search</H1>
Enter your search in the search field.<P>
This is a case-insensitive substring search: thus
searching for 'ian' will find 'Ian' and Adriana'.
</BODY>
```

Figure 4.13 Nonparsed header output returned upon accessing the URL: `http://leonardo.utirc.utoronto.ca:8080/cgi-bin/nph-srch-example`.

```
#!/bin/sh
echo Content-TYPE:  text/html
echo
 if [ $# = 0 ]    # is the number of arguments == 0 ?
then              # do this part if there are NO arguments
  echo "<HEAD>"
  echo "<TITLE>Local Phonebook Search</TITLE>"
  echo "<ISINDEX>"
  echo "</HEAD>"
  echo "<BODY>"
  echo "<H1>Local Phonebook Search</H1>"
  echo "Enter your search in the search field.<P>"
  echo "This is a case-insensitive substring search: thus"
  echo "searching for 'ian' will find 'Ian' and Adriana'."
  echo "</BODY>"
else              # this part if there ARE arguments
  echo "<HEAD>"
  echo "<TITLE>Result of search for \"$*\".</TITLE>"
```

```
    echo "</HEAD>"
    echo ""<BODY>"
    echo "<P> Number of Command-line Arguments = $#. They are:"
    for i in $*
    do
            echo " <code> $i </code> "
    done
    echo "<h2> The Environment Variables </h2>"
    echo "<pre>"      # print the environment variables
    echo " SERVER_SOFTWARE = $SERVER_SOFTWARE"
    echo " SERVER_NAME = $SERVER_NAME"
    echo " GATEWAY_INTERFACE = $GATEWAY_INTERFACE"
    echo " SERVER_PROTOCOL = $SERVER_PROTOCOL"
    echo " SERVER_PORT = $SERVER_PORT"
    echo " REQUEST_METHOD = $REQUEST_METHOD"
    echo " HTTP_ACCEPT = $HTTP_ACCEPT"
    echo " PATH_INFO = $PATH_INFO"
    echo " PATH_TRANSLATED = $PATH_TRANSLATED"
    echo " SCRIPT_NAME = $SCRIPT_NAME"
    echo " QUERY_STRING = $QUERY_STRING"
    echo " REMOTE_HOST = $REMOTE_HOST"
    echo " REMOTE_ADDR = $REMOTE_ADDR"
    echo " REMOTE_USER = $REMOTE_USER"
    echo " AUTH_TYPE = $AUTH_TYPE"
    echo " CONTENT_TYPE = $CONTENT_TYPE"
    echo " CONTENT_LENGTH = $CONTENT_LENGTH"
    echo "</pre>"
    echo "<H2>Result of search for \"$*\".</H2>"
    echo "<PRE>"
    for i in $*
    do
        grep -i $i /vast/igraham/Personnel done
    echo "</PRE>"
    echo "</BODY>"
fi
```

Figure 4.14 Bourne-shell script ***srch-example-2***. This is a modification of the script in Figure 4.7 to explicitly print the environment variables and the command-line arguments.

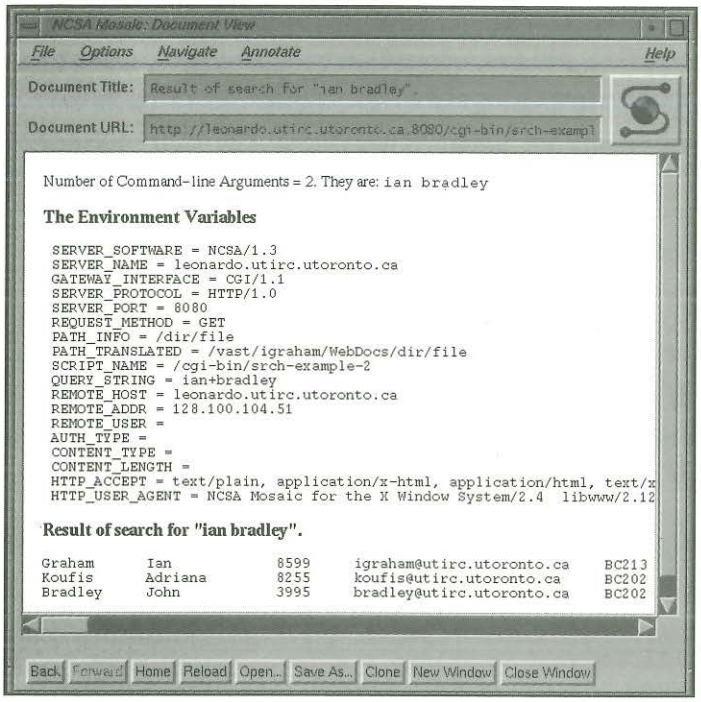


Figure 4.15 Document returned from the script in Figure 4.14 after accessing the URL: `http://leonardo.utirc.utoronto.ca:8080/cgi-bin/srch-example-2/dir/file?ian+bradley`.

Most of the environment variables shown in Figure 4.15 are easy to understand. Some are set by default and do not depend on the nature of the request, while others are set only when particular client-server-gateway interactions are involved. The following is a list of the different variables and their contents, discussed with reference to the data displayed in Figure 4.15.

SERVER_SOFTWARE=`server_name/version`—The name and version of the server software answering the request. The format is `name/version`, as in `NCSA/1.3` for this example.

SERVER_NAME=`domain.name`—The Internet domain name of the server; here, `leonardo.utirc.utoronto.ca`. If the domain name is not available, this will be the numerical IP address.

GATEWAY_INTERFACE=`CGI/version`—This gives the particular version of the CGI interface specification being used by the server. The current version is 1.1, so this should be `CGI/1.1`.

SERVER_PROTOCOL=`HTTP/version`—This specifies the protocol being used, namely HTTP, and the related version number. This allows the CGI specification to be used with servers that support protocols other than HTTP, or that support multiple protocols (for example, both Gopher and HTTP). The current HTTP protocol version is 1.0, so this should be `HTTP/1.0`.

SERVER_PORT=`number`—The port number called by the client: 8080 in this example.

REQUEST_METHOD=`HTTP_method`—The method associated with the request. For HTTP server access, this will be GET, HEAD, POST PUT, and so on. This allows you to write a single gateway script or program supporting multiple methods; the script can use the contents of this variable to decide which portions of the program to execute.

PATH_INFO=`extra_path`—This contains any extra path information found in the URL. In our example, this contains the string: `PATH_INFO=/dir/file`.

PATH_TRANSLATED=`/transl/extra_path`—This contains the *PATH_INFO* path translated into an *absolute* document path on the local system. For example, suppose the server document directory (the directory under which all documents are kept) is ***/vast/igraham/WebDocs***. The translated path is then:

PATH_TRANSLATED=`/vast/igraham/WebDocs/dir/file`

Note that this is not related to where the gateway program is physically kept. The translated path can be used to locate configuration files for use by the gateway program.