DRAFT
DO NOT CIRCULATE

.

.

Introduction to Interactive Web Design

Timothy J. Hickey
Associate Professor of Computer Science
Brandeis University

July 8, 2002

Contents

List of Figures

Preface

Currently web programming is viewed as the domain of computer science majors and out of the reach of most computer users. One of the aims of this book is to provide a path for putting web programming expertise into the hands of any motivated computer users.

This text has been designed to be accessible to non-programmers assuming only that they have a strong interest in learning how to build interactive web sites. The material presented here has evolved over five years of teaching web programming in an Introduction to Computers class attended primarily by non-science majors. We have developed a bounty of evidence that bright students (and others) can easilly learn to develop interesting and sophisticated interactive web sites within a few short weeks.

In the process of learning to build interactive web sites, you will also have a general introduction to computer programming and the joys and frustrations of this enterprise. I wish you a satisfying journey!

Foreword

The web is a rapidly evolving technology that has already changed many aspects of our lives, from the mundane to the sublime. It is rapidly becoming a kind of super encyclopedia where we can search for the latest information on movie stars or cancer treatments. It is also evolving into a new communication medium – email has saturated the population in developed countries and now, in 2002, instant messaging is rapidly expanding into the youth culture.

Part of the great success (or at least growth) of the internet is due to the fact that it is a collaborative enterprise. Every computer that is connected to the internet can offer web services, such as hosting web pages or running instant messaging clients or acting as a remote file sharing server for MP3's or other data. One does not need a licence from the government to offer such services, but some internet access providers have been putting restrictions on home services that can be offered.

During the first five years (1990-1995) of the World Wide Web most of the web content that had been developed was in the form of web pages. As we will see in this book, web pages are written in a very simple language called HTML (HyperText Markup Language) which was designed to be easy learn, which would in turn encourage non-scientists to develop web content and help build the web. This design decision has been spectacularly effective. Personal web pages are becoming increasingly common among college students (and even among high school students). Most often these pages are a kind of autobiographical folk art. They provide a glimpse into the author's life and provide a few links to sites of personal interest. Occasionally though these web authors find a topic of deep personal interest and they create an informational web site dedicated to that topic – favorite bands or movie stars are a common theme, political or religious causes are also frequently found. Many web authors publish their photography, poetry, or other creative works. A less common, but perhaps more influential, class of web authors are those that develop well-designed and useful informational sites such as web sites listing local bike paths and hiking trails.

During the second five years (1995-2000) we saw the rise of interactive sites. There have always been search engines, but the late 90's saw the rise of ecommerce and other forms of interactive web pages. For the most part, this type of web content has remained within the purview of the computer science professionals.

There are several reasons for interactive web page development to be slow to

spread to the general population. Originally, web pages were only served from dedicated machines and system administrators were loathe to allow any but the most trusted of their colleagues to deploy interactive web pages. There are good reasons for this. A poorly written interactive web page can be a major security risk. It can also devour system resources and greatly slow down, if not crash, a server. In the past few years this has become less of a problem as most personal computers now come with preinstalled web servers; thereby democratizing the process for all those who have an internet connection.

The more important impediment to the spread of interactive web pages among general users is that the langauges for developing such pages have just been too difficult for novices to learn. Originally these pages were written in primarily in PERL, an interesting but rather bizarre language favored by hackers both for its power and for its mystique! More recently, Java has become a popular language for writing these pages, but this still requires authors to have at least one semester's worth of programming experience before they can even start to build interactive web pages.

Goals of this book

In this book, we return to the original ideals of the World Wide Web and present a simple language, Scheme, for developing interactive web pages and other web programs. Scheme is similar to HTML in that it is easy for non-experts to learn the basic language and to build build fairly sophisticated interactive web pages. Scheme is a dialect of Lisp (a language which was first developed in 1957 at the dawn of the computer era).

We begin with an introduction to HTML and CSS for developing static web pages.

Next we give an introduction to writing Scheme servlets, the most basic form of interactive web page. These pages are written in a mixture of Scheme and HTML.

We then introduce the JLIB windowing library and show how to write and deploy programs that use Graphical User Interfaces. These programs are entirely written in Scheme.

In the second part of the text we move on to more sophisticated topics such as developing web programs that access databases and writing programs that involve communication among multiple users.

Hardware and Software Requirements

The software that is used in this text is all free and open source. It can be easily downloaded and installed on almost all platforms (Windows, Mac, Linux, Unix). If this text is being used in a course, we also provide instructions for setting up a central server which removes the requirement for all students to

set up their own server. On the other hand, it is relatively easy to do and a lot more fun to run your own web server.

Chapter 1

The Internet

The internet is a worldwide network of computers with the property that each computer can send data to and receive data from any other computer on the internet.

1.1 A brief history of the internet

The idea of connecting computers via phone lines or some other long distance network was first tested in 1965 when two university researchers, Larry Roberts and Thomas Merrill, connected a computer in Massachusetts with one in California using a phone line and demonstrated that they coould run programs and receive data on the remote machines. A key idea behind this research was that the computers would communicate by breaking up their data into many small packets and sending these packets individually. If any packets were lost (due to background noise on the line), they could easily be resent.

This experiment led directly to a DARPA (Defense Advanced Research Projects Agency) proposal in 1967 to build the ARPANET, which is a military precursor of the internet. In 1968 a group led by Frank Heart at BBN in Boston won the government contract to build the initial ARPANET hardware. In 1969, the initial ARPANET was constructed and consisted of four computers: three in California and one in Utah. In 1972, Roberts wrote the first email program, and email quickly became the most frequently used network application. In 1973, Vint Cerf and Robert Kahn proposed a new set of communication rules for the computer networks called TCP/IP (Tranmission Control Protocol/ Internet Protocol) which allowed users to implement a wide range of network applications including network telephony, email, and network disk sharing. The ARPANET was converted to a TCP/IP net in 1983 at which point it was split into two nets: the MILNET for military applications and the ARPANET for civilian applications. Throughout the 70's several other networks were developed. These included CSNET (connecting Computer Science Departments), USENET (connecting UNIX computers), and BITNET (connecting academic

mainframe computers).

The 80s saw the rapid prolifieration of PC's and workstations combined into small local area networks (LANs) and these LANs came to be added to the ARPANET in greater numbers, resulting in a rapid growth of the internet. Also, in 1985, the NSFNET was formed by the National Science Foundation with the stipulation that a university could connect to this network only if it provided access to all scholars at the institution, not just the science departments. Another important development during the 1980's was the connection of networks into a single internet all using the TCP/IP protocol for communication. The 90's saw the birth of the World Wide Web and the rapid expansion of the internet both in terms of size and in terms of its use by the general population.

1.2 Internet Addressing: domain names and IP addresses

The internet currently consists of about 100 million servers although this number grows every month (and actually oscillates minute by minute for reasons that will become clear). Each computer on the internet has a unique identification number called its IP address (for Internet Protocol). An IP number consists of a sequence of four numbers in the range 0-255. For example, a typical IP address at Brandeis in 2002 is 129.64.2.10, where the numbers in the IP address are separated by periods by convention. This is the dotted decimal form of an IP address.

IP addresses are actually stored on the computer and transmitted as 32 bit long binary numbers. Please read the appendix on binary numbers to learn about binary numbers and how they are used to represent decimal numbers.

Most computers on the internet also have an identifying name known as a domain name. For example, the domain name for the main Brandeis web server is `www.brandeis.edu` and its IP address is `129.64.99.138`. The relationship between domain names and IP addresses is available on the net from computers known as domain name servers.

The internet actually consists of a large number of networks which are seamlessly interconnected. For example, the Local Area Network (LAN) at Brandeis University consists of a few thousand computers. These computers are all directly connected to the internet and have IP addresses of the form

```
129.64.xxx.yyy
```

where xxx and yyy are numbers in the range 0-255. Conversely, any IP address of this form refers to the Brandeis LAN. Thus, the Brandeis LAN can expand to include up to 256 x 256 = 65536 computers which can all be simultaneously directly connected to the internet. This method of allocating IP addresses in blocks is widely used today.

1.3 Ports, Sockets, and Services

The computers on the internet interact in a wide variety of ways, but their interaction is nonetheless restricted. It would not be wise to allow any computer on the internet to have full access to every other computer on the net because an unscrupulous user might decide to delete all of your disk files or to otherwise use your computer without permission.

To get around this problem, the internet is modelled on an abstract view of the net in which each computer specifies exactly what kinds of interactions it will allow. These types of interactions are called **services** and each computer on the net can offer up to 65536 services.

These services are specified by a number from 0 to 65535 called a **port**. Typically, the ports with numbers under 1024 are reserved for system services (such as email and web page serving), but anyone is free to offer any service they please on ports numbered greater than 1024.

A computer that offers a service to another computer is called a **server** and a computer that requests a service is called a **client**. It is typical for computers on the internet to be both clients and servers and the same time. The communication between client and server is initiated by the client by specifying the IP address of the server computer and the port number of the service to be provided. If the specified computer is offering that service, then a special connection called a **socket** is created. The socket allows the two computers to send data back and forth between themselves.

1.3.1 Common Services on the net

Some of the more common system services are listed in Figure 1.1. Each service has a set of rules governing how the client and server interact. These rules are called protocols and they simply represent the conventions that the two computers will use when communicating on that port.

You can access some of these ports from Linux using the telnet command. For example, Figures 1.2 and 1.3 give examples of accessing the date and echo services respectively: The date service returns the local time on the server being queried. The echo service is used for testing whether a connection is active and just echo back each line of text that it receives.

1.4 Web Browsers and Servers

The HTTP service is perhaps the most revolutionary service that has been developed for use on the internet. It provides a mechanism for clients to access files on the server by giving the name of the file in the webserver folder. The HTTP server then responds to such a request by returning several lines of information about the file (e.g. what kind of data it contains, text, image, movie, sound, etc.) when it was last modified, how large the file is, etc. HTTP services are generally provided on port 80.

- Echo (port 7) an echo service, simply echos back what it receives
- Daytime (port 13) this returns the date and local time and ignores client input
- FTP (ports 20,21) allows the client to transfer files of data to and from the server.
- Telnet (port 23) allows the client to interact with the servers operating system remotely
- SMTP (port 25) offers an email service for delivering email to a user on the server
- DNS (port 53) domain name serving, returns IP addresses for domain names
- WWW (port 80) uses the HTTP protocol and sends specified web pages to the client.
- POP3 (port 110) offers another email service

Figure 1.1: Some of the most common services

```
USER  % telnet www.cs.brandeis.edu 13
      Trying 129.64.2.3...
      Connected to diamond.cs.brandeis.edu.
      Escape character is '^]'.
      Thu Aug 31 15:55:41 2000
      Connection closed by foreign host.
```

Figure 1.2: Accessing the date service on port 13

```
USER  % telnet www.cs.brandeis.edu 7
      Trying 129.64.2.3...
      Connected to diamond.cs.brandeis.edu.
      Escape character is '^]'.
USER  This is the echo port
      This is the echo port
USER  bye bye
      bye bye
USER  ^]
      telnet>
USER     quit
      Connection closed.
```

Figure 1.3: Accessing the echo service on port 7

The HTTP service is one half of the technological foundation of the World Wide Web. The other half is the HTML language. HTML is an acronym for Hypertext Markup Language. HTML specifies the layout of webpages and provides mechanisms for including links to other webpages and to images, sounds, movies, and other content. In the next Chapter we will provide an introduction to HTML and some related technology (CSS and XML).

Figure 1.4 gives an example of the use of this service to request the web page "/ cs2a/index.html" from the server "www.cs.brandeis.edu". Observe that the request specifies the page to access and the response provides quite a bit of information about the file including its size, its last modification date, its size, what type of information is in the file, the kind of server that is providing the service, the locate time at which the page is being served, and some more arcane information as well.

There are many web browsers that are currently available. The most common browsers at the moment are Internet Explorer and Netscape, but some of the lesser known browsers such as Opera and Amaya, provide additional features which are not currently supported by the mainstream browsers such as mathematical and graphical markup processing.

1.5 URLs and Domain Names

As you probably know, all of the content on the World Wide Web can be accessed by providing its address to a browser. The formal name for a web address is **URL** which stands for Universal Resource Locator[1] URLs are our first example of a formal language. Each URL has several parts, some of which are optional. Some examples of URLs are:

`http://www.brandeis.edu`

[1]Some people also use URI for Universal Resource Identifier.

```
USER  % telnet www.cs.brandeis.edu 80
      Trying 129.64.2.3...
      Connected to diamond.cs.brandeis.edu.
      Escape character is '^]'.
USER  GET /~cs2a/index.html HTTP/1.0

      HTTP/1.1 200 OK
      Date: Thu, 31 Aug 2000 20:21:23 GMT
      Server: Apache/1.3.4 (Unix)
      Last-Modified: Wed, 23 Aug 2000 21:32:40 GMT
      ETag: "4962a3-217-39a442f8"
      Accept-Ranges: bytes
      Content-Length: 535
      Connection: close
      Content-Type: text/html

      <HTML>
      <TITLE>Brandeis University, Intro to Computers,
              CoSci 2a, Aut 99</TITLE>
      <BODY bgcolor="#ffffff">
      <META HTTP-EQUIV="Refresh" CONTENT="1;
       URL=http://www.cs.brandeis.edu/~tim/Classes/Aut00/CS2a/">

      The Home page for CS2a has moved to
      <A HREF=
        "http://www.cs.brandeis.edu/~tim/Classes/Aut00/CS2a/">
      http://www.cs.brandeis.edu/~tim/Classes/Aut00/CS2a
      </A>
      <p>
      You can click
      <A
      HREF="http://www.cs.brandeis.edu/~tim/Classes/Aut00/CS2a/">
      here
      </A>
      to get to that page.
      <p>
      Sorry for the inconvenience.
      <p>
      Tim Hickey

      </BODY>
      </HTML>
       Connection closed by foreign host.
```

Figure 1.4: Accessing the HTTP service on port 80

```
http://www.brandeis.edu:80
http://www.brandeis.edu:80/index.html
http://www.brandeis.edu:80/go/index.php?go=cosi
http://129.64.2.3/~tim
http://jscheme.cs.brandeis.edu:8080
ftp://ftp.cc.gatech.edu/pub/linux/
```

The simplest form of a URL is just:

```
http://DOMAINNAME
```

where "http" , and "DOMAINNAME" .

The general form for a URL is

```
PROTOCOL://DOMAINNAME:PORT/PATH/FILE.EXT#P?N=V&N2=V2 ...
```

Lets break this apart.

- "PROTOCOL" specifies the protocol that the web browser must use to communicate with the web server. There are many other protocols besides http. The most common is "ftp" which is the "file transfer protocol." The "mailto:" protocol is also common and is used to allow the user to send email from a browser.
- DOMAINNAME is the symbolic name of the web server. All web servers have a unique IP address (as described above). The conversion between domain names and IP addresses is performed using special servers on the net called "Domain Name Servers." These servers accept domain names and send back the corresponding IP address. They are the equivalent of the "411" service on phone networks and every browser must have the address of at least one Domain Name Server if it is going to use domain names. You can use IP addresses directly in the URL instead of a domain name, but this is rarely done as IP addresses can be hard to remember.
- "PORT" is a number between 0 and 65535 which specifes the port used by the server, the default value is 80 for http protocols
- "PATH" specifies the location of the file on the server
- "FILE.EXT" is the name of the file to be returned
- "#POS" specifies a location in the file. POSITION is a symbolic name (containing no blank spaces). The browser will scroll the window down to that position when first viewing the file.
- "?N=V&..." This is a mechanism for passing values to the server which it can use to generate the web page it will send back to you. You often seen this kind of an address after filling out a search form for a search engine. When you bookmark this page and return to it, the URL contains a precise description of everything you entered on the form.

These more complex URLs will be come easier to understand after we have discussed servlets in a later chapter.

1.6 Scheme Servers

In this course you will need to have access to a web server in order to publish your HTML pages, your servlets, and you applets. Appendix A explains how to download and setup a web server on your PC, Mac, or Linux platform. When you install the server, a folder `scheme` will be created.

This folder represents the public web services that you will offer to the world. The types of service that you can currently offer using this server are listed below and are determined by the suffix of the filename. Each different suffix refers to a different web programming language that is handled by the server:

html Files ending in `.html` are simply sent directly to the client as `text/html` files.

css These are cascading style sheet files and are used to define extensions of the basic HTML language.

servlet These are scheme XML files which provide a mechanism for reading information from HTML forms and using that information to generate web pages which are then sent to the client. (There are two other variants of this type of file with the extensions `sssp` and `sxml`. They are essentially the same language, but have slightly different grammars).

applet These are programs that run on the browser that downloads the page. These programs usually pop up windows with menus, buttons, textfields, choices, and all the usual facets of graphical user interfaces that you have become accustomed to.

snlp These are programs that also run on the client machine, but they require the Java Web Start plugin and, once downloaded, they can be run without the browser. One particular interesting type of snlp program that we will consider is the class of peer-to-peer programs. The Napster and IM programs are a well-known examples of this type of application. But there are many others possible uses for peer-to-peer computing.

Part I

Static Web Site Design

In the next two chapters we provide an introduction to the core languages used for writing web pages: HTML and CSS. The HTML language is used for specifying the structure of a document. It is used to divide the content into paragraphs, headers, tables, images, movies, hyperlinks in such a way that the browser can then present the page content to the user either visually (the usual way), or aurally (using webpage readers), or even tactilely (using braille readers). The CSS language is used for specifying visual, auditory, and/or tactile style. It allows one to change the appearance, sound, or feeling of any paragraph, header, table or other web component.

Chapter 2

HTML

A fundamental component of the original conception of the World Wide Web is that it should be a collaborative enterprise. The web was designed so that everyone would be able to add content. To make this feasible, the language for constructing web pages (HTML) was designed to be simple, powerful, and easy for non-computer specialist to master.

In this chapter we give an introduction to the design of static webpages using HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and eXtensible Markup Language (XML). As you will see, these languages are quite easy to master as they are each based on a few simple ideas.

We begin with HyperText Markup Language (HTML). In this section we provide a sufficient introduction so that you will be able to design static web pages using the most common markup elements. For more information you can visit the official HTML website at `http://www.w3c.org` or peruse any of the many HTML texts.

HTML is build upon one fundamental idea which is the notion of expressing the layout for a web page using HTML elements. As we will see below there are a few dozen basic elements which are used to express the basic layout of the page including line breaks, headings, layout of images, tables, and lists.

2.1 Simple HTML elements

Simple HTML elements have the form

```
<TAGNAME>
   .....
</TAGNAME>
```

where `TAGNAME` is the name of the tag. A complete list of the HTML 4.01 elements is available at the URL of the official source for HTML, the World Wide Web Consortium:

`http://www.w3.org/TR/html4/index/elements.html`

There are 90 different standard tag names, but we will only discuss the most common of these.

Thus a web page consists of an html element that contains a head and body element. The head element in turn contains a title element.

For example, the following text defines a simple "Hello World" webpage

```
<html>
  <head>
    <title>My first web page
   </title>
  </head>
  <body>
    Hello World
  </body>
</html>
```

If you put this in a file called `first.html` and then view the webpage with your browser (using the Open... menu item). You will see a simple web page consisting only of the words "Hello World" whose title bar contains the phrase "My first web page." If you bookmark this page, then the title "My first web page" is what will appear in the bookmark.

Exercise 1 Creating your first web page. *Use your favorite editor to create a file containing the "Hello World" web page shown above. Store your file as "text" with the name "first.html" on your disk. You can view your page by starting up a browser (e.g. Netscape, Internet Explorer, Amaya, Opera, etc.) and selecting "Open File" from the "File" menu. Select the "first.html" file that you just created and you should see a simple page with the words "Hello World" in black on a white or gray background. Beware: this simple exercise has many pitfalls and it may take you a while to complete it. This is the kind of task that is best done with someone helping you in person as the details vary from computer to computer. Some of the problems that may arise are:*

- *Some operating systems hide the extensions (.html, .txt, .jpg, .mov) that specify what type of data is in the file. These operating systems also will automatically add extensions (e.g. .txt) to files. So you may think you have a file named "first.html" when it is actually called "first.html.txt" Ouch!*

- *You may have difficulty saving the file as text. Many word processors store the document you create with a lot of extra information besides the characters that you have typed. For example, they might store the font you have used and the margins and tab setting, etc. The "first.html" file needs to be stored in a simpler format which contains only the characters you typed. This is called "text format" and is usually listed as a choice on the "save as" window.*

2.2 HTML elements with attributes

The most general form of HTML tags is as follows:

```
<TAGNAME A1=V1 A2=V2 ... An=Vn>
   .....
</TAGNAME}
```

where `A1,A2,...,An` are the names of attributes that are allowed for that tag, and `V1,V2,...,Vn` are values that those attributes can accept. In general, the attributes should always be enclosed in double quotes as this will simplify migration to XHTML which is poised to become the successor to HTML4.0 as the next interational standard.

For example, to include an image in a webpage you use the `img` tag as follows:

```
<img src="mypic.jpg" alt="picture of the author" width="200">
```

The `src` attribute of the image tag specifies the name of the image file to display, the `alt` attribute specifies the closed-captioned reading of the image, and the `width` specifies the size to make the picture (in pixels). In this case, the attributes are used to provide information needed to properly display the element.

For another example, the `<body>` tag allows one to specify the background color of the page using the `bgcolor` attribute and to specify the color of the text on the page using the `text` attribute. It also has attributes for specifying the color of unvisited links `link`, already visited links `vlink`, and links that are being clicked `alink`.

For example, the following HTML specifies a page with red letters on a black background and also specifies the link colors:

```
<body bgcolor=black text=red link=yellow alink=blue vlink=red>
  ....
</body>
```

2.3 Style and class attributes

One of the most common problems encountered when writing HTML pages is that each tag has its own set of attributes, and one must know which attributes are allowed for which tags. For example, almost all tags have a `bgcolor` attribute, but `body` is the only tag with a `text` attribute. The CSS language, discussed in detail in the next section, was developed partly in response to this problem. It provides a uniform method of specifying the "style" (e.g. color, font, border, etc.) of any HTML tag. For example, the CSS-method for specifying the red-on-black body tag shown above is the following:

```
<html><head><title>test</title></head>
  <body style="background:black; color:red">
    ....
```

```
  </body>
</html>
```

The CSS language can be used to specify four kinds of properties: fonts, colors, borders, and text spacing.

To simplify the presentation of HTML and CSS in this text, we will only use the CSS-method for specifying style and will ignore all other methods (e.g. style-based attributes of HTML tags). This allows us to ignore many HTML tags (e.g. the ones that change the font type or font color) and many HTML attributes. Indeed, Figure 2.1 show the 28 tags that we will consider and also shows their main attributes.

The tags are divided into groups that we consider one at a time. We have already seen the first four tags.

2.4 Hyperlinks

The hyperlink element has the form:

```
<a href="WEBADDRESS"> CONTENTS </a>
```

The CONTENTS is typically some text or an image. The WEBADDRESS is a URL that specifies the location of some web content online. The most commonly used URLs are

- names of files on the server,e.g. `report.html` or `doc/manual.html`
- addresses of other pages on the web, e.g. `http://www.whitehouse.gov`
- mailto URLs which allow the user to use the browser to send email to a prespecified address, e.g. `mailto:gwb@whitehouse.gov`.
- a link to some other form of media, e.g. a movie or document `home.mov` or `whitepaper.pdf`.

2.5 Images

The image element has the form:

```
<img src="WEBADDRESS" alt=TEXT width=NUM height=NUM>
```

This is one of the few tags that does not have a matching "close" tag. The only required attribute is the `src`, but it is a good idea to include a description of the image for the vision-impaired in the `alt` attribute. This may even be mandatory if you want the page to meet minimum Federal Accessibility Standards.

The `width, height` attributes are optional and they can be used to rescale the size of your image. Giving only the width will cause the height to scale proportionately. Giving height and width may result in a picture that looks stretched or flattened.

```
Structural Elements
   HTML
   HEAD
   TITLE
   BODY    link=COLOR vlink=COLOR alink=COLOR
Links
   A       href=URL name=STRING
Images
   IMG     src=URL alt="TEXT" ...NO-CLOSE-TAG
Headings, text separation
    H1,H2,H3,
    H4,H5,H6
    HR                         ...NO-CLOSE-TAG
    P
    DIV
    BR                         ...NO-CLOSE-TAG
    SPAN
Preformated text
    PRE
Lists
    OL
    UL
    LI
Tables
    TABLE  cellspacing=DISTANCE cellpadding=DISTANCE
    TR
    TD
    TH
Frames
    FRAME  name=STRING src=URL  ...NO-CLOSE-TAG
    FRAMESET rows=LENGTHS cols=LENGTHS
    NOFRAMES
Forms
    FORM     method=POST action=URL
    INPUT    type=TYPE name=STRING ...NO-CLOSE-TAG
    TEXTAREA  name=STRING
    SELECT    name=STRING size=NUMBER multiple
```

Figure 2.1: The Main Elements/Attributes of HTML 4.01

/hrule

2.6 Headings

There are six levels of headings; from the largest `h1`, to the smallest `h6`. Their general form is

```
<h1> CONTENT </h1>
```

where CONTENT is typically text and/or images. The `style` attribute can be used to specify the font size, background color, and text color.

2.7 Text Separation

HTML offers several elements that can be used to separate text on a page. When these elements are combined with CSS, they allow the web page designer to specify the style of different sections of the webpage.

The `span` element is used to group together some part of a line (or lines) of text. It has the form:

```
<span> CONTENT </span>
```

where the CONTENT is typically text and/or images. The span element does absolutely nothing by itself. It only becomes useful when combined with CSS, as it allows one to apply a `style` attribute to a short inline segment of words and/or images as in

```
See the blue <span style="border: thick blue solid"> box </span>
around a single word.
```

The `span` element was introduced as a hook on which to attach CSS to small segments of text.

The `br` elemeent is used to insert line breaks into the page. It has the form:

```
<br>
```

and observe that it does not have a close tag. Another way to break lines is to insert a horizontal rule, which is a line which stretches all or part way across the page and is used to visually separate parts of a page. These "rules" are inserted using the

```
<hr>
```

tag which also does not have a close tag.

Paragraphs are inserted into a page using the

```
<p>
 CONTENT
</p>
```

tag, where CONTENT is typically text and other markup with some restrictions (e.g. it can not contain any block-level elements, e.g. `p`, `table`, `div`, or `h1`. elements).

Finally, the most general way to separate the content in a page is to use the `div` element, which has the form:

```
<div>
  CONTENT
</div>
```

where CONTENT can be any of the HTML elements that can appear in the body, including `p` and `div` elements. The `div` tag is similar to the `p` tag except that a `div` element can contain a wider variety of tags. The `p` element should only be used for paragraphs containing text and images.

2.8 Preformatted text

Browsers, by default, will reformat any text that you provide so that it fits the page nicely. Thus, if you type a paragraph as one long line, the browsers will generally add appropriate line breaks. Sometimes however, one wants the browser to respect the formatting and not to insert any line breaks or remove any spaces or tabs. This effect is provided using the `pre` element which has the form

```
<pre>
     Pre
       Formatted
   Content
</pre>
```

The preformatted content inside the `pre` element typically contains images and text.

2.9 Lists

HTML offers several different types of lists. We consider only two types here: `ul` and `ol` The `ul` element is used for "unnumbered lists" and has the form:

```
<ul>
  <li> Content </li>
  ....
  <li> Content </li>
</ul>
```

Observe that the `ul` element must contain a sequence of `li` elements, and each `li` element can contain any of the HTML elements that can appear in the body. These lists are rendered with asterisks or bullets or some other non-alphanumeric

list item markers. CSS can be used to specify the type of list item maker used, e.g.

```
<ul style="list-style-type:square>
  <li> Content </li>
  ....
  <li> Content </li>
</ul>
```

will create square boxes instead of the usual round bullets, for each list element.

The `ol` lists are used for "ordered lists" and have the same format:

```
<ol>
  <li> Content </li>
  ....
  <li> Content </li>
</ol>
```

but they render their list item markers using numbers. CSS can be used to specify that letters or roman numerals be used instead (e.g. using "lower-roman" or "upper-alpha" for the list-style-type).

2.10 Tables

Tables are a very useful formatting tool for web pages. They provide a mechanism for presenting tabular data and specifying how the table should appear on the page. The general form of the table element is:

```
<table cellspacing=DISTANCE cellpadding=DIST>
 <tr>
   <th>Content</th> .... <th>Content</th>
 </tr>

 <tr>
   <td>Content</td> .... <td>Content</td>
 </tr>

 ....

 <tr>
   <td>Content</td> .... <td>Content</td>
 </tr>
</table>
```

The contents of the table must consist of a sequence of `tr` elements representing the rows of the table. Each row consists of a sequence of `td` elements corresponding to the data stored in each cell of the table. Optionally, the first row

can contain `th` elements which correspond to heading data. The `cellspacing` attribute is used to specify how much space should appear between the cells of each table. The `cellpadding` attribute specifies how much space should appear within a cell around the content of that cell.

2.11 Comments

Comments are pieces of text that are, for the most part[1], ignored by the browser. They are used to provide information to the person maintaining the web page, not the person viewing it. You add comments to an HTML page using the following syntax:

```
<!--  comment goes here
      can include any text, except
      cannot have two consecutive dashes (-)
      as that indicates the end of the comment
  -->
```

Note that the "–" dashes delimit the comment part of the tag. The tag's name is "!" and it has not close tag. Also, spaces are forbidden between the ! and the –.

2.12 Frames

Frames provide a mechanism for combining several webpages into a single webpage.

[1]The only exception is in the `<style>` tag which appears in the head, the CSS style is enclosed in a comment so that older browsers will ignore it. This was needed because older browsers would treat the CSS specification as text and show it on the screen.

Chapter 3

CSS

Cascading Style Sheets (CSS) is a language for specifying various aspects of the style of HTML elements. In this section we will provide only an overivew of the most common uses of CSS. The reader who want to go into more depth can visit the official source for CSS, the World Wide Web Consortium site at

```
http://www.w3.org/Style/CSS
```

There are several ways of adding style to HTML. We will focus on the "inline" method and discuss the other methods briefly at the end.

3.1 The Style Attribute

The key idea behind CSS is that it provides *uniform* methods for defining the style of HTML elements. The `style` attribute is one such method.

Every HTML element in the body of an web page can have a `style` attribute. The style attribute has the form:

```
style="Prop1:Value1; Prop2: Value2; ... ; PropN: ValueN"
```

Observe that the style specification is enclosed in double quotes and consists of a sequence of `Prop:Value` specifiers, separated by semicolons.

There are eight basic properties: font, color, background, margin, padding, border, width, height. These are applied to an HTML element by imagining that the element is contained in an invisible box. The element can be as small as a single charater,chapter like `<span>word</span>`. At the other extreme, the box for the `body` tag is the entire web page.

3.2 Parent and Children styles

As you have noticed, many HTML elements can contain other HTML elements. For example, the body element can contain heading, paragraphs, tables, etc. We

say that the outer element (in this case the body element) is the **parent** and the inner elements are the `children`. Many CSS styles are inherited by children. For example, if you set the background color of the body to yellow, then all elements in the body will also be yellow (unless explicitly specified otherwise).

3.3 The `font` Property

The `font` property accepts values that specify how the text that appears within the element should be rendered. The minimal form of the font propery specifier is

```
style="font: STYLE SIZE FAMILY ; .... "
```

where `STYLE` is typically "bold", "italic" or simply omitted, `SIZE` is typically expressed in points, e.g. `12pt`, and `FAMILY` is one of the following standard font families: `serif`, `sans-serif`, `cursive`, `fantasy`, `monospace`. Thus, one could create a heading with large sans-serif letters as follows:

```
<h1 style="font: 60pt sans-serif">Intro to CSS</h1>
```

One can also specify that the font should be `italic` or `bold` and one can specify the spacing between the lines along with the font size. For example, the following element defines a paragraph with a bold italic 12 point serif font, and the paragraph is doubled spaced as the spacing between every two lines is 24 point.

```
<p style=
     "font: bold italic 12pt/24pt serif">
  This is bold, italic font in a 12 point serif font
  with 24 point interline spacing
</p>
```

The five font families listed above are supported on all CSS-capable browsers, but CSS allows the web designer to specify less common font families as well (e.g. 'Helvetica'). One problem that may arise with this freedom is that there is no guarantee that the browser that views your web page will have the font you have specified. CSS compensates for this by allowing the designer to specify a sequence of font families, separated by commas, ending with one of the standard font families. A CSS-capable browser will use the first font on that list which is currently available, and in the worst case will just use one of the five generic families. For example, the following heading specifies that the `Irish Ultra` font should be used if possible, otherwise the browser should use ariel, or helvetica, or if all else fails, sans-serif.

```
<h1 style=
      "font: 60pt 'Irish Ultra',ariel,helvetica,sans-serif">
  CSS Fonts
</h1>
```

Note that the single quotes around 'Irish Ultra' are needed because the font name contains a space.

3.4 The color Property

The color property of a style attribute specifies the text color of an HTML element. The particular color itself can be specified in several ways:

- Using one of the sixteen standard HTML colors: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow.
- As a hexadecimal number in the form `#rrggbb` where `r,g,b` are hexadecimal digits: 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f.
- As a 3 digit hexadecimal number `#rgb`
- As a term of the form `rgb(R,G,B)`, where `R,G,B` are numbers between 0 and 255
- As a term of the form `rgb(R%,G%,B%)`, where `R,G,B` are decimal numbers between 0.0 and 1.0.

In each case but the first, the color is specified as a combination of red, green, and blue light, and the R,G,B parameters specify how much of each of the primary colors to use in the desired color.

For example, the following html segment shows how to specify the color of individual words in a sentence:

```
Here are a few colors:
<span style="color:red"> red,</span>
<span style="color:#ffff00"> yellow,</span>
<span style="color:#0f0"> green,</span>
<span style="color:rgb(0,0,255)"> blue,</span>
<span style="color:#rgb(0.9\%,0.8\%,0.7\%)">brown</span>
```

3.5 The background Property

The background property can be used to specify the color of the background using the same syntax as the color property. For example, to make a heading with red letters on a black background you could use:

```
<h1 style="font: 48pt serif; color:red; background: black">
    Warning!</h1>
```

This property is more versatile however, in that it can also be used to specify a background image rather than a color. In this case, the value of the property is a webaddress of the form

```
  url(page.gif)  or url(http://x.com/page.gif) or
```

For example, the following body tag specifies that the“sincos.gif” image should be used as the background for the page:

```
<body style="background: url(sincos.gif)">
  ...
</body>
```

3.6 The `border` Property

CSS allows you to put borders around the box enclosing any HTML element. The general form is

```
style="border: WIDTH STYLE COLOR"
```

where

- `WIDTH` is one of `thin`, `medium`, `thick`, or is a distance measured in pixels (e.g. `10px`,) or absolute units, e.g. `0.1in`, `1cm`, `8.2mm`, `2pt`, `0.3pc`.
- `STYLE` is one of `none`, `dotted`, `dashed`, `solid`, `double`, `groove`, `ridge`, `inset`, `outset`
- `COLOR` is a color as specified in the color property discussed above.

For example, we can specify a heading with a thin, solid, blue border as follows:

```
<h1 style=
      "border: thin solid blue; font: 24 sans-serif">
  Greetings
</h1>
```

Observe that the order of the width, style, and color parameters is not important.

3.7 The `width, height` Property

These two properties refer to the size of the box that contains the element. These can be expressed in distance units (as with the WIDTH property of borders above). The width can also be exrpessed as a percentage (being interpreted as a percentage of the width of the parent element.) For example, the following code creates a 1x2 inch blue box:

```
<div style="width:1in; height:2in;
     background: blue; color:black">
</div>
```

3.8 The `margin` and `padding` Properties

The padding is the distance between the border of an element and the content. The margin specifies the distance between the parent and the border of an

element. These can be expressed as lengths or as percentages of the parent elements width. You can express a single distance for the padding of all four sides (top, right, bottom, left) as

```
<li style="padding:0.5in">first example</li>
```

Or you can specify a separate padding for each, in the order top, right, bottom, left.

```
<li style="padding:0.1in,0.2cm,0.3in,10px">second example</li>
```

The margins are specified in the same way:

```
<li style="margin:0.5in">first example</li>
<li style="margin:0.1in,0.2cm,0.3in,10px">second example</li>
```

3.9 The vertical-align and text-align Properties

The horizontal alignment can be specified to be one of the following `left, right, center, justify`. For example, a centered heading is specifed by

```
<h1 style="text-align:center;
           font: bold 24pt sans-serif">Greetings</a>
```

Finally, the vertical alignment of an HTML element can be specified to be one of `baseline`, `sub`, `super`, `top`, `text-top`, `middle`, `bottom`, `text-bottom` or a percentage (which is relative to the interline width and can be negative for lowering an element). For example, you can lower an image by

```
<img src="abc.jpg" alt="dog" style="vertical-align: -50\%">
```

3.10 Other CSS Capabilities

There are several more CSS properties (especially having to do with text, paragraph indenting, etc.), but these are the most common ones.

3.11 Using CSS via the Class attribute

In this section we show how to give names to styles in the "head" element of a webpage. This allows you to define the style once and then use it many places.

There are three ways of naming a style. You can either assign the style to an entire html tag (e.g. `p`) in which case it is applied whenever that tag appears. Or you can can give it a class name that can be used with any tag. The `NAME` in this case should start with a period (.) as in `.toocool`. Finally, you can name the style with both an HTML tag and a class name, as in `p.toocool`. In case

of conflicts, the most specific specification is used, e.g. `p.toocool` style will be used in a paragraph with the `toocool` attribute. Likewise, the `bland` style will be used in a bland paragraph. Finally, note that the style names can be any name you wish to create. The names must begin with a letter and contain only letters, digits, and dashes (-).

The style definitions can go into the head element of the html tag for example, the following code defines a "toocool" class and uses it for a heading and a sentence:

```
<html>
  <head><title>test</title>
    <style type="text/css" media="screen">
      <!--
        .toocool  {background:black; color:red}
        p         {background:white; color:red}
        p.toocoll {background:black; color:white}
        .bland    {background:white; color:black}
      -->
    </style>
  </head>
  <body>
    <h1 class="toocool">Cool page</a>
    This is neat and
    <span class="toocool">this is too cool!</span>
    <p> This is a normal paragraph with CSS style</p>
    <p class="toocool">This uses the special toocool-paragraph
    style</p> and
    <p class="bland">This is a bland paragraph</p>
  </body>
</html>
```

The style file has the form:

```
NAME {STYLE-SPEC}
NAME {STYLE-SPEC}
NAME {STYLE-SPEC}
```

where the STYLE-SPEC is a specification of the style as we have seen for inline CSS.

The "media" specification in the style tag allows you to specify different styles for different media. Current media include "screen" for usual webpage browsing, "print" for printing, and "aural" for screen readers.

You can also store the CSS in a separate file and link it to the current page using a "link" tag in the head:

```
<html>
  <head><title>test</title>
    <link rel="stylesheet" href="demo.css" media="screen">
```

```
  </head>
  <body>
    <h1 class="toocool">Cool page</h1>
    This is neat and
    <span class="toocool">this is too cool!</span>
  </body>
</html>
```

The final interesting use of CSS is that one style sheet can import one or more style sheets, creating a cascading effect. This is done with the `@import` directive in the style file or style definition, e.g.

```
<html>
  <head><title>test</title>
    <style type="text/css" media="screen">
      <!--
        @import url(http://www.whitehouse.gov/css);
        .toocool  {background:black; color:red}
      -->
    </style>
  </head>
  <body>
    <h1 class="toocool">Cool page</a\h1>
    This is neat and
    <span class="toocool">this is too cool!</span>
  </body>
</html>
```

Part II

Dynamic Web Site Design

Chapter 4

Scheme Servlets

In this chapter we explain how to develop interactive web pages. These pages will typically prompt the user for some information (using a web form or a hyperlink) and then will generate a new web page, based on the user input. They may also perform other actions such as read/writing information on the servers disk, sending email, or accssing a database.

The language we use to specify these interactive web pages is a simple combination of Scheme, HTML, and CSS. For most of the examples in this section, the scheme will be used in relatively simple ways and hence you will not need to know much about the language itself beyond the few examples we demonstrate below. In Chapter ??, we will give a full introduction to Scheme and you can then use that to build even more complex servlets.

4.1 Dynamic Content and Scheme Servlets

The key idea behind servlets is that they provide a way to generate dynamic webpages, that is webpages in which the HTML is different each time someone visits it. Servlets can be specified in many different languages. In this book, we will look closely at servlets specified in Scheme.

To run these servlets you will need to have access to a server which has a Scheme webapp. The appendix explains how to download and install such a server. If someone can provide you access to a Scheme server then you will not need to download and install the server yourself.

Once you have access to a server, you create a scheme servlet simply by creating a file in the "webapp" folder and adding the suffix ".servlet" to the file name. When someone tries to view that file, the server will read the code that you have written in the ".servlet" file and will use that code to generate a new webpage which is then sent back to the user.

Let us now consider a few simple examples of Scheme servlets. These examples are not very useful by themselves but they allow us to explore the "idea" of Scheme servlets in a simpler context. The first example is the following pro-

gram which simply returns the date. We assume this is stored in a file named "date.servlet" stored in the "scheme" folder of the server directory. This file contains the following lines:

```
; date.servlet -- returns the current time and date
; lines beginning with a semicolon are comments
(Date.)
```

The first two lines are comments (as they begin with a semicolon) and so are ignored by Scheme. The second line is a Scheme expression which returns the current local date and time[1]. To view this servlet, one must visit the URL:

```
http://MY.DOMAIN.EDU:8080/scheme/date.servlet
```

where `MY.DOMAIN.EDU` is the domain name of the scheme server you are using. If you have installed a Scheme server on your home computer, then you can use the IP address of your home computer in place of `MY.DOMAIN.EDU` or you can use the "self-loop" IP address

```
127.0.0.1
```

which always refers to the computer itself. Visiting this address with a standard browser will bring you to page that contains the current time and date:

```
Sun Jan 20 07:46:44 EST 2002
```

The expression `(Date.)` is an example of a Scheme expression. It is an invocation of the `Date.` procedure with no arguments.

4.2 Arithmetic Expressions in Scheme

Another simple example of a servlet is the following which we assume is stored in a file named `sumToTen.servlet`:

```
; sumToTen.servlet -- this returns the sum of 1 to 10
(+ 1 2 3 4 5 6 7 8 9 10)
```

When one visits this servlet with a browser, the server evaluates this expression and applies the addition operator `+` to the arguments that follow it in the parenthesized list. The result that is return to the browser is the number

```
55
```

In a way the scheme server can be viewed as a calculator, where you put the expression you want to compute into a file and when you view the file with a browser it will evaluate that expression and return your answer. A slightly more complex example is the following:

[1] Actually, `(Date.)` returns a "Date" object representing the current time and date. This Date object is turned into a string representing the date in the local dialect.

```
; hypotenuse34.servlet -- this returns the length
; of the hypotenuse of a right triangle who other sides
; have length 3 and 4
(sqrt (+ (* 3 3) (* 4 4)))
```

This servlet contains a more complex scheme expression with four operators (two multiplication operators "*", one addition "+", and one square root "sqrt"). The Scheme server evaluates such an expression by first evaluating the innermost expressions `(* 3 3)` and `(* 4 4)` to get `9` and `16` respectively. These values are subsituted back into the expression to yield the simpler expression:

```
(sqrt (+ 9 16))
```

Next the sum is evaluated to yield `25` which is substituted back into the expression to get:

```
(sqrt 25)
```

which evaluates to

```
5.0
```

and this is sent back to the browser.

Scheme provides a large set of arithmetic operators including the following

```
(+ a b ... c) addition
(- a b) subtraction
(* a b ... c) multiplication
(/ a b) division
(sqrt a) square root
(exp a) exponential function
(log a) natural logarithm
(sin a) sine (in radians)
(cos a) cosine (in radians)
(tan a) tangent (in radians)
(asin a) arcsin
(acos a) arccosine
(atan a) arctangent
(round a) rounds to the nearest integer
(expt a b) raise a to the b power
```

Exercise 2 *What is the result of evaluating the following Scheme expressions:*

```
(* 1 2 3 4 5)
(+ 5 (* 3 (+ 6 (* 7 2))))
(+ (* 1 1) (* 2 2) (* 3 3) (* 4 4) (* 5 5))
```

4.3 Interacting with HTTP headers

Another example of a servlet is the following which sends back a message to the browser (using the HTTP protocol) telling it to visit another page instead:

```
(.sendRedirect response "http://www.nsf.gov")
```

Visiting this page in a browser will bring you directly to the web page of the National Science Foundation. In this example, the scheme expression that is evaluated has an operator `.sendRedirect` and two operands: `response` and `"http://www.nsf.gov"`. The first operand is a special one that refers to the page that is sent back to the browser. The second operator is a string of characters which is indicated by the enclosing double quotes. There are two other "special" symbols used by the scheme server: `request` and `httpservlet`. The `request` symbol can be used to get information about the current request. For example, the IP address of the browser that is visiting the current page can be returned using the following servlet:

```
; yourURL.servlet
(.getRemoteAddr request)
```

Visiting this page with your browser will return your current IP address. If you are running the server on your own machine, it will probably return the "loop back" IP address:

```
  127.0.0.1
```

You should try the examples in this section with your own server and browser.

Exercise 3 *Try out servlets that report on the client's CPU and operating system using the Scheme calls*

```
   (.getHeader request "ua-os")
and
   (.getHeader request "ua-cpu")
```

4.4 Quasi-strings

The simplest type of useful servlet is one which just returns the same HTML each time to the user. These servlets are written as standard HTML files except that the file name must end in ".servlet" and the HTML ust be enclosed in a pair of curly braces.

```
{<html><head><title>Simple servlet</title></head>
  <body>
    <h1> This is a simple servlet</h1>
     it has no dynamic content!
  </body>
 </html>
}
```

Note that if you want to include any curly braces in your webpage you have to put backslashes in front of them[2].

```
{
  <html><head><title>Simple servlet</title></head>
    <body>
      Like this \{ and this \}
    </body>
  </html
}
```

A slightly more interesting servlet is one that display the current local time. This is done by including the text

```
   [(Date.)]
```

at the point in the web page where you want the current time and date to appear.

```
; date.servlet -- this returns a page with the date/time
{
 <html>
   <head><title>Current Time and Date</title></head>
   <body>
     The current local time and date is <br>
    [(Date.)]
    <br>
   (This page is powered by Scheme servlets!)
   </body>
 </html>
}
```

The square braces "[" and "]" in the `date.servlet` servlet tell the server that the element they enclose is a Scheme expression that should be run to get its value. That value is then inserted directly into the page. Returning to the `date.servlet` example, if you then visit the URL

```
http://SERVER.DOMAIN:8080/scheme/date.servlet
```

you will get a web page that contains the following text:

```
     The current local time and date is
     Sun Jan 20 07:46:44 EST 2002
    (This page is powered by Scheme servlets!)
```

Clearly, one can create a more interesting page by using CSS and more HTML markup.

[2]The backslash indicates that the following character is to be viewed as just a character.

Exercise 4 *Modify the date.servlet example above to include more sophisticated HTML and CSS, then visit the page and hit the refresh button several times. You should notice that the time changes each time you hit the refresh button.*

Exercise 5 *Write a servlet that displays a random number using*

```
[(Math.random)]
```

This scheme expression generates a random decimal number between 0.0 and 1.0. If you want a whole number between say 0 and 100, then you can use the following expression instead:

```
[(round (* 100 (Math.random)))]
```

By multiplying the random number by 100, we get a decimal number between 0.0 and 100.0. The `round` *operator then rounds that number to the nearest whole number in the range 0 to 100.*

Exercise 6 *What is the result of evaluating the following Scheme expressions:*

```
{ 3 + 4 = [(+ 3 4)] }
{ [(* 3 3)] - [(+ 2 5)] = [(/ 4 2)] }
{ A day contains [(* 24 60 60)] seconds!}
```

As mentioned above, the curly braces in a servlet {} indicate that the enclosed text is to be sent verbatim to the client, except that text enclosed in square braces

is first evaluated to get some "interesting" value which is then inserted into the text. This curly brace/square brace notation is called quasi-string notation. It is another example of a Scheme expression.

4.5 Quasi-strings and `string-append`

The general form of a quasi-string expression is as follows:

```
{...text...[A].....
 ...text...[B].....
 ....
 ...text...[C].....}
```

The rule for evaluating such an expression is to first evaluate the embedded Scheme expressions `A`, `B`, ..., `C`, and then to insert the values one obtains in the corresponding places in the string. There is a Scheme function `string-append` which can be used to achieve a similar effect, e.g. instead of writing:

```
{the sum of 5 and 7 is [(+ 5 7)] the difference is [(- 5 7)]}
```

one could write

```
(string-append
  {the sum of 5 and 7 is }
  (+ 5 7)
  { the difference is }
  (- 5 7) )
```

both of these return the string

```
"the sum of 5 and 7 is 12 the difference is -2"
```

Thus, the quasi-string notation is just a short-hand for the `string-append` expression.

4.6 Servlet parameters

Next we show how to create a simple kind of interactive web page in which you can pass values to the servlet which will then be used to generate the webpage returned to the browser.

Our first example is the following simple echo servlet

```
; echo.servlet -- this returns the value of the
; servlet parameter "a"
(servlet (a)
  a)
```

If we visit the URL

```
http://SERVER.DOMAIN:8080/scheme/echo.servlet?a=HELLO!
```

Then the server will call the `echo.servlet` and will pass it the parameter string `?a=HELLO!`. The expression

```
(servlet (a) EXPR )
```

will extract the string `HELLO!` and bind it to the variable `a`. It will then evaluate the body (in this case `a` itself, which it returns to the browser. Thus, if we visit this URL, we will get a page containing

```
HELLO!
```

Note that we would get the same result if we visited the following URL:

```
http://SERVER.DOMAIN:8080/scheme/echo.servlet?b=zz&a=HELLO!&c=7
```

This URL shows how to pass three values to a servlet. Our servlet is only interested in the value for the parameter `a` so it ignores the other two parameters.

4.7 The case expression

The echo servlet will behave poorly however if you don't give it any parameter. It will usually just cause the browser to hang. To rectify this problem, we can use the following nicer echo servlet:

```
; niceEcho.servlet
(servlet (a)
  (case a
    ((#null){Please add "?a=Hi" to the URL and hit return})
    (("Hi") {Good job. Now try "?a=Hello"})
    (("Hello") {Great. Now try "?a=ANYTHING"})
    (("ANYTHING") {OK, OK. Now try something more original})
    (("original" "something") {Are you trying to be funny?})
    (else {You just typed [a]})))
```

If one visits this page without an "a" parameter, then the value assigned to "a" by the servlet will be the special value "#null" and it will return some instructions about how it should be called. Otherwise, it will look at the value passed in for "a" and if it matches one of the quoted expressions "Hi", "Hello" etc. it will return the corresponding expression. Note that "original" and "something" both trigger the same response "Are you trying to be funny?". If no match is found, then the servlet just echoes back what the user entered.

The `case` expression in general has the form

```
(case a
  ((value1) result1)
  ((value2 value3) result2)
   ...
  (else lastresult))
```

When the case expression is evaluated, the Scheme system tries to find the first match of `a` with one of the `value`s. When a match is found, the corresponding `result` expression is evaluated. If no match is found then the `else` clause applies and the `lastresult` expression is evaluated. The `niceEcho.servlet` shows how to use the `case` to support an interaction between the user and the servlet.

Exercise 7 *What is the result of evaluating the following Scheme expressions:*

```
(case "b"
  (("a" "c") "hello")
  (("b" "d") "goodbye")
  (("a" "b") "wow")
  (else "darn"))

(case (* 2 2)
  ((1 3 5 7 9) "odd digit")
```

```
  ((0 2 4 6 8) "even digit"))
```

Next we give an example of a servlet that handles many parameters:

```
;multiEcho.servlet
(servlet (color name age)
  {
    <html><head><title>multiEcho</title></head>
      <body style="background:[color]">
        <h1> Hi [name], so you are [age] years old!</h1>
     </body>
   </html>
  }
)
```

If this servlet is accessed by visiting the "multiEcho.servlet" URL with the following parameter string:

```
  ?name=Tim&age=46&color=lightgreen
```

Then the browser will return a lightgreen page with an h1 heading containing the text

```
  Hi Tim, so you are 46 years old!
```

Note that the order in which the parameters appear in the URL or in the parameter list of the servlet does not matter.

4.8 Processing numbers using servlets

Let us now revisit and extend some of the first servlets we considered above. For example, lets generalize the hypotenuse servlet to handle any triangle provided we enter the lengths of the two other sides.

One way to write this servlet is the following:

```
; hypotenuse.servlet
(servlet (a b)
  (case (a)
    ((#null) {Enter data by adding "?a=3&b=4" to the URL})
    (else {The right triangle with sides of length [a] and [b],
           has a hypotenuse of length
            [(sqrt (+ (* a a) (* b b)))]})))
```

The servlet parameters `a` and `b` must be numbers or else the servlet will generate an error! We will discuss methods for dealing with user input errors in a later section on "error checking". For now, we assume that the user will follow instructions and enter numbers when they are expected.

Exercise 8 *Write a servlet "sumtoN.servlet" which finds the sum of the numbers from 1 to N using the formula*

$$1 + 2 + 3 + \ldots + N = N * (N + 1)/2$$

and use it to find the sum from 1 to 100 (You should get 5050).

Exercise 9 *Write a servlet "tip.servlet" which calculates the tip required for a meal assuming that you tip at the 15% rate.*

4.9 Giving names to values using `let*`

When processing complex formulas it is often helpful to break the formula into pieces and to give names to each individual piece. For example, the body mass index provides a rough measure of whether your are overweight. The formula consists of dividing your weight in kilograms by the square of your height in meters. The following servlet computes your BMI by first converting your weight in pounds to your weight in kilograms, and your height in inches to your height in meters, then it applies the formula:

```
; BMI.servlet
(servlet (w h)
  (case w
    ((#null) {Enter data using ?w=170&h=68})
    (else
      (let* (
             (k   (/ w 2.2))  ; weight in kilograms
             (m (/ h 39.36))  ; height in meters
             (bmi (/ k (* m m)))
            )
       {If your weight is [w] pounds <br>
        and your height is [h] inches,
        <br> Then your BMI is [bmi]. <br>
        Note: a BMI over 30 is considered
        medically dangerous}))))
```

The `let*` expression above provides temporary names `k`, `m`, and `bmi` for the computed values. The general form of a `let*` statement is

```
  (let* (
          (var1 Expr1)
          (var2 Expr2)
           ...
          (varn Exprn)
        )
    Expr)
```

where the Exprs are evaluated and stored in the corresponding variables var1,.... These variables are then used in the final Expr to compute the value returned by the let* expression. The key points to note here are that each variable can be used to define the values of the following variables, but none of the variables has a value outside of the let*. Note: there is also a let expression which is almost exactly the same, except that the defined variables cannot depend on each other.

Exercise 10 *Write a servlet that uses a* let* *expression to compute the area of a triangle whose sides are a, b, and c using the following formula:*

$$A = \sqrt{(s-a)(s-b)(s-c)s}$$

where $s = (a+b+c)/2$ *is the semiperimeter. You should try this with* $a = 3$, $b = 4$, *and* $c = 5$. *These are the sides of a 3-4-5 right triangle and so the areas should be half of base times width, which is* $0.5 * 3 * 4 = 6$.

4.10 The if form and conditional execution

As a final example of these simple servlets, we now consider a liquor test servlet which determines whether you can buy liquor in Massachusetts.

```
; liquorTest.servlet
(case (age)
  ((#null) {Enter your age by appending ?age=19 to the URL})
  (else
    (if (< age 21)
        {Sorry, you have to wait [(- 21 age)] years before
         you can buy alcohol here.}
        {You've been able to buy liquor here for [(- age 21)]
         years})))
```

This servlet uses the (if TEST THEN ELSE form to test whether the buyer's age is 21 or higher. This form first evaluates the TEST. If the result is false, it then evaluates the ELSE code, otherwise it evaluates the THEN code. This example also introduces the comparison operator <. Scheme has a rich set of operators for creating tests. The numeric comparison operators you can use are

```
(<  a b)   -- true if a is less than b
(<= a b)   -- true if a is less than or equal to b
(>  a b)   -- true if a is greater than b
(>= a b)   -- true if a is greater than or equal to b
(=  a b)   -- true if a is equal to b
(!= a b)   -- true if a is not equal to b
```

In addition, the arguments a and b can be arithmetic expressions (like (/ age 10)).

You can also combine tests using the and, or, and not operators, e.g. to test if someone is "college age" you could use the following expression:

```
(and (<= age 22) (>= age 16))
```

To test whether someone is "not" college age, you could use either

```
(not (and (<= age 22) (>= age 16)))
```

or

```
(or (> age 22) (< age 16))
```

which are equivalent for any numeric value of `age`.

Exercise 11 *What is the result of evaluating the following Scheme expressions:*

```
(if (< 5 (* 2 4))
    (* 23 20)
    (- 18 5))

(if (> (+ 23 9) (* 6 5))
    {Cool! [(+ 23 9)] rules!}
    {Hmmm, [(* 6 5)] is biggest})
```

Exercise 12 *Write a servlet that determines whether someone is a senior citizen by testing if their age is at least 60.*

4.11 The `cond` form and multiple tests

Sometimes one wants to combine several tests and do something different for each of the possible test results. For example, one might want to classify someone as a child, teenager, adult, or senior citizen based on their age. The simplest way to do this is using the `cond` form, as in the following servlet

```
; ageClassifier.servlet
(servlet (age)
  (cond
    ((< age 13) "child")
    ((< age 20) "teenager")
    ((< age 60) "adult")
    (else "senior citizen")))
```

The `cond` expression consists of several "clauses" each of which begins with a test. If the test is true, then the rest of the clause is evaluated. Otherwise, testing continues with the next clause. The very last clause should always be an "else" clause which applies to all remaining cases. The tests can be simple comparisons as shown above, or can be very complex expressions.

Exercise 13 *Write a servlet which computes the tip given the cost of the meal and a number between 0.0 and 10.0 representing the quality of service. You should use a cond to determine whether to leave no tip, a 5%, 10%, 15%, or 20% tip based on the service number.*

4.12 HTML Forms and Servlet Parameters

So far we have been forced to pass parameters to a servlet explicitly by encoding them in a parameter string which we then append to the servlet's URL. In this section we show how to use the `form` tags in HTML to do this for us.

HTML forms provide a mechanism for soliciting information from the user and sending it to the server. Lets first look at a simple example. The following servlet generates an HTML form the first time the user visits a page. Once the user fills out the form and presses "submit," the data is sent back to the same servlet which then processes the data. In this case it just computes the age of the user in millions of seconds!

```
; ageCalc.servlet
(servlet (age)
  (case age
    ((#null)
       {<html><head><title>Age Calculator Form</title></head>
         <body>
           <form method="get" action="ageCalc.servlet">
             Enter your age in years:
               <input type="text" name="age"><br>
             <input type="submit">
           </form>
         </body></html>})
    (else
      {<html><head><title>Age Calculator Form</title></head>
         <body>
          If you are [age] years old, then <br>
          Your have lived for
           [(* age 365.25 24 60 60 0.000001)]
          million seconds
          </body></html>})))
```

Try this servlet yourself. At age 31 one is about a billion seconds old. The `form` element contains attributes that specify which "server" the information should be sent to (`action`) and how the information should be sent (`method`). The method can be either `get` or `post`. The `get` method encodes the arguments into a parameter string which is then attached to the URL, just as we have done by hand in the previous sections. The `post` method passes the parameters using another method in which they do not appear on the URL. This is safer (as the parameters are not visible on server logs) and allows for larger parameters than is possible with the `get` method. The action specifies where the data should be sent.

Exercise 14 *Modify the ageCalc servlet so that it also gets the user's name, and uses it in the response page.*

There are three main HTML elements for getting input from the user: `input`, `select`, and `textarea`. Each of these elements must have a `name` parameter which gives a name to the data that is sent to the server. The server will then use this name to determine how to handle the data.

The `input` element The input element has several variants including checkboxes, textfields, and file browers. We describe the most common variants below. All of these variants have the following form:

```
<input type=TYPE name="NAME" value="VALUE" size=SIZE>
```

Observe that there is no close tag for the `input` element. The `type` attribute can have one of the following values: `text, password, checkbox, radio, submit, reset, file, hidden, image, button`.

The `text` and `password` variants create a textfield in which the user can enter characters. The initial size is SIZE characters and the textfield is initialized with the string of characters in "VALUE". The password variant displays asterisks for each character typed by the user.

The `checkbox` and `radio` variants create checkable buttons. If a button is checked, then its "VALUE" will be sent to the server. Several buttons can have the same name, in which case the server will receive several values for that name. The `radio` variant allows the user to check at most one of the buttons that share the same name.

The `submit` variant sends the data to the server when it is pressed and the `reset` variant sets all fields in the form to their initial values.

The last four are more specialized and won't be discussed in detail here: `file` is used for selecting a file on the users disk, `hidden` is an element that doesn't appear on the webpage (but still specifies a value to be sent to the server), `image` specifies a graphical submit button, and `button` is used for client-side scripting, which we do not discuss here.

The `textarea` element This element is used for soliciting multiline input from the user, it has the form

```
<textarea name=NAME rows=ROWS cols=COLS>
  initial text goes
  here
</textarea>
```

In addition to the `name`, you must specify the size of the textarea in rows and columns.

The `select` element The select element is used to provide a fixed list of choices from which the user must choose. It has the form:

```
<select name=NAME size=SIZE multiple>
  <option value=V1> A1</option>
```

```
   <option value=V2> A2</option>
   ...
   <option value=Vn> An</option>
 </select>
```

If the `multiple` keyword is present in the attribute list of the `select` element, then the user is allowed to select several of the options simultaenously; otherwise, the user can only select one. The `value` attributes in the `option` elements are sent to the server if the item is selected. If there is no `value` attribute then the text between the `option` tags is sent in its place.

Exercise 15 *Rewrite all of the examples given in the earlier sections of this Chapter using HTML forms to get the information from the user instead of requiring the user to explicitly encode the information in the URL.*

Exercise 16 *Write a servlet which first generates a form containing all of the form tags given above. The servlet should echo back all of the information collected (and may add some commentary if you want).*

4.13 Summary of Scheme Syntax

Figure 4.1 hows all of the Scheme expressions we have seen in this chapter and Figure 4.2 shows the Scheme procedures we have encountered thus far.

S-expressions where the `Ei` are numbers, strings, or S-expressions.

```
(E0 E1 ... Ek)
```

Quasi-strings where `Ei` are S-expressions

```
{....[E1].....[E2].....  ....[En]....}
```

Servlets where `E` is an S-expression

```
(servlet (a b ...)
  E)
```

Case-based execution where the `Ci` are constants and the `Ei` are expressions.

```
(case E0
  ((C1) E1)
  ((C2 C3 C4 C5) E2)
   ...
  (else E))
```

Simple tests where `TEST, THEN, ELSE` are expressions.

```
(if TEST THEN ELSE)
```

Multiple tests where `Ti` are tests and the `Ei` are expressions.

```
(cond
  (T1 E1)
  (T2 E2)
  ...
  (else E))
```

Local Variable Binding where the `Vi` are variables and the `Ei` are expressions.

```
(let* ((V1 E1)
       (V2 E2)
       ...
       (Vn En))
   E)
```

Figure 4.1: Summary of Scheme Syntax

Logical operators:

```
(and T1 T2 ...)
(or T1 T2 ...)
(not T)
```

Arithmetic operators:

```
(+ a b ...) (- a b) (* a b ...) (/ a b)
(sqrt a) (exp a) (log a) (sin a) (cos a) (tan a)
(asin a) (acos a) (atan a) (round a)
```

Comparison operators:

```
(< a b) (> a b) (<- a b) (>- a b) (= a b) (!= a b)
(equal? a b)
```

Java procedures:

```
(Date.)
(Math.random)
(.getHeader request "....")
(.getRemoteAddr request)
(.sendRedirect response URL)
```

Figure 4.2: Scheme procedures seen so far

Chapter 5

Examples of Scheme Servlets

In this chapter we present several, increasingly more interesting, examples of Scheme servlets. Each servlet will introduce a few more concepts about Scheme servlets. Hopefully, by the end of the chapter you will begin to understand how to create your own Scheme servlets.

5.1 Password-protected pages

It is often useful to restrict the audience that has access to any given webpage. One way to restrict access is to require the users to enter a password before they can gain access to any other pages on the site. In this first example, we show how to create a simple password protected page that uses one password for all users. A more interesting application would allow users to register and select their own password, but we will get to that later.

The servlet is Figure5.1 shows our servlet. The first time a user visits this page, there will be no form data (and the "pw" variable will have the value `#null`, which is a Scheme constant representing an empty object. In this case, a form will be generated. If the user fills out the form and presses the submit button, the servlet will then read an non-null value from the "pw" parameter. If the password is correct, the secret page will be displayed otherwise, the user will be prompted for the correct password again!

Security Warning – note this servlet is not very secure because the source code contains the password. Generally, the source code will not be visible to the outside world, but if you make a backup copy of your servlet and store it in some file that doesn't end in ".servlet" (say ".txt" or `".servlet~"`) then the server will assume that the file is just text and will let anyone see the contents of this file. It is very common for editors to make exactly this kind of a backup file and hence by editting a file in your servers webapp folder, you could let everyone see the password!

```
; pwpage.servlet
(servlet (pw)
  (case pw
     ((#null)
       {<html><head><title>Password Form</title></head>
         <body>
           <h1>Please enter the correct password below</h1>
           [(java.util.Date.)]
           <br>
           <form method="post" action="pwpage.servlet">
             Please enter the password for this page:
             <input type="password" name="pw" size=20>
            <br><input type="submit">
           </form>
         </body>
        </html>})

    (("yes! scheme")
       {<html><head><title>Secret Page</title></head>
         <body>
           <h1 style="font: bold 24pt Times,serif">
             Welcome to the secret page
           </h1>
            The door combination is 6-20-5
         </body>
        </html>})

     (else
       {<html><head><title>Password Form</title></head>
         <body>
           <h1>Wrong password, go back and try again</h1>
         </body></html>})))
  ))]
```

Figure 5.1: A simple Password Protected Page

There are various techniques for getting around this, but the general lesson is that passwords are not helpful unless they are very carefully hidden! We will discuss security in more detail later.

There are other ways of restricting access beside asking for a password. Restricting access to certain IP addresses is a common way of creating an "intranet"-like interface for your servlet. For example, Brandeis University "owns" all IP addresses have the form

```
129.64.*.*
```

where the two asterisks represent any integers between 0 and 255. One can test whether the user's IP address is from Brandeis using the following test:

```
(.startsWith (.getRemoteAddr request) "129.64.")
```

Similarly, one can restrict access to computers running the MacIntosh Operating System or to all IP address except a few "forbidden" ones.

5.2 Counters and Files

Next we show how to create servlets that read and write to files on the server. Some of the simple applications this enables are servlets that contain counters, and servlets that allow you to leave a message.

We first consider the problem of implementing a servlet that keeps track of how often it has been visited. There are many strategies one can employ for such counters. For this example, we will write a servlet "counter.servlet" that stores its count in a file called counter.servlet_count.

The code for the "counter.servlet" servlet is in Figure 5.2. This servlet introduces a few new ideas. First of all, it relies on a small library "files.scm" of scheme procedures for reading and writing to the count file. We assume that this file is loaded when the server is first started up.

The servlet must first load the library before it can use the special counter library procedures. This is done using the "load" procedures

```
(load "webapps/scheme/lib/forms.scm")
```

Also, since the servlet now does two things:

- update a counter and
- generate a webpage,

we must use a feature of the `let*` expression that we haven't mentioned so far – the ability to do several things before computing the value to be returned by the expression. The general form of the `let*` is

```
(let* ((V1 E1)
       (V2 E2)
       ...
```

```
; counter.servlet
(let ((count
         (+ 1
           (read-from-file (servlet-file request "count") 0))))

    (write-to-file (servlet-file request "count") count)

    {<html><head><title>counter</title></head>
      <body style="background:white; color:black">
        <h1> You are visitor number [count]</h1>
      </body> </html>}
)
```

Figure 5.2: A simple counter program

```
      (Vn En))
  F1
  F2
  ...
  F
)
```

where the `Ei` are evaluated first and their values stored temporarily the variables `Vi`, then the expressions `Fi` are evaluated and finally the value of expression `F` is returned.[1] The only reason we would evaluated the `Fi` is that they have some `side-effect` like writing something into a file on the disk or sending an email message, or updating a database, etc.

This servlet illustrates the use of three new procedures:

- `(read-from-file FILENAME DEFAULT-VALUE)` this reads a scheme terms from the specified FILENAME, and if no such file exists then it returns the DEFAULT-VALUE instead.
- `(write-to-file FILENAME VALUE)` this writes the specified VALUE to the specified FILE, and if the FILE does not already exist then it creates it.
- `(servlet-file request "XYZ")` – this returns the file-path of the current servlet with the suffix "_XYZ" appended to it. This file-path can be used to read or write to this new file. The advantage of this is that the servlet can be moved to another place on the server and will create a use a counter in the same directory! This is very similar to the advantages of using relative file names rather than explicit URLs when writing websites.

[1] Actually, one can have several expressions between `Vi` and `Ei` as well, and these expressions are evaluated sequentially.

```
; this is in the file "logger.servlet"
(let ((logdata (read-string-from-file
                  (servlet-file request "log")
                  ""))
      (current-entry {[(Date.)]: [(.getRemoteAddr request)]}))

  (append-to-file (servlet-file request "log") current-entry)

  {<html><head><title>logger</title></head>
      <body style="background:white; color:black">
        <h1> The list of previous visitors to this site is </h1>
        <pre>[logdata]</pre>
        <br>Your entry is <pre>[current-entry]</pre>
      </body>
     </html>}
)
```

Figure 5.3: A simple log servlet

5.3 log files

In this next example, we show how to create a servlet that generates an entry in a log file for each time the servlet is visited. The logger servlet in Figure 5.3 is quite similar to the counter servlet above.

We begin by appending the current date and time to the logfile ”

```
logger.servlet_log
```

Instead of “writing” to the counter file, we “append” to the log file. Then we read the logfile and store it in the ”logdata” variable which we include in the HTML file sent back to the user.

This servlet uses two new procedures for reading/writing files:

- `(read-string-from-file FILE DEFAULT-VALUE)` – this reads the entire contents of the file as a string of characters, and if the file does not yet exist it returns the DEFAULT-VALUE instead.
- `(append-to-file FILE VALUE)` – this appends the specified VALUE to the next line of the specified FILE. If the file does not already exist then it creates it.

Also note that the `current-entry` is created using quasi-string notation. We could just as easily have implemented it using `string-append` as follows

```
(string-append (Date.) ": " (.getRemoteAddr request))
```

which may even be easier to read in this example.

```
{<html>
    <head><title>table demo</title> </head>
    <body style="background:blue; color:black;
                 text-align:center">
        <h1> Office hours</h1>
        <table cellpadding=10 cellspacing=5 border=5
               style="background:white">
        [(trs "" '(
          ( time  mon   tue   wed   thu  fri  sat  sun)
          ( 9     th    -     -     th   -    -    -  )
          (10     -     aj    -     -    -    -    -  )
          (11     -     aj    -     -    -    -    -  )
          (12     -     aj    -     -    -    -    -  )
          ( 1     -     -     -     -    -    ef   ef )
          ( 2     -     -     -     -    -    ef   ef )
          ( 3     -     -     -     -    -    -    -  )
          ( 4     -     -     rs    rs   rs   -    -  )
         ))]
       </table>
      </body></html>}
```

Figure 5.4: A simple table servlet

5.4 Scheme Tables and Lists

Next we give an example of using Scheme to create HTML tables. This will turn out to be quite useful when we are working with databases, as the answer to a database query often takes the form of a table. Figure 5.4 gives an example of creating an HTML table from a list using the `trs` procedurfe from the "tables.scm" library (which we assume has been preloaded by the server).

The new Scheme idea we need to use here is the notion of a list which is represented in Scheme by placing a single quote (') in front of an S-expression. The effect is for Scheme to treat the S-expression as "data" and, in particular, not to try to evaluate it.

One of the most powerful features of Scheme is that it allows you to work in this way with lists of data in a relatively simple way. You can create a list of the weekdays by putting a single quote in front of a parenthesized sequence of symbols:

```
'(mon tue wed thu fri)
```

The single quote indicates to the Scheme interpreter that the following term is just "data" and should not be "evaluated. " For example,

```
(+ 1 2 3)
```

evaluates to 6, but

```
  '(+ 1 2 3)
```

evaluates to the list `(+ 1 2 3)`.

Exercise 17 *Create the following servlet*

```
; thisisalist.servlet
'(string-append "this" "is" "a" "list")
```

and see what happens when you visit this servlet with and without the initial single quote ('). Try other S-expressions with and without quotes. In fact, you can put a quote in front of any of the previous servlets you've created and see what happens. Try to write a simple explanation for what is going on.

Scheme lists can contain sublists and this is a convenient way of representing a table of data, e.g.

```
'(
  (name age sex)
  (john 22  male)
  (jiri 20  male)
  (anzy 18  female)
  (miri 17  female)
 )
```

To make this into a table we use the "trs" procedure defined in the "tables.scm" library. This procedure takes a list of lists (one for each row) and creates a "tr" element for each of these lists.

5.5 Lists and Quasi-quoted lists

You can also create lists using the `list` procedure, e.g.

```
(list
  (list {Hello there} (+ 1 2 3 4 5))
  (list {Goodbye} (* 1 2 3 4 5)))
```

Note that this expression evaluates to

```
'(("Hello there" 15) ("Goodbye" 120))
```

You can also have the same effect using a back quote "`" instead of a single quote ('), and then putting a comma (,) in front of those expression you want to be evaluated. This is called quasi-quote notation. In our case, we could write the previous example as:

```
`(("Hello there"  ,(+ 1 2 3 4 5))
  ("Goodbye"      ,(* 1 2 3 4 5)))
```

5.6 Automated Email

Our final example of basic servlets demonstrates how to write a servlet that sends email. Again, here we assume that the library "mail.scm" has been preloaded. The code for this library is in the appendix. This library defines the `send-mail` procedure which is called as follows:

```
(send-mail request TO FROM SUBJECT TEXT)
```

Here TO, FROM, SUBJECT, and TEXT are strings.

For example, to send mail to jj@xyyaz.com from "me@uzw.com" you would write

```
(send-mail
   request
   "jj@xyyaz.com"
   "me@uzw.com"
   "This is a test"
   { Cool, this really works and I can include
        scheme expressions like this:
      [(* 111 111)] is 111 squared! })
```

The "to" and "from" email addresses can come from the servlet parameters (for sending confirmation email for example). You should avoid using "fake" return addresses as this is annoying and the email can often be easily traced back to you anyway!

The structure of a servlet that generates a form and then uses the form data to send several emails is as follows:

```
(servlet (user-email a b c)
  (case user-email
    ((#null) {... generate form to get email, a, b, c
                  from user and send it back to this servlet})
    (else
       ; first we send email to the user and the owner
       (send-mail request email "OWNER@XYYAZ.COM"
          "Thanks" {Thanks for visiting XYYAZ at [(Date.)]})
       (send-mail request "OWNER@XYYAZ.COM" SUBJECT
         {date="[(Date.)]\n email=[email]
           a=[a] \n b=[b]\n c=[c]})
       ; finally we send back a confirmation webpage
       {Thanks for visiting XYYAZ.com})))
```

The key point here is that the confirmation webpage must be the last expression in the "else" clause, as this is what the server will return to the user after evaluating the else clause of the case.

```
; SurveyMail.servlet
(servlet (email birthyear favcol)
  (case email
    ((#null)
     {<html><head><title>Survey Input Form</title></head>
       <body>
        <h1>Please enter the data listed below</h1>
        <form method="post" action="SurveyMail.servlet">
        <table border=5 cellpadding=5 cellspacing=5>
         [(trs
           (list
             (list {Your email address}
                   {<input type="text" name="email">})
             (list {Your birthyear}
                   {<input type="text" name="birthyear">})
             (list {Your favorite color}
                   {<input type="text" name="favcol">})))]
         </table>
         <input type="submit">
       </form>
      </body></html>})

  (else
    (send-mail request "tjhickey@cs.brandeis.edu" email
       "survey results"
        (list (java.util.Date.) email birthyear favcol))
    ;; and finally generate the survey confirmation
    {<html><head><title>email survey</title></head>
      <body style="background:white; color:black">
        <h1> Thanks [email]<br>
             for completing the survey</h1>
      </body></html>})))
```

Figure 5.5: An Email survey servlet

5.7 Redundancy, Refactoring and Abstraction

One of the weak points of HTML as a language is that it provides only limited means for addressing the problems of redundancy in web pages. An example will help illustrate this problem. Consider the HTML in Figure 5.6 which generates a web page listing posters for sale. Each poster has an associate image file, a title, and a price. The webpage has a fairly complex collection of HTML for each poster in which these three elements are embedded (and the tax and total price also appear in the HTML for each poster). Only the first two posters are shown in this HTML, but one can imagine a website with hundreds of posters for sale, all using the same "template."

With cut-and-paste technology, it is fairly easy to add a new poster to the page. Just copy the HTML for an existing poster, calculate the tax and total price for the new poster, and change the five fields in the HTML. There are two problems with this approach. First, the need to do so much cutting and pasting creates opportunities for making mistakes (e.g., forgetting to close HTML tags, miscalculating the tax, etc.). Secondly, and more importantly. If one want to change the way in which the posters are presented, one must change each of the poster entries by hand. We call this the redundancy problem. If you have redundancy in your HTML, then you might make copying errors, and if you need to change the HTML, you may have to change all the redundant code (again opening up opportunities for copying errors).

The CSS technology provides some relief for this problem if you are only interested in changing the style – rather than include the style directly in the HTML for each poster, one adds `class` attributes to the tags and then imports a CSS page to define the style. If one must make structural changes, the CSS appproach offers no help (e.g. if the tax rate changes or if one wants the images in the center instead of the left).

In this section, we introduce the notion of abstraction as a way of handling the redundancy problem. The idea is to create a "template" which contains all of the HTML for each poster, except that the values that change are replaced by variables. One then defines this template at the beginning of the servlet and then uses the template to generate the HTML for each poster. This template is called an `abstraction` of the original HTML, and the process of going from a redundant webpage one that uses an abstraction is called `refactoring`.

Lets walk through the process for this particular example. First, we define a procedure `poster` which abstracts the HTML for each poster entry in Figure 5.6 using three parameters `name`, `imagefile`, and `price`:

```
(define (poster name imagefile price)
  {<div style="background:rgb(0,150,150)">
    <table  width="100%"><tr><td>
     <a href="[imagefile]">
      <img src="[imagefile]" alt="[name]" width="150">
     </a><br>
    </td><td>
```

```
<html><body style="background:white; color:red">
  <h1> Posters for Sale</h1>

  <div style="background:rgb(0,150,150)">
    <table  width="100%"><tr><td>
     <a href="dew.jpg">
      <img src="dew.jpg" alt="The Dew" width="150">
     </a><br>
    </td><td>
    <h1 style="background:lightgreen">The Dew</h1>
     </td><td>
    <ul><li>Cost: $15.00</li>
        <li>Tax: $0.75</li>
        <li>Total: $15.75</li>
    </ul>
    </tr></table>
   </div>
   <br>   <br>   <br>

   <div style="background:rgb(0,150,150)">
    <table  width="100%"><tr><td>
     <a href="foil.jpg">
      <img src="foil.jpg" alt="Abstract #17" width="150">
     </a><br>
    </td><td>
    <h1 style="background:lightgreen">Abstract #17</h1>
     </td><td>
    <ul><li>Cost: $13.5</li>
        <li>Tax: $0.675</li>
        <li>Total: $14.175</li>
    </ul>
    </tr></table>
   </div>
   <br>   <br>   <br>

   ....

   </body></html>
```

Figure 5.6: A Posters Sale site

```
    <h1 style="background:lightgreen">[name]</h1>
     </td><td>
    <ul><li>Cost: $[price]</li>
        <li>Tax: $[(* price 0.05)]</li>
        <li>Total: $[(* price 1.05)]</li>
    </ul>
    </tr></table>
   </div>
   <br>   <br>   <br>
  })
```

After this procedure has been defined, it can be invoked by supplying the desired values for the three parameters. The server uses these values to generate the HTML in the template, which is then inserted into the page. For example, the full refactored servlet is shown in Figure 5.7.

One can actually make one more refactoring by using the `map` procedure discussed earlier. The idea is to use the `apply` procedure which allows us to apply the `poster` procedure to the list of its arguments, hence we can rewrite the body of the Posters page as

```
(servlet ()
  (define (poster ...) ...)

  (define (make-poster L) (apply poster L))

{<html><body style="background:white; color:red">
  <h1> Posters for Sale</h1>
  [(map make-poster
   '(
     ("The Dew" "dew.jpg" 15.00)
     ("Abstract #17"  "foil.jpg" 13.50)
     ("Childhood Remembered"   "sueno.jpg"  20.00)
     ("A Cat's Eye ViewTable"   "table.jpg"  16.00)
     ("A Covey of Toys"   "toys.jpg"  13.00)
     ...))
   ]
   </body></html>}
```

The advantage of this final refactoring is that one can then store the poster data in a separate file (or a database) and thereby completely separate the HTML from the data.

```
(servlet ()
 ; insert some code here ....
 (define (poster name imagefile cost)
  {<div style="background:rgb(0,150,150)">
    <table  width="100%"><tr><td>
     <a href="[imagefile]">
      <img src="[imagefile]" alt="[name]" width="150">
     </a><br>
    </td><td>
    <h1 style="background:lightgreen">[name]</h1>
     </td><td>
    <ul><li>Cost: $[cost]</li>
        <li>Tax: $[(* cost 0.05)]</li>
        <li>Total: $[(* cost 1.05)]</li>
    </ul>
    </tr></table>
   </div>
   <br>   <br>   <br>
  })

 {<html><body style="background:white; color:red">
  <h1> Posters for Sale</h1>
  [
     (poster "The Dew" "dew.jpg" 15.00)
     (poster "Abstract #17"  "foil.jpg" 13.50)
     (poster "Childhood Remembered"   "sueno.jpg"  20.00)
     (poster "A Cat's Eye ViewTable"   "table.jpg"  16.00)
     (poster "A Covey of Toys"   "toys.jpg"  13.00)
     ...
   ]
   </body></html>}
)
```

Figure 5.7: A Posters Sale site

Part III

Reactive GUI Programming

Chapter 6

Graphical User Interface Design in Scheme

In the previous chapters we have been concerned with server-side web programming in which the client interacts with the server using a browser. We used HTML with CSS to create the Graphical User Interface (GUI). The browser and the server communicate using the HTTP protocol, the responses to the users input are specified using Scheme.

In this section we consider another approach to web programming in which the Graphical User Interface is provided by a Scheme program rather than the browser. The communication is done by sending Scheme expressions back and forth between clients, and the responses to user input and to communication input is specified using Scheme.

Web programming, as we discuss in the next few chapters, provides a more interactive style of programming than is possible with web pages and servlets. We will focus on writing "reactive" programs in which each user interaction triggers a response consisting of some fixed number of operations. This paradigm includes a large number of interesting examples.

6.1 Intro to the JLIB toolkits

We begin with a introduction to a simple toolkit for building graphical user interfaces (i.e. windows, buttons, menus, etc.) The applets we consider will pop up one or more windows and allow the user to enter data, press buttons, connect to databases on the server, chat with other users, etc.

For example, the program in Figure 6.1 is stored in a file **"hello.applet"** and if you visit this file with your browser. You will see a page that contains the five lines of comments describing the program and its author, and your browser is Java-enabled, it will pop up a window with the word "greetings" at the top in red letters with a Helvetica Bold 60pt font, and a button labelled "goodbye" beneath. When you click on the button, the window disappears.

```
"Greeting"
"Tim Hickey"
"http://www.cs.brandeis.edu/~tim"
"This pops up a greeting window"
"http://www.cs.brandeis.edu/~tim/hickey.new.gif"

(jlib.JLIB.load)
(define win
  (window "hello"
   (col
    (label "Greetings"
         red  (HelveticaBold 60)
    (button
        "goodbye"
        (action (lambda(e) (.hide win))))))))
(.pack win)
(.show win)
```

Figure 6.1: hello.applet

If you change the filename from **hello.applet** to **hello.snlp**, and if you have installed the Java Web Start plug-in[1] then the browser will download the program as a Java Web Start application and it will again pop-up a window as before. The difference with Java Web Start is that the next time you visit that link, the browser will only download the program if there has been a change in the code. If not, then it will use the version it has stored on your disk. Moreover, you can disconnect your computer from the net and still run the stored copy of the program.

The program if Figure 6.1 illustrates a number of features of the JLIB toolkit. First, observe that it defines three components:

- a `label`
- a `button`, and
- a `window`.

Each of these components has a string written somewhere on it ("Greetings", "goodbye", and "hello" respectively). The `label` has also been given a specified color and font, and the `button` has been given an action. The relative position of the `label` and `button` has been specified by the `col` layout procedure, which states that the `label` and `button` should appear in a column. Finally, the `action` on the `button` causes the `window` to disappear when the `button` is pushed. The last two commands are `(.pack win)` which does the window layout

[1] `http://java.sun.com/products/javawebstart`

and determines the precise minimum size of the window so that everything just fits inside it. The `(.show win)` command makes the window appear.

Although this little program does quite a bit (requiring a paragraph to decide), the code for the program is relatively concise. Each word that appears in the program has a particular effect (except for the `lambda(e)` which we will explain later). By replacing `col` with `row` we would get a horizontal layout, and by replacing `label` with `button` our greeting would be able to take an action.

6.2 Overview of the JLIB toolkits

The key idea of the JLIB toolkit is to use the flexibility and expressiveness of Scheme to create a GUI-building library which allows complex GUIs to be built by evaluating a simple and declarative expression whose structure closely resembles the structure of the GUI itself.

The JLIB model is based on five fundamental concepts:

- COMPONENTS – there are a small number of ways to construct basic components (buttons, windows, ...)
- LAYOUTS – there are a small number of ways to layout basic components (row, col, table, grid, ...)
- ACTIONS – there is a simple mechanisms for associating an action to a component
- PROPERTIES – there are easy ways for setting the font and color of components
- TAGS – this is a mechanism for giving names to components while they are being laid out.

Another key idea is that operations on all components should be as uniform as possible. For example, there are procedures "readstring" and "writestring" which allow one to read a "string" from a component, and write a string onto a component. Thus "writestring" can change the string on a label, a button, a textfield, a textarea. It can also change the title of a window or add an item to a choice component. Likewise, readstring returns the label of a button, the text in a textarea or textfield, the text of the currently selected item in a choice, the title of a window, and the text of a label.

Similarly, JLIB is designed so that the component and layout constructors require a few initial arguments of specified types, followed by many optional arguments which may apper in any order and whose types are used to determine their effect.

For example, a font object will change the font of the component, a color object will change the background color, a Listener object will add an action listener to the component, a string will generally become the label of the object.

We introduce each of these components in turn. As we go through these examples you should try them out yourself in a Jscheme interpreter window.

Feel free to write variations of the programs we show here. This will help you learn how to "think" in Scheme. Programming is a skill that is best learned by doing rather than by reading alone.

6.3 Windows

Window are the most basic component of Graphical User Interfaces. All other components must appear inside a window and any particular GUI might use several different windows (e.g. help windows, "about" windows, file selection windows, warning windows, editting windows).

The most basic window contains just a title (which is a string that usually appears in the title bar). The following code creates a window with the title "Hello", sets is size to be 300 pixels wide by 400 pixels high, and makes it appear on the screen:

```
(define w (window "Hello"))
(.resize w 300 400)
(.show w)
```

Note that we have used the "define" syntax to give the name "w" to the window. This name is need when we resize the window and make it visible in the following two expressions. We will see many more windows below, so we don't dwell any further on them at this point.

6.4 Labels

A label is a component used to display some text on the GUI. For example, the following program creates a window with the text "Hello" in the titlebar and the phrase "Hello again" in the inside of the window:

```
(define w (window "Hello"  (label "Hello again")))
(.resize w 300 400)
(.show w)
```

The first argument of the "label" procedure must be the string that will appear on the label. We will see below that one can also specify the font and the color of a label by adding more arguments to the "label" procedure.

6.5 Fonts

There are nine built-in fonts that you can use in the JLIB library, and for each font you can pick any font size (as long as it is a positive whole number). These fonts are:

```
(Helvetica 23)      (HelveticaBold 100)   (HelveticaItalic 12)
(TimesRoman 29)     (TimesRomanBold 88)   (TimesRomanItalic 9)
(Courier 33)        (CourierBold 32)      (CourierItalic 33)
```

For example, the following program extends our previous example, by specifying that the text on the label should be displayed in a 60 point bold-face Helvetica font:

```
(define w
   (window "Hello"
      (label "Hello again" (HelveticaBold 60))))
(.pack w)
(.show w)
```

The `(.pack w)` expression changes the size of the window so that it is just large enough to fit the label.

One can also specify other fonts using the `Font.` procedure. This requires you to specify the name of the font, the style (bold, italic, plain), and the size (in points).

```
(define w
   (window "Hello"
      (label "Hello again"
        (Font. "New Century Schoolbook" Font.BOLD$ 32))))
(.pack w)
(.show w)
```

The style is specified by using one of the following style names:

```
Font.BOLD$
Font.Plain$
Font.Italic$
```

The name of the font is a string (enclosed in curly braces), and the point size is any positive integer. Note: this will cause an error if you ask for a font that the computer does not have access to.

6.6 Colors

There are several ways to specify colors. The simplest is to use one of the named colors. The current set of named colors for JLIB is

```
 white, lightGray gray darkGray black
red pink orange yellow green magentan cyan blue
```

You can also specify a color by specifying how much red, green, and blue light should be mixed to form the color:

```
(color RED GREEN BLUE)
```

where RED, GREEN, BLUE are numbers between 0 and 255. Note that this is additive color mixing which is quite different from the subtractive color mixing of paints. For example to get yellow you mix red and green, to get white you mix red, green, and blue. Some of the color mixes are shown below:

```
yellow      (color 255 255 0)
white       (color 255 255 255)
black       (color 0 0 0)
blue        (color 0 0 255)
lightblue   (color 200 200 255)
darkblue    (color 0 0 150)
```

You can set the background of a label (or any other component) just by including the color as one of the arguments of the label. Continuing with our running example, we can make the label yellow as follows:

```
(define w
   (window "Hello"
      (label "Hello again" (HelveticaBold 60) yellow)))
(.pack w)
(.show w)
```

6.7 Tabular Layouts

So far we have considered windows containing a single label. We now look at four procedures for specifying the layout of multiple components in a window. Although these layouts can be applied to any component, we will stick with labels for simplicity.

The first two layouts we discuss are the row and column layouts. The `row` procedure arranges its arguments horizontally across the window. The `col` procedures arranges its arguments vertically. For example, the following code produces a window with labels "red", "green", "yellow", and "blue" having the same color as their label, laid out horizontally across the window:

```
(define w
   (window "Hello"
     (row
      (label "red" red)
      (label "green" green)
      (label "yellow" yellow)
      (label "blue" blue)  )))
(.pack w)
(.show w)
```

By swapping "col" for "row", the labels would be arranged vertically.

The row and col procedures also take additional arguments which specify what should happen when the window is resized. One possibility is to have the components in the row stay the same size even though the window grows. On the otherhand, one might want the components in the row to expand as the window expands (the expansion can be horizontal, or vertical, or both). To specify this expansion property one adds one of the following to the row or col argument lists:

```
none
horizontal
vertical
both
```

If the components do not expand horizontally and vertically, the one must specify how they should be placed inside the row or col. There are several choices, listed below:

```
north northeast northwest east west
south southeast southwest center
```

So, we can modify the running example and have the labels in the row stay the same size, and be centered in the row, as follows:

```
(define w
   (window "Hello"
     (row 'none 'center white
      (label "red" red)
      (label "green" green)
      (label "yellow" yellow)
      (label "blue" blue)  )))
(.pack w)
(.show w)
```

Note that one must precede the specifiers with a single quote. Also observe that we can specify the color of the row – this is the color of the space between the components in the row.

One can also lay components out in a tabular form with R rows and C columns. There are two procedures for doing this:

```
(grid  R C ....)
(table R C ....)
```

The difference between them is that each of the cells in the grid is exactly the same size, where as in the table the cells may vary in size. Continuing with our example, we can make a table of four labels as follows:

```
(define w
   (window "Hello"
     (table 2 2
      (label "red" red)
      (label "green" green)
      (label "yellow" yellow)
      (label "blue" blue)  )))
(.pack w)
(.show w)
```

Likewise we could use a "grid" in place of "table" in this example. Try it and see how they differ.

There is one other major layout mechanism that is quite useful – the border layout. This layout allows one to put five components on a page. They can go in the north, south, east, west, and center. When the window is resized, the center component grows (or shrinks) horizontally and vertically, while the other components stay near the edges. For example, we could extend the previous example, by putting the table in the center of a border layout with labels in the four other positions:

```
(define w
   (window "Hello"
     (border
       (center
         (table 2 2
           (label "red" red)
           (label "green" green)
           (label "yellow" yellow)
           (label "blue" blue)  ))
      (north (label "north" white))
      (south (label "south" yellow))
      (east (label "east" green))
      (west (label "west" red))
    )))
(.pack w)
(.show w)
```

This example also shows how layouts can be "nested." Notice that we have put a table in the "center" of the border layout. Likewise, we could put a table or a column in the east or west of the border. Or we could put a grid inside one of the table entries.

Exercise 18 *Build several GUIs with colored labels as above. Try to create layouts that are nested several levels deep and see what happens when you expand and contract the window.*

6.8 Buttons

Buttons are similar to labels except that you can attach "actions" to them. These actions are carried out when the button is pushed. The following program creates a window with four buttons, but these buttons have no actions associated with them. WHen you press them they change color slightly to indicate that they have been pressed, but nothing else happens.

```
(define w
   (window "Hello"
     (row 'none 'center white
```

```
        (button "big" red)
        (button "small" green )
        (button "just right" yellow )
        (button "gone" blue )
     )))
(.pack w)
(.show w)
```

6.9 Actions

One of the most important arguments to components is the "action" argument which specifies what to do when the component is selected (e.g. when a button is pushed or a menuitem selected, or a choice selected, or a textfield value entered). The syntax for the action argument is:

```
(action (lambda (e) COMMANDS))
```

the "lambda" specifies that the "COMMANDS" are to be delayed until the action is performed. The "e" represent the kind of event that triggered the action. From the "e" variable we can get the time the action was triggered, which component it came from, where the mouse was (for a mouse event), etc.

```
(define w
   (window "Hello"
     (row 'none 'center white
      (button "big" red
          (action (lambda(e) (.resize w 500 500))))
      (button "small" green
          (action (lambda(e) (.resize w 10 10))))
      (button "just right" yellow
          (action (lambda(e) (.pack w))))
      (button "gone"
          (action (lambda(e) (.hide w))))
    )))
(.pack w)
(.show w)
```

Observe that we add an action to a button by adding a term of the following form to its argument list:

```
(action (lambda(e) .....))
```

where the "...." represents some scheme code that will be executed when the button is pushed.

6.10 Choice components

The `choice` procedure produces a component that allows the user to choose from a fixed set of choices. The choice components often will use a "case" expression in their action argument as this allows the program to take different actions based on the particular choice that was made. The first several arguments of the choice component are strings that the user is able to choose among. The remaining optional arguments are the color, font, and actions (in any order).

The following GUI lets the user make a choice of size for the window.

```
(define w
   (window "Hello"
       (choice "big" "small" "just right" "gone"
         (action (lambda(e)
           (case (readstring (.getSource e))
             (("big") (.resize win 500 500))
             (("small") (.resize win 10 10))
             (("just right") (.pack win))
             (("gone") (.hide win))
           ))))))
(.pack w)
(.show w)
```

This example shows one way of implementing an action that depends on the choice taken by the user. The tricky part is the

```
(readstring (.getSource e))
```

The "e" in this case is the "event" that occurs when the user selects a choice. The ".getSource" procedure determines the component that generated that event (in this case, the choice), and the "readstring" reads the string labelling that choice. The rest of the action is a "case" expression that selects the appropriate action based on the user's choice.

6.11 Tags and Naming

As we will see it is often useful to be able to give names to components (for example if we want to read from or write to a component, we must have a way of refering to it).

The "Tag" mechanism provides a simple way to name and refer to objects. A tagger is created using the following expression

```
(let ((t (maketagger)))
   EXPR)
```

This creates a tagger "t" which can be used in the following expression. To assign a name "abc" to an object "X" you use the syntax:

```
(t "abc" X)
```

to lookup the object whose name is "abc", you use the syntax

```
(t "abc")
```

For example, the choice example above could have been rewritten to avoid the user of `.getSource` as follows:

```
(define t (maketaqger))
(define w
   (window "Hello"
     (t "size"
       (choice "big" "small" "just right" "gone"
         (action (lambda(e)
           (case (readstring (t "size"))
             (("big") (.resize win 500 500))
             (("small") (.resize win 10 10))
             (("just right") (.pack win))
             (("gone") (.hide win))
           )))))))
(.pack w)
(.show w)
```

Notice that the `choice` expression is an argument of a call to the tagger. When the action is invoked, the expression `(t "size")` refers to the choice component itself and hence has the same effect as `(.getSource e)`. Yet another way to write this program is to give the choice component a name using the `define` syntax:

```
(define sizechooser
     (choice "big" "small" "just right" "gone"
         (action (lambda(e)
           (case (readstring sizechooser)
             (("big") (.resize win 500 500))
             (("small") (.resize win 10 10))
             (("just right") (.pack win))
             (("gone") (.hide win))
           )))))

  (define w
   (window "Hello"
     sizechooser))

(.pack w)
(.show w)
```

Observe that the name `sizechooser` is used both in the action, where we need to read the user's choice, and it is used in the window when we want to add the choice to the window.

Of the three versions shown above, the last has the disadvantage of requiring components to be defined outside of their context. Thus, when the `sizechooser` component is defined, you don't know where it will be appearing.

6.12 Textfields and Texareas

Textfields and Textareas provide places for users to write text. They are created as follows:

```
(textfield "initial text" 30 ...)
(textarea 10 30 ...)
```

The first two arguments of the textfield expression are required and they specify the initial string and the width (in columns), respectively. Likewise, the first two arguments of the textarea expression are required and specify the number of rows and the number of columns, respectively. Textfields can also take an action (which will be invoked when the user hits the enter key).

The following program shows how to write use a textarea to store and display the program's responses to the user. In this case, the user selects choices of languages, and the program appends "Hello" in that language into the textarea?

```
(define ta (textarea 10 50)); 10 rows and 50 cols
(define w
   (window "Hello"
     (col
       (choice "English" "Spanish" "Japanese"
         (action (lambda(e)
           (case (readstring (.getSource e))
             (("English") (.append ta "Hello\n"))
             (("Spanish") (.append ta "Hola\n"))
             (("Japanese") (.append ta "Konichi-wa\n"))
            ))))
       ta)))
(.pack w)
(.show w)
```

Here we have named the textarea `ta` using a `define` expression? and the expression

```
  (.append ta "TEXT")
```

has the effect of adding the "TEXT" to the end of the textarea

Observe that each of the three choices result in something being written to a textarea, we can make a slight change to this program to eliminate this redundnacy by movingb the case inside the writeexpr, as follows:

```
(action (lambda(e)
     (.append ta
```

```
          (case (readstring (.getSource e))
            (("English") "Hello\n")
            (("Spanish") "Hola\n")
            (("Japanese") "Konichi-wa\n")
        ))))
```

This version is preferable because by removing the redunancy, we also remove oppurtunities for making mistakes. If we wanted to change this program so that instead of appending to the textarea it just erased the previous contents wrote the "TEXT" by itself, this could be done by replacing `.append` with `writestring`. In the first version, one would have to make this replacement for each of the clauses of the case expression. In the second version, one only makes one change.

6.13 A Chat Applet

This next example (Figure 6.2) shows an IM-like interface for talking with an electronic psychiatrist. The GUI consists of a central textarea for displaying the conversation (in an IM-like manner), together with a textfield at the bottom for the user's responses

```
(define t (maketagger))
(define w (window "doctor silicon"
  (border
    (center
       (t "ta" (textarea 10 60
 "Hello. Why don't you tell me how you are feeling today?")))
    (south
      (table 1 2
       (label ">> ")
       (t "user" (textfield "" 60 yellow (action (lambda(e)
            ....the action  goes here....
         )))))))))
(.pack w)
(.show w)
```

This psychiatrist randomly picks out responses from a short list every time you enter your own sentence. Observe that the GUI has exactly one action, which is applied when the user hits return after typing in a response.

```
 (action (lambda(e)
  (appendlnexpr (t "ta")
       (string-append
           "\n\n Client: "
           (readstring (t "user"))
           "\n\n Doctor: "
           (respond (readstring (t "user")))))
```

```
(jlib.JLIB.load)
(define (random N)  ;return random number between 0 and N
  (.intValue (Math.round (* N (Math.random)))))

(define (respond sentence)
    (case (random 10)
     ((0)    "Relax. Tell me about your first memories.")
     ((1)    "I'm a computer. You can tell me anything!")
     ((2)    "Are you often depressed?")
     ((3)    "What is your deepest darkest secret?")
     ((4)    "If you want help, youhave to open up!")
     ((5)    "Good. Now we are getting somewhere.")
     ((6)    "Ummmmm. Go on.")
     ((7)    "I see. Can you elaborate.")
     ((8)    "No. I just can't believe that.")
     (else   "Hmmm, interesting, please continue.")))

(define t (maketagger))
(define w (window "doctor silicon"
  (border
    (center
       (t "ta" (textarea 10 60 "Hi. How are you today?")))
    (south
      (table 1 2
       (label ">> ")
       (t "user"
        (textfield "" 60 yellow
         (action (lambda(e)
           (.append (t "ta") "\n Client: ")
           (.append (t "ta") (readstring (t "user")))
           (.append (t "ta") "\n Doctor: ")
           (.append (t "ta") (respond (readstring (t "user"))))
           (.append (t "ta") "\n\n")))))))))
(.pack w)
(.show w)
```

Figure 6.2: An IM window with a robot-Doctor

```
     (writeexpr (t "user") "")))
```

The action appends the user's response to the center textarea (along with a tag identifying it as the user's sentence). It also generates a response to the sentence, and appends that one as well to the centeral textarea (this time with the "doctor" label).

Observe the redundnacy in the action. Each line starts with the same `.append` call. We can eliminate this redundancy by having just one call to `.append` if we first append the five strings together using `string-append`. The new action would become:

```
 (.append (t "ta")
    (string-append
            "\n Client: "
            (readstring (t "user"))
            "\n Doctor: "
           (respond (readstring (t "user")))
            "\n\n"))
```

Yet another approach would be to use the "quasi-string" notation that we saw in our work on servlets. The action would then have the form:

```
 (action (lambda (e)
   (.append (t "ta") {
Client: [(readstring (t "user"))]
Doctor: [(respond (readstring (t "user")))]

  })))
```

Recall that the curly braces {} are used to enclose text and that the expressions in the square braces are evaluated to produce strings which are then inserted into the text.

Exercise 19 *Try extending the psychiatrist applet by adding more keyword lists and more responses.*

6.14 Calculator Programs

So far we have worked with programs that reading and write text from various components. The following demo shows how to write programs that operate with numbers. The key difference is that instead of using `readstring` we use `readexpr`.

The program below doubles whatever the user types into the textfield. The action is taken when the user hits the enter key:

```
  (define w
     (window "Hello"
         (textfield "1" 20
```

```
          (action (lambda(e)
            (let* ((t (.getSource e))
                   (x (readexpr t))
                   (y (* 2 x)))
              (writeexpr t y)))))))

(.pack w)
(.show w)
```

The first two arguments to "textfield" are the initial contents (in this case the number 1) and the size of the textfield (in columns). One can also set the font and background color of the textfield.

6.15 Menus

Menus are constructed using three procedures `menubar` this contains a list of the menus, `menu` this contains a string (the name of the menu) followed by menus and menuitems, `menuitem` these contain strings labelling the menuitem. One can also provide actions to be carried out when then menuitem is selected.

Below is a standard example of a File menu with a quit option that hides the window. This also has GUI components for implementing a chat window (although none of the needed actions are there!)

```
(define w
  (window "Hello"
    (menubar
      (menu "File"
        (menuitem "quit"
          (action (lambda(e) (.hide w)))))
      (menu "Help"
        (menuitem "about")
        (menuitem "documentation")))
    (border
      (north
          (label "Chat Central" (HelveticaBold 40)))
      (center
          (textarea 10 50))
      (south
          (row
            (label ">>")
            (textfield "chat here" 30))))))
(.pack w)
(.show w)
```

Observe that each menu can have zero or more menuitems and can also contain submenus.

We illustrate the process of adding actions to a GUI by showing how to attach actions to buttons and menuitems that make various windows appear and dissappear. For example, we will add the "quit" and "about" menuitems to a GUI. The idea is that when the user selects the "quit" item, the main window should disappear. This is done by adding the following expression into the "Quit" menuitem

```
(action (lambda(e) (.hide main)))
```

where "main" is the name of the main window. When the user selects the "about" item from the "Help" menu, a window should popup that gives some information about the program. This is done with a similar action, except that we use `.show` instead of `.hide`.

We implement this sample program by first creating both the "about" window and the main window, but we only "show" the main window. When the user clicks on the "about" menuitem, the action should "show" the "about" window. We also put an OK button on the "about" window which will make the window disappear when it is pressed.

The code for this program is below:

```
"AboutWin"
"Tim Hickey"
"http://www.cs.brandeis.edu/~tim"
"This demo shows how to show and hide windows in a GUI"
"http://www.cs.brandeis.edu/~tim/jscheme.gif"

(define lib (jlib.JLIB.load))

(define main
 (window "About Window Demo"
   (menubar
      (menu "File"
        (menuitem "Quit"
          (action (lambda(e) (.hide main)))))
      (menu "Help"
        (menuitem "About"
          (action (lambda(e) (.show aboutwin))))))
   (border
     (north (label "Multi-window Demo" (HelveticaBold 40)))
     (center
      (textarea 5 20
         "Try clicking on the about and and quit menuitems")))))
(define aboutwin
  (window "about"
    (border
     (center
      (textarea 10 40 (color 255 245 235)
```

```
"
Title: AboutWin
Author: Tim Hickey
Home: http://www.cs.brandeis.edu/~tim
Description: This demo shows how to show/hide windows in a GUI
Icon: http://www.cs.brandeis.edu/~tim/jscheme.gif
"))
     (south
        (button "ok"
           (action (lambda(e)
              (.hide aboutwin))))))))
(.pack main)
(.pack aboutwin)
(.show main)
```

6.16 MoreCalculator Programs

Here we give a few more examples of programs that get numeric input from components and use that to compute some numerical result. Our example is the Fahrenheit to Celsius converter in Figure 6.3.

Lets look over this application. First, the GUI itself uses a border layout with a menubar. The menubar has a File menu with a quit menuitem and when the user clicks the quit item, the window is hidden.

```
    (menubar
     (menu "File"
      (menuitem "quit"
        (action (lambda(e) (.hide win))))))
```

The main part of the window has a label in the north, a labelled input textfield in the center, and two buttons with actions in the south. The actions are the trickiest part of this program. Lets look at them in detail.

```
 (let* (
           (f (readexpr (t "x")))
           (c (* (/ 5.0 9.0) (- f 32.0)))
         )
   (writeexpr (t "x") c))
```

This action starts by reading the expression in the textfield (t "x"). It then uses that number, f, to compute the celsius equivalent. This is done by first subtracting 32 and then multiplying the result by 5/9. Once f and c have been computed, the value of c is written back onto the textfield with the "x" tag.

Rather than using two buttons, we could have used a "choice" component as follows:

```
(t "operation" (choice "convert to F" "convert to C"
```

```
"F to C converter"
"Tim Hickey"
"http://www.cs.brandeis.edu/~tim"
"This converts Fahrenheit to Centigrade"
"http://www.cs.brandeis.edu/~tim/hickey.new.gif"

(begin
  (jlib.JLIB.load)
  (define t (maketagger))
  (define win (window "F to C Converter"
    (menubar
     (menu "File"
      (menuitem "quit"
        (action (lambda(e) (.hide win))))))
    (border
      (north (label "F to C Converter" (HelveticaBold 32)))
      (center
        (row
         (label "temperature")
         (t "x" (textfield "212" 20))))
      (south
        (grid 1 2
          (button "convert to C" (action (lambda (e)
            (let* (
              (f (readexpr (t "x")))
              (c (* (/ 5.0 9.0) (- f 32.0))))
            (writeexpr (t "x") c)))))
          (button "convert to F" (action (lambda (e)
            (let* (
              (c (readexpr (t "x")))
              (f (+ 32.0 (*  c (/ 9.0 5.0)))))
            (writeexpr (t "x") f)))))))))
  (.pack win)
  (.show win)
```

Figure 6.3: FtoC.applet

```
(action (lambda(e)
 (case (readstring (t "operation"))
   (("convert to F")
       (let* ( (f (readexpr (t "x")))
                 (c (* (/ 5.0 9.0) (- f 32.0))) )
         (writeexpr (t "x") c)))
   (("convert to C")
        (let* ( (c (readexpr (t "x")))
                  (f (+ 32.0 (*  c (/ 9.0 5.0)))))
          (writeexpr (t "x") f)))))))
```

The key points to notice here are that we use "readstring" because the choice labels have spaces in them and in the cases we use double-quotes around the strings, for the same reason.

Exercise 20 *Modify the calculator program so that it contains two textfields (labelled F and C). Add actions to the textfields such that typing a number in the F field and hitting return will write the equivalent Celsius temperature in the C field, and vice versa.*

Exercise 21 *Write a program that computes the area of a triangle give then length of its three sides. The formula, which holds for all triangles, is*

$$area = \sqrt{(s-a)(s-b)(s-c)s)}$$

where the three sides have lengths a, b, and c and s is the semi-perimeter, defined by

$$s = (a+b+c)/2$$

To do this in Scheme you will need to use the square root function `sqrt`*. For example,* `sqrt 16` *returns 4.0.*

6.17 Quizzes

The program in Figure 6.4 demonstrates the use of choice components and the use of a "case" expression in the action for the choice. There are a few points to note here:

- We can read what is written in the "choice" component by using
  `(readstring (.getSource e))`
  in the action. The `e` refers to the **event** that occurs when the choice is selected, the call to `.getSource` returns the components that generated that event (i.e. the choice itself), and `readstring` returns the string written on the choice component.

- The `cond` expression allows one to generated different responses depending on what the choice was selected.

```
(define t (maketagger))
(define w
 (window "quiz game"
   (border
      (north (label "Quiz Game" (HelveticaBold 32)))
      (center
       (col
         (row
           (label "What is the capital of Kenya")
           (choice "Cairo" "Nairobi" "Sudan" "El Paso"
             (action (lambda(e)
                (let ((G (readstring (.getSource e))))
                  (writestring (t "response")
                    (cond
                      ((equal? G "Cairo") "No, that's in Egypt")
                      ((equal? G "Nairobi") "Yes, good job!")
                      ((equal? G "Sudan") "That's a country!!")
                      ((equal? G "El Paso") "Wrong hemisphere!")
           )))))))

         (row
           (label "What is the capital of Bolivia")
           (choice "Cairo" "Nairobi" "Sudan" "El Paso"
             (action (lambda(e)
               (let ((G (readstring (.getSource e))))
                 (writestring (t "response")
                   (cond
                     ((equal? G "Cairo") "No, that's in Egypt")
                     ((equal? G "La Paz") "Yes, good guess?!")
                     ((equal? G "Sudan") "That's a country!!")
                     ((equal? G "El Paso") "Wrong hemisphere!")
           )))))))))
       (south
          (t "response"
            (label "----------------------"))))))
(.pack w)
(.show w)
```

Figure 6.4: Quiz Game

- The responses are written into a label at the bottom of the quiz game.

Note that this example illustrates quite a bit of redunancy again. We can eliminate some of this redundancy by trying to abstract out that which is the same across each of the redundant versions and capture it in a procedure as shown in Figure 6.5 This version has many advantages over the earlier one. In particular, if one wants to change the layout of the quiz questions, the change need only be made once (in the definition of quizzer). Also, the specification of each quiz question contains only the minimal amount of information needed (the questions and the ir responses).

6.18 Graphics

The simplest way to create graphics using JLIB is to use the canvas procedure to create a panel for drawing on.

```
(define c (jlib.SchemeCanvas. 400 500))
```

The arguments to the canvas method specify the width and height in pixels.

There are several methods available for drawing on a canvas. They all require that you use the "graphics context" of the canvas which you get using

```
(.bufferg$ c)
```

The graphical operations you can use are:

```
(define g (.bufferg$ c))
(.setColor g (color R G B))
(.drawLine g x1 y1 x2 y2)
(.drawRect g x1 y1 w h)
(.drawOval g x1 y1 w h)
(.drawArc g x1 y1 w h a d)
(.fillRect g x1 y1 w h)
(.fillOval g x1 y1 w h)
(.fillArc g x1 y1 w h a d)
```

You can also draw a string S starting at location x,y using:

```
(.setFont g (Helvetica 12))
(.drawString g x y str)
\end{verbatgim}
General Polygons can be drawn by first creating a polygon from points, and then drawing
\begin{verbatim}
(define p (java.awt.Polygon.))
(.addPoint p x1 y1)
(.addPoint p x2 y2)
      ...
(.drawPolygon g p)
(.fillPolygon g p)
```

```
(define (quizzer Question A1 R1 A2 R2 A3 R3 A4 R4)
  (row
    (label Question)
    (choice A1 A2 A3 A4
      (action (lambda(e)
        (writestring (t "response")
          (let ((G (readstring (.getSource e))))
            (cond
              ((equal? G A1) R1)
              ((equal? G A2) R2)
              ((equal? G A3) R3)
              ((equal? G A4) R4)))))))))
(define t (maketagger))
(define w
 (window "quiz game"
   (border
      (north (label "Quiz Game" (HelveticaBold 32)))
      (center
       (col
         (quizzer "What is the capital of Kenya"
             "Cairo" "No, that's in Egypt"
             "Nairobi" "Yes, good job!"
             "Sudan" "That's a country!!"
             "El Paso" "Wrong hemisphere!")

         (quizzer "What is the capital of Bolivia"
               "Cairo" "No, that's in Egypt"
               "La Paz" "Yes, good guess?!"
               "Sudan" "That's a country!!"
               "El Paso" "Wrong hemisphere!")))
       (south
        (t "response"
           (label "----------------------"))))))
(.pack w)
(.show w)
```

Figure 6.5: Quiz Game

```
"gdemo0"
"Tim Hickey"
"http://www.cs.brandeis.edu/~tim"
"Simple demo showing use of graphics in JLIB"
""

(jlib.JLIB.load)
(define c (canvas 400 400)) ;; a 400x400 area for drawing
(define w (window "graphics1"
 (border
   (center c)
   (south
    (row
      (button "clear" (action (lambda(e) (run-it clear))))
      (button "draw" (action (lambda(e) (run-it drawface))))
     )))))

(define (run-it F) ;run animation and report errors
  (define ta (textarea 10 60))
  (define errwin (window "error" ta))
  (.pack errwin)
  (.start (Thread. (lambda()
      (tryCatch  (F)  ; run graphics code F
         (lambda(e)   ; display any errors that arise
           (writestring ta e)  (.show errwin)))))))


(define (clear)
  (define g (.bufferg$ c))
  (.setColor g white)
  (.fillRect g 0 0 1000 1000)
  (.setColor g black)
  (.repaint c))

(define (drawface)
  (define g (.bufferg$ c))  ;; get graphics object of canvas
  (.setColor g blue)
  (.fillOval g 200 200 100 100)  ;; draw a blue "face"
  (.setColor g red)
  (.fillOval g 220 220 10 10)    ;; draw left "eye"
  (.fillOval g 270 220  10 10)   ;; draw right "eye"
  (.fillOval g 220 260 60 30)    ;; draw mouth
  (.repaint c))

(.resize w 400 400)
(.show w)
```

Figure 6.6: A simple graphics program

Figure 6.6 gives an example of a simple graphics program. The GUI code creates a "canvas" for drawing on, and two buttons – one for clearing the screen and one for drawing a face. Both procedures are called using the "run-it" procedure.

```
(define (run-it F) ;run animation and report errors
  (define ta (textarea 10 60))
  (define errwin (window "error" ta))
  (.pack errwin)
  (.start (Thread. (lambda()
      (tryCatch  (F)  ; run graphics code F
         (lambda(e)   ; display any errors that arise
           (writestring ta e)  (.show errwin)))))))
```

The "run-it" procedure starts the graphics running in a separate "thread," this allows you to still interact with the program while the graphics are going on. Its not too important in this case, but if we had a long animation (e.g. with a face moving around the screen), you would want to be able to resize the window, change its position, close it, etc, and for this you need the graphics to be running "separately" from the rest of the program. The "run-it" procedure also contains a "tryCatch" expression. This is used for catching in errors and reporting them (in the lambda section).

The "draw-face" procedure shows a straightforward use of the graphics commands. The points to observe are that one must first get the graphics buffer "g" of the canvas before drawing on it, and finally that one must call "repaint" at the end to make the changes visible. The "repaint" command causes the current contents of the buffer to be drawn onto the canvas all at once.

Exercise 22 *You might try modifying this example to have the face move across the page when a new "move face" button is pushed. The idea is just to repeatedly clear the buffer and call "draw-face".*

6.19 Security

The software we discuss in this chapter can be run in a browser or independently of a browser.

There are two approaches to running these programs in browsers. We assume they are being served by the Jscheme webapp of the Tomcat server described in the previous chapters.

If the program has the extension .applet it will be run as a java applet inside the browser. It it has the extension .snlp, it will be run as a Java Web Start application. In both of these cases, the program must begin with five lines of comments:

```
program name
program author
URL of program website
```

```
one line description of program
URL of image to be used as icon for program
```

The programs can also be run directly from a command window using the Jscheme archive file (jscheme.jar) which is available inside the webapps/scheme/lib folder in the jscheme tomcat webapp. The command for running the chat.applet is:

```
% java -cp jscheme.jar jscheme.REPL chat.applet
```

assuming that the jscheme.jar and chat.applet are in the same location.

The applet and snlp versions of the program will run in a "sandbox", which is a protected run space that prevents the programs from viewing or modifying any files on your computer. It also prevents the program from making certain internet connections and puts other more subtle limits on the program.

When the program is run directly from the command window there is no sandbox used by default.

The current JLIB primitives are listed below:

6.20 Summary of GUI building procedures

Components are created using the functions below. The optional arguments are described in the next section and can be used to set the color, font, and other properties of the component. The type of the argument is used to determine how it affect the component.

```
* (window TITLE arg1 arg2 ...)
* (button STRING arg1 arg2 ...)
* (textfield STRING NUMCOLS arg1 arg2 ...)
* (textarea NUMROWS NUMCOLS arg1 arg2 ...)
* (choice arg1 arg2 ...)
* (label STRING arg1 arg2 ...)
* (canvas W H ...)
* (menubar NAME arg1 arg2 ...)
* (menu NAME arg1 arg2 ...)
* (menuitem NAME arg1 arg2 ...)
```

When you have more than one component in a window you need to decide where each component should appear in the window and what should happen when the window size is changed. These placement decisions are specified using layout procedures which we will introduce as we go along.

Reading/Writing

- `(readstring COMPONENT)` – reads the text on a component as a string
- `(readexpr COMPONENT)` – reads the text on a component as a Scheme term

- `(writeexpr COMPONENT OBJ)` – convert the object to a string and write on the component
- `(.append COMPONENT OBJ)` – append the string to the text on the component

Other Common Actions

- `(.hide COMP)` `(.show COMP)` hiding or showing a component (or window)
- `(.pack WIN)` `(.resize WIN ROWS COLS)` resizing a window
- `(.setForeground COMP COLOR)` `(.setBackground COMP COLOR)`
- `(.setFont COMP FONT)`

Part IV

Recursion and Algorithms

Chapter 7

Overview of Scheme

In this chapter we provide an overview of those parts of the Scheme language that we have seen so far, in addition to a few aspects that have not yet been covered. In the following Chapter we investigate the substitution model of Scheme and discuss how to write more complex programs. This leads naturally to a discussion of the limits of computation and we provide some examples of explicit problems that can not be solved by computers (but may be solvable by humans!)

7.1 Scheme expressions and values

Every Scheme program is a sequence of Scheme expressions where a Scheme expressions is either a constant, symbol, quoted element, or a compound expression called an s-expressions.

The constants are numbers, double quoted strings (e.g. "this is a string"), single characters (e.g. `#'a', #' '`), and boolean values (true is `#t` and false is `#f`).

Symbols are typically sequences of printable characters (excluding spaces, quotes, double quotes, parentheses, square brackets, or curly brackets). For example, `name`, `feet->meters`, `cool!!!` are all symbols.

A quoted element is formed by putting a single quote (') in front of the object. For example, `'a ''b '1.234` are three quoted elements, the second being a quote of a quoted element.

The constants and symbols of a Scheme program are combined together using s-expressions, which are a kind of generalized list. A list is represented in Scheme by enclosing the elements of the list in a pair of parentheses:

```
( element1 element2 ... elementk)
```

For example, `()` is the empty list, `(1 2 3 4 5)` is a list of numbers and `(a b "cool list" 2.718281828459045 -3 #t)` is a list of various constants.

An S-expression is a list whose elements are either constants, symbols, quoted elements, or s-expressions. For example,

```
(+ 1 2 (* 3 4 5) (/ (* 9 10) 15))
```

is an s-expression.

7.2 Evaluation

A Scheme program is a set of S-expressions and one "runs" the program by evaluating these expressions in order, from the first to the last. Evaluation of an expression is a process in which the expression is transformed, possibly in several steps, into a value.

7.2.1 Arithmetic

The simplest expressions to evaluate are the mathematical expressions. These are combinations of numbers and mathematical operators. The rules for evaluating such expressions are the following:

E1 One must evaluate the innermost expressions first

E2 In an expression `(F A1 ... An)`, the expression `F` must evaluate to an operator f (e.g. addition), and the expressions `Ai` must evaluate to values a_i. One then applies the operator f to the values a_1, ..., a_n, to compete the evaluation.

Thus, to evaluate an expression one first evaluates the innermost parenthesized subexpressions and replaces them with their values[1]. This process is repeated until no parentheses remain. For example, if we start with

```
(+ 1 2 (* 3 4 5) (/ (* 9 10) 15))
```

The innermost s-expressions are `(* 3 4 5)` and `(* 9 10)`. These evaluate to 60 and 90, respectively, so the original expression evolves to

```
(+ 1 2 60 (/ 90 15))
```

Now the innermost expression is `(/ 90 15)` which evaluates to 6. Hence the expression evolves to:

```
(+ 1 2 60 6)
```

This has no subexpressions and so the final evaluation yields

```
69
```

which is the result of evaluating the original s-expression.

The list of mathematical operators available in Scheme is shown in Figure 7.1.

[1] Technically, we should distinguish between the numerals 1,2,3,... and their values, but we will gloss over this distinction for the time being

```
(+ num num num ... num)  ;; addition
(* num num num ... num)  ;; multiplication
(- num num)              ;; subtraction
(- num)                  ;; negation
(/ num num)              ;; division
(sqrt num)
(sin num)
(cos num)
(tan num)
(asin num)
(acos num)
(atan num)
(exp num)
(log num)
```

Figure 7.1: Mathematical Operators in Scheme

7.2.2 GUI expressions and side-effects

The rules for the evaluation of simple GUI expressions are almost the same as for arithmetic. As before one evaluates the innermost expressions first, but the values are now graphical widgets. For example, to evaluate the expression

```
(.show
 (window "test"
  (menubar
    (menu "File" (menuitem "quit"))
    (menu "Help" (menuitem "about")))))
```

we first evaluate the innermost `menuitems` to get two menuitems which we denote by `MQ` and `MA` respectively. Evaluating these expressions causes the windowing system of the computer you are using to create two menuitem widgets (and in the process assigning many default values, e.g. the font used, the background color, the border, the size, etc.) The creation of these menuitems is called a **side-effect** of the evaluation of the expression. If you evaluate a single menuitem, the Scheme system will return some textual representation for that object. For example, evaluating

```
(menuitem "Quit")
```

might result in the following

```
java.awt.MenuItem[menuitem0,label=Quit]
```

Returning to our example, the expression evolves to

```
(.show
 (window "test"
  (menubar (menu "File" MQ) (menu "Help" MA))))
```

where `MQ, MA` are our representation of the menuitem objects that were created in the previous step.

Now the `menu` expressions are innermost and the evaluate to two menu elements, say `MF` and `MH`, yielding

```
(.show (window "test"  (menubar MF MH)))
```

The `menubar` expression evaluates to a menubar object `MB`:

```
(.show (window "test"  MB))
```

The `window` expression evaluates to a window object `W`.

```
(.show W)
```

and the `show` expression evaluates to the special value `#null` and at the same time makes the window appear on the screen.

7.3 Special Forms

There are about a dozen special forms in Scheme. These are expressions whose evaluation rules differ slightly from the standard evaluation described above. Most expressions `(F A ... B)` are evaluated by evaluating all of their components `F,A,...,B` to get Scheme objects `f,a,...,b` and then applying `f` (which must be a function) to the arguments `a,...,b`. Special forms usually postpone the evaluation of some of their components until later. Once you know the special forms and a reasonable subset of the primitives, you will find yourself able to write substantial programs fairly easily.

7.4 `define` and symbol values

We have already seen that some Scheme symbols come with preassigned values, e.g. the arithemtic operators (+,-,*,/) and the mathematical functions (sin,cos,...). Others can be given values by loading libraries. For example, loading the JLIB library gives values to the `window`, `menubar`, and other procedures.

Users can directly assign values to symbols using the special form `define`.

```
(define NAME EXPRESSION)
```

For example, to compute the volume of a cylinder whose base is a circle of radius 10 and which has height 34 we can evaluate the following expressions.

```
(define pi 3.14159265358979323846264338327 6)
(define area10 (* pi 10 10))
(define cylvol (* area10 34))
```

This binds `area10` to the area of the base, and `cylvol` to the volume of the cylinder.

7.5 lambda and Anonymous functions

Scheme also lets one define functions and assign them to symbols. For example, evaluating the following expression binds the symbol `square` to the function that squares its input value.

```
(define (square x) (* x x))
```

To handle these user definitions we must add a new evaluation rule for Scheme

E4 To evaluate an expression `(F A1 ... An)` where `F` is a user defined procedure of the form

```
(define (F p1 ... pn) BODY)
```

one replaces the expression by the `BODY` of the procedure, but with the parameter symbols `p1`, ...,`pn` replaced by the corresponding arguments `A1`, ...,`An`.

Thus, to evaluate

```
(sqrt (+ (square 4) (square 3)))
```

We first replace the `(square 4)` and `(square 3)` expressions with their respective bodies

```
(sqrt (+ (* 4 4) (* 3 3)))
```

Evaluation then proceeds as before, yielding

```
(sqrt (+ 16 9))
```

which becomes

```
(sqrt 25)
```

and finally yields

```
5
```

as expected as $5 = \sqrt{3^2 + 4^2}$.

There is another equivalent way to define functions using the the "lambda" special form. This allows us to separate the process of defining a function from the process of binding it to a symbol. For example, the square function can be defined as follows:

```
(define square (lambda (x) (* x x)))
```

Here the special form `(lambda(x) (* x x))` returns the function that replaces its single argument `x` with its body `(* x x)`. The `define` expression, then binds that function value to the symbol `square`. You can think of "lambda" as being a synonym for "function." Its general form is

```
(lambda (v1 ... vn)
   E1
   E2
   ...
   Ek)
```

This represents a function of n variables `(v1 ... vn)`. When this function is called it evaluates each of the expressions E1,..., Ek and returns the value of the last expression Ek. The sequence of expressions `E1`, ..., `Ek` is called the `body` of the procedure.

7.6 Internal defintions

The first few expressions inside a procedure body can themselves be `define` expressions. These are called local definitions and are visible only inside the procedure itself.

For example, we could write a procedure to create a show a warning window as follows:

```
(define (warning message)
   (define w
    (window "Warning"
     (label message red (HelveticaBold 40))))
   (.pack w)
   (.show w)
    w)
```

Each time this procedure is called it creates a window containing the warning message and makes it visible. For example, evaluating the following expressions will create two warning windows.

```
(warning "Vulgar conversation is prohibited.")
(warning "Press Quit to end this session.")
```

Note that the value of the symbol `w` is not visible outside of the definition, which is good since it allows us to call the procedure several times and have several windows without creating a conflict over the variable `w`.

7.7 `set!` and changing defined values

You can cahnge the value of a defined variable using the "set!" special form (it is pronounced "set-bang")

```
(set! NAME EXPRESSION)
```

For example, in the following example, each time we call the square function it adds on to the variable "a. " This provides a way of counting how many times "square" is called.

```
(define a 0)
(define (square x) (set! a (+ a 1)) (* x x))
```

7.8 if and Conditional Execution

Scheme offers three special forms for conditional execution (that is for providing a means for testing the input and executing different code depending on the result of the test). The first is the if-then-else expression. It evalutes the TEST if the test is false (`#f`) it evaluates the ELSE code; otherwise, if the test is true (`#t`) it evaluates the THEN code. (In fact, if the test is any non-false value, it evaluates the THEN code):

```
(if TEST THEN ELSE)
```

For example, we can write the function that returns the maximum of its two inputs as follows:

```
(define (max2 a b)
  (if (< a b) b a))
```

7.9 cond and multiple conditions

Sometimes one has many different tests and for each test there is a different piece of code that should be evaluated. This can be implemented with nested "if"s, but a better approach is to use the "cond" special form:

```
(cond (
   (TEST1 E1 ... E2)
   (TEST2 F1 ... F2)
   ...
   (else G1 ... G2)
     )
)
```

The expression evaluates the first "TEST1" if it is true (actually just non-false), then it evaluates E1,... E2 and returns the value of E2. If "TEST1" is false, then it proceeds to TEST2, and repeats the process. If none of the tests are true, then it evaluates the expressions in the "else" clause (G1, ..., G2) and returns the value of the last one.

7.10 case and constant values

The case is a good expression to use when the tests are all of the form `(equal? X Vi)` for some constant values V1, V2, ... The general form of this expression is

```
(case EXPR
  ((C1 C2 ...) E1 ... E2)
  ((D1 D2 ...) F1 ... F2)
  ...
  (else G1 ... G2)
)
```

For example, the following procedure can be used to determine whether a symbol is a vowel, consonant, digit, or something else:

```
(define (classify-symbol L)
 (case L
  ((a e i o u y) 'vowel)
  ((b c d f g h j k l m n p q r s t v w x z) 'consonant)
  ((0 1 2 3 4 5 6 7 8 9) 'digit)
  (else 'non-alpha-numeric)))

(classify-symbol 'e)  ;; this returns 'vowel
```

7.11 Grouping for side effects

The "begin" expression is useful when you want to evaluate several expression and return the value of the last one. It has the form:

```
 (begin
    E1
    ...
    E2
 )
```

This if often useful with an "if" statement, e.g. the following expression writes onto two components if "age" is less than 21 and otherwise just writes into one component. The `begin` is needed to group the first two write actions into a single expression.

```
  (if (< age 21)
      (begin
         (writestring (t "response") "You are too young")
         (writestring (t "price") "------"))
      (writestring (t "price") "29.95"))
```

7.12 `let` and block structure

Scheme provides several mechanisms for giving names to values for use in a sequence of expressions. The most common of these is the "let*" whose syntax is as follows:

```
(let* (
  (V1 E1)
  (V2 E2)
  ...
  (Vk Ek)
    )
 F1
 F2
 ...
 Fn)
```

This expression is evaluated by first evaluating "E1" and storing the value in "V1," then evaluating "E2" and storing the value in "V2", etc. Finally, these bindings are used to evaluate "F1", "F2",... "Fn" and the value of "Fn" is returned.

7.13 Exception Handling

Sometime procedures are unable to complete their task as expected. In such cases they often "throw an exception." A standard example of this is dividing 1 by 0. The "tryCatch" special form provides a way for handling such exceptions. Its syntax is:

```
(tryCatch E1 (lambda(e) F1 ... Fn))
```

It evaluates the expression "E1" and if there is an exception (say e), then it evaluates the expressions "F1",... and returns the value of "Fn"

7.14 Threads

Sometimes we want part of a program to run "by itself" (e.g. a graphics animation can run "in the background" allowing the user to continue to interact with the GUI). This is done by creating a thread and starting it. The standard idiom for starting up a new thread is as follows:

```
(.start (java.lang.Thread. (lambda() CODE)))
```

Chapter 8

Recursion

In this chapter we investigate an interesting feature of the Scheme evaluation rules. Namely, that the body of Scheme procedure can contain a call to itself.

8.1 Teaching Scheme to Count

Consider, for example, the following procedure:

```
(define (f x)
  (if (= x 0) x (f (- x 1))))
```

Lets now trace the evaluation of the following expression:

```
  (f 2)
```

The first step is to replace the procedure call `(f 2)` with the body of the procedure, where the parameter `x` is replaced by the argument `2`. This yields,

```
  (if (= 2 0) 2 (f (- 2 1)))
```

The test `(= 2 0)` evaluates to false, and so this expression evolves to the `ELSE` part:

```
  (f (- 2 1))
```

which evolves as usual to

```
  (f 1)
```

Again, we have an instance of a user-defined procedure call. This evolves to

```
  (if (= 1 0) 1 (f (- 1 1)))
```

and since `(= 1 0)` evaluates to false, this evolves to the `ELSE` part again:

```
  (f (- 1 1))
```

which evolves to

```
(f 0)
```

Applying the substitution rule yet again we get

```
(if (= 0 0) 0 (f (- 0 1)))
```

but this time the test `(= 0 0)` evaluates to true, so the expression evaluates to the `THEN` part:

```
0
```

It should not be hard to convince yourself that a call to

```
(f 1000)
```

would evolve to `(f 999)` and then to `(f 998)`, and would continue counting down until it reaches `(f 0)` at which point it evolves to `0` and evolves no more.

It should take only a moment for your computer to countdown from one million using this procedure. Counting down from a trillion (or a quadrillion) on the other hand can take days or years (depending on how fast your computer is).

This simple countdown example also illustrates one of the pitfalls of using recursive programs. Consider what happens if we evaluate the expression

```
(f -1.0)
```

This evaluates to `(f -2.0)` and continues counting down getting numbers that are progressively more negative and hence farther from zero. Since the test `(= x 0)` will never be true, this evaluation will never stop. Such never-ending evaluations are called infinite loops.

Actually, this example is somewhat subtle because our initial analysis would require the computer to manuipulate numbers with ever increasing numbers of digits. This would require ever increasing amounts of memory and at some point all matter in the universe would be required just to represent the numbers computed in this loop. Hence in a physical sense the program would generate an out-of-memory error at some point in the future. Another subtle aspect of this problem is that in most implementations of Scheme a decimal point in a number indicates an approximate value and once such numbers get large the system will only be approximately correct, in particular at some point it will reach a number x such that subtracting 1 from x using approimate arithmetic yields the same x back. This then generates an infinite loop without a memory error. Nevertheless, if physical theories are correct, all of the protons in the computer (and the universe) will eventually decay into other other particles, once again causing an computer error that will stop the evaluation. Thus, arguments about infinite loops should always be taken in an abstract, non-physical sense.

8.2 Useful computation

Next we consider a slightly more complex procedure.

```
(define (g x y)
  (if (= y 0)
      x
     (g y (% x y))))
```

Here the percent symbol `(% x y)` denotes the remainder function which returns the remainder of x divided by y, e.g. `(% 17 5)` is 2, as 17 divided by 5 is 3 with a remainder of 2.

Lets now trace the evolution of

```
(g 252 112)
```

This evolves, by the substitution rule, to

```
(if (= 112 0) 252 (g 112 (\% 252 112)))
```

Since the test `(= 112 0)` is false, this evolves to the `ELSE` part of the expression:

```
(g 112 (\% 252 112))
```

and since 252 divided by 112 is 2 with a remainder of 28, this evolves to

```
(g 112 28)
```

Again, by the substitution rule, we get

```
(if (= 28 0) 112 (g 28 (\% 112 28)))
```

which evolves to

```
(g 28 (\% 112 28))
```

but 28 goes into 112 exactly four times with no remainder, so this evolves to

```
(g 28 0)
```

and one last application of the substitution rule yields:

```
(if (= 0 0) 28 (g 0 (\% 0 28)))
```

which evolves to

```
28
```

as the test `(= 0 0)` is true. This evaluation has yielded the greatest common divisor (28) of the two arguments (252,112) of the call.

8.3 The Halting Problem

We have seen that recursion can result in programs that run a very long time, or that never stop at all. It can also be used to right programs that perform fairly complex computations that are difficult for humans to undertake. After starting to write a few recursive programs you will find that infinite loops are an all too common occurrence. Even worse, it is difficult to tell whether your program is in an infinite loop or if it is just taking a long time to compute its answer and will soon return the result.

It would be nice, therefore, to have a procedure `halts?` which you could apply to a function and a procedure to determine whether it would eventually halt. Thus

```
(halts? F 5)
```

would return true if the call `(F 5)` would eventually halt and would return false otherwise. Here we consider only functions that don't interact with the user or the system to simplify matters.

Unfortunately, it is impossible to build such a Scheme system. To see why, imagine someone has given you access to a Scheme system that purports to have a correct implementation of this `halts?` procedure and that this procedure is itself guaranteed to eventually halt with either a true or false value.

Before continuing with this example, lets pause for a moment and consider a related problem of trying to debunk a fortune teller. Suppose someone claims to be able to tell your future in exquisite detail and to always give an answer to any question you may ask. If you ask this charleton which of your two hands you will raise next, then, after he returns his answer, you simply raise the other hand and you have disproven his claim. This situation is relevant to the `halts?` procedure because the designers have claimed to write a procedure that can tell a particular aspect of the future about any procedure we write.

With such a challenge, we could write the following procedure:

```
(define (skeptic x)
  (if (halts? skeptic x) (skeptic x) 0))
```

The idea behind this procedure is that when it is called with an argument `x` it will ask the `halts` procedure whether `skeptic x` will halt or not, and then it will do the opposite!

Lets trace the evolution of a call to skeptic. For example,

```
(skeptic 7)
```

evolves, by the substitution rule to

```
(if (halts? skeptic 7) (skeptic 7) 0)
```

By the guarantee on the `halts?` procedure we know that the test will eventually stop with either a true or false value. But, no matter which value the `halts?` procedure returns, it will be the wrong value! For example, if

```
(halts? skeptic 7)
```

returns true and the `halts?` procedure is working as advertized, then `(skeptic 7)` must eventually halt, but since the test returns true, the expression evolves to

```
(skeptic 7)
```

and has clearly entered an infinite loop. Likewise if

```
(halts? skeptic 7)
```

returns false, and if `halts?` is working as advertized, then `(skeptic 7)` must loop forever, but since the test returns false, the expression evolves to the `ELSE` part which is

```
0
```

and it clearly has halted.

"Ahhh", you say, but maybe we could write a `halts?` procedure that works on all programs that don't contain the `halts?` procedure, i.e., maybe it is the self-reference that is causing the problem. That would be a good guess, but in fact, a slightly more complex argument also shows that no such `halts?` procedure can be written. This demonstrates an interesting situation in which there is a problem (determine whether the evolution of a given Scheme expression will eventually stop or not) which can not be solved by computers. More precisely, there is no computer program that can examine a Scheme expression and always determine whether or not it will eventually halt. This is an inherent limit on the abilities of computers. It does not however imply that humans are similarly limited, unless of course we are merely computers ourselves.

Part V

Advanced Topics

Chapter 9

Databases

In this chapter we give an introduction to the use of databases in web programming. Databases provide a means of storing and retrieving large amounts of data efficiently. In the previous chapter, we have seen a simple way to read data from and write data to the server's disk. For small amounts of data this works well, but it becomes unacceptably slow for large data sets.

We will first give an introduction to databases in general and to the SQL (Structure Query Language) in particular, then we explain how to process database queries in Scheme. In the next Chapter we give several examples of accessing databases from Scheme servlets.

9.1 Database concepts

A **database** is a named collection of data which is organized into **tables**. When you create a table, you must provide a unique name for the table, and you must specify the number of `columns` in the table and the names for each columns. Moreover, you must specify what type of value each column can contain. The most common types are

- `INTEGER` for whole numbers
- `DECIMAL` for decimal numbers
- `VARCHAR` for character strings

9.2 Intro to SQL, SQL clients, and SQL servers

SQL (Structured Query Language) is the most widely used language for managing databases. It is a relatively simple language that allows you to

- create new databases and tables
- put data into a table

- add users to a database and give them permissions
- select information from a database
- modify information in a database

An SQL system usually consists of two components:

- an SQL engine that creates, modifies, and searches through the databases, and
- an SQL client that connects to the engine and provides a graphical interface for the user to construct and execute SQL queries.

The SQL server can often be setup as a web service, so that anyone can connect to the server over the web and make SQL queries, provided they know a user name and password for accessing the system.

In this Chapter we will describe the use of one particular open source database server and client, the hsql Database Engine – a sourceforge open-source project. We have chosen this database because it is multiplatform, easy to install, and free. Like almost all software, it comes with no warranty, and in this case, it should not be used for commercial purposes unless you invest great care in analyzing the code looking for security problems. Nevertheless, it will serve fine as an educational tool. Another database

The hsqldb system and its documentation can be downloaded from

```
http://hsqldb.sourceforge.net
```

We have included the hsqldb system itself in the scheme webapp (actually it is in the file `tomcat/webapps/scheme/WEB-INF/lib/hsqldb.jar`.

9.3 A Simple Database-backed Survey

In the next section we give a complete example of setting up a database and a table and building a servlet that uses that table to implement a simple survey and analyze the results. In the later sections we will discuss the concepts underlying this example. In the following Chapter we give more examples of database-backed servlets.

9.3.1 Overview of the survey servlet

The survey is just an html file that asks the users age, political party, and and who they voted for in the last presidential election. When the user answers these questions and submits the form, the data will be stored as a row in a database and then several queries will be run to analyze the data gather so far. This data will then be presented to the user on the response page.

Thus we must create a database with a table to store the survey answers, an HTML page to get the users answers and a servlet page to store the data in

the database and generate a response page with a summary of the poll results so far.

It would be better to try and make sure the same person from the same computer doesn't vote many times to influence the outcome, but in this simple example we will ignore these concerns.

9.3.2 Creating a new database

The first step is to start a SQL client and create a database and table for your data. You start the client with the following commands from inside the hsqldb folder:

```
% cd demo
% java -cp ../lib/hsqldb.jar org.hsqldb.utils.DatabaseManager
```

Note: Windows users must use backslash instead of slash. Also, you must set your path so that it contains the java program.

Starting the DatabaseManager pops up a window titled "Connect" with five fields. You should select the following field values:

```
PROMPT    VALUE                              DEFAULT VALUE
Type:     HSQL Database Engine Standalone (the second choice)
Driver:   org.hsqldb.jdbcDriver           (this one)
URL:      jdbc:hsqldb:webDB               (jdbc:hsqldb:test)
USER:     sa                              (sa)
Password:                                 ("")
```

The USER "sa" is the System Administrator, which is the only user able to create new databases and to have full control over the database. The "webdb" in the URL, is the name of your new database.

Once you click "OK" you will have created the `webdb` database (if it didn't already exist it will be created) and you will see the "HSQL Database Manager" window which provides a front-end to the database.

9.3.3 Creating a new table in the database

You can now create the "survey" table by selecting `CREATE TABLE` from the `Command` menu, and then completing the command to read as follows:

```
CREATE TABLE survey(age INTEGER,party VARCHAR,votedfor VARCHAR);
```

Hitting the `Execute` button will create that table and you can examine it by selecting the `Refresh Tree` menuitem in the `View` menu.

Next change the system administrator password to what ever you want by executing the following query.

```
SET PASSWORD mynewpassword;
```

This will prevent others from being able to connect to your server as the system adminstrator. The next time you visit this database, you will need to specify "mynewpassword" as the "sa" password, rather than the default empty password.

9.3.4 Starting a server for the database

Next you should go to the tomcat/webapps/scheme/WEB-INF/db folder and start up the hsqldb database engine server as follows:

```
% cd webapps/scheme/WEB-INF/db
% java -cp ../lib/hsqldb.jar org.hsqldb.Server -port 9001 -database webdb
```

This starts a service on your computer allowing access to the webdb database on port 9001. If you want to provide access to several different databases, you need to start a new server for each one, and use different ports for each database.

9.3.5 The survey servlet

We have now finished creating the database and table. Next, we discuss the servlet `dbsurvey.servlet` which creates an HTML form and processes the user's responses to that form. The code for the form is shown in Figure 9.1.

The general structure of the servlet is that it first gets the three form parameters (age, party, and votedfor) from the browser. If `age` is `#null` this means that the user has not yet visited the form page, so the servlet generates the HTML form. If `age` is not `#null`, then we must store the user vote in the database, get the current tally, and generate a webpage with the current tally on it.

```
[; this is in the file "dbsurvey.servlet"
 (let (
  (age (.getParameter request "age"))
  (party (.getParameter request "party"))
  (votedfor (.getParameter request "votedfor"))
    )
(if (equal? #null age)  ;; first visit to this page
  THEN ....make web page with form to get user data....
  ELSE ....store user data in database, report current tally....
))
]
```

We have already seen how to create an HTML form, and there is nothing new here, so lets look at the "ELSE" part of the "if" expression.

Before we can access the database, we need to load the "db.scm" library. Also, we will be converting a database answer into an HTML table, so we need the "table.scm" library. The code for storing the user's vote into the database uses the `runquery` procedure defined in `lib/db.scm` and has the following form

```
[; this is in the file "dbsurvey.servlet"
 (let (
  (age (.getParameter request "age"))
  (party (.getParameter request "party"))
  (votedfor (.getParameter request "votedfor"))
    )
(if (equal? #null age)  ;; first visit to this page
  {<html><head><title>Survey Form </title></head> <body>
     Please enter the following data:
     <form method="post" action="dbsurvey.servlet">
      <ul>
       <li> Your age: <input type=text name="age"></li>
       <li> Your political party:
         <select name="party"> <option>democrat</option>
         <option>republican</option><option>green</option>
         <option>other</option>  </select></li>
       <li> Who you voted for in last election:
         <select name="votedfor">
           <option>Bush</option><option>Gore</option>
           <option>Nader</option><option>other</option>
         </select></li>       </ul>
       <input type=submit></form> </body></html>}
  (begin
    (load "webapps/scheme/lib/db.scm")
    (load "webapps/scheme/lib/table.scm")
    (runquery "jdbc:hsqldb:hsql://localhost" "sa" "mynewpassword"
      {INSERT INTO survey VALUES([age],'[party]','[votedfor]')})
    {<html><head><title>Survey response</title></head></body>
     <h1> Thank you for your response</h1>
      The current tally is
     <table border=5 cellpadding=5 cellspacing=5>
      [(make-trs (runquery "jdbc:hsqldb:hsql://localhost"
                           "sa" "mynewpassword"
       {SELECT VOTEDFOR,count(*)  FROM survey GROUP BY VOTEDFOR}
       ))]
     </table></body></html>}
)))]
```

Figure 9.1: The database-backed survey servlet

```
(runquery
  HOST/DB USER PASSWORD
  QUERY)
```

where the HOST/DB is given by a string that refers to the SQL server and the USER and PASSWORD are the current system administrator USER and PASSWORD. [1] In our case, we will make two database queries. The first will insert the user's selections into the database. The second will summarize the voting totals for each listed candidate. Both of these queries have the following form:

```
(runquery
  "jdbc:hsqldb:hsql://localhost:9001" "sa" "mynewpassword"
  QUERY)
```

where QUERY is the actual SQL query that accesses or modifies the database. Note that we have set up a server on port 9001 which is serving the `webdb` database that contains the `survey` table we created above. This query provides access to that database and that table, assuming the server is running.

The first SQL query, that inserts the user's choices into the `survey` table is :

```
{INSERT INTO survey VALUES([age],'[party]','[votedfor]');}
```

The SQL keywords are capitalized and the words that we have created are lowercase. The `INSERT INTO` query always has the form

```
{INSERT INTO ..tablename... VALUES([value1],'[value2]',... );}
```

where the string values must be enclosed in single quote ('), but the numeric values should not be quoted.

Observe that the query is enclosed in curly braces "" so that it is just viewed as data by Scheme. The age, party, and votedfor variables are enclosed in square brackets "[]" indicating that their values as Scheme variables should be used. This may be a little confusing at first, but all SQL queries in Scheme servlets have this form. They are enclosed in curly braces and the parts come from the users data are enclosed in square brackets.

The next part of the servlet generates the webpage that will be returned to the user. It consists of the usual HTML code, but we escape into scheme to create the rows of an HTML table:

```
[(make-trs
  (runquery "jdbc:hsqldb:hsql://localhost" "sa" "mynewpassword"
   {SELECT VOTEDFOR,count(*)  FROM survey GROUP BY VOTEDFOR ;}
))]
```

[1] It would be safer to create a new user and grant them limited access to the database, but that will take us too far afield for the moment.

Note that we are again using the `runquery` procedure, but this time we are using the `SELECT` query. Its syntax will be described in more detail later in this section. This particular query returns a table whose rows consist of the distinct strings in the VOTEDFOR column of the table, together with a count of how often that string occurred in the table.

The `runquery` procedure always returns a list of lists. Each inner list corresponds to a row of the table, except the first, which is a row of column names. The `make-trs` converts this list into a sequence of HTML `tr` rows.

For this example, the "runquery" procedure returns a list of the voting totals of the form:

```
((Bush 1) (Gore 1) (Nader 2))
```

The "make-trs" procedure turns this into rows of an HTML table of the form

```
<tr><td>Bush </td><td> 1</td></tr>
<tr><td>Gore </td><td> 1</td></tr>
<tr><td>Nader</td><td> 2</td></tr>
```

9.4 A Quick Intro to SQL

A complete list of the SQL commands which are supported by `hsqldb` is given at the hsql website:

`http://hsqldb.sourceforge.net/internet/hSql.html`

In this section, we will give several examples of SQL queries that can illustrate these general commands.

9.4.1 Creating and removing tables

The "CREATE TABLE" command, creates a table in the current database. You must specify the name of the table as well as the names and types of each column. For example, the following command creates the "movies" table in the current database:

```
CREATE TABLE movies(title VARCHAR,minutes DECIMAL,rating VARCAR);
```

The name of the table is "movies" and it has three columns. The first and third columns are strings of characters and the middle column is a decimal number. The names of the three columns are "title,minutes,rating" respectively. Each column must have a name and a type. The simplest types are VARCHAR (for strings of characters), INTEGER (for whole numbers), and DECIMAL (for decimal numbers). Removing an entire table is easy:

```
DROP TABLE movies;
```

but note that this is a permanent operation. You cannot easily undo a dropped table.

9.4.2 Adding,Modifying, and Removing rows of tables

To insert a row into a table we use the `INSERT INTO` command. The values we insert must be in the same order in the `CREATE TABLE` command.

```
INSERT INTO movies VALUES('Star Warriors', 150, 'G');
```

Once values are in a table we may want to modify or delete them. We can remove all rows that meet some criterion using the `DELETE FROM` command.

```
DELETE FROM movies  WHERE title='Star Warriors';
DELETE FROM movies  WHERE minutes/60 > 3.5;
DELETE FROM movies  WHERE (rating='R') OR (rating = 'NR');
```

The "where" section allows you to specify a fairly complex condition using equality, arithmetic, comparison, and logic operations (AND, OR, NOT). Likewise, we can update individual columns in the rows of a table using the update command:

```
UPDATE movies SET rating='NC17' WHERE rating='X';
```

This would change all rows with an "X" rating to the new "NC17" rating.

9.4.3 Selecting rows of a table

One of the most common uses of databases is to select out some interesting subset of rows. This is done using the `SELECT` command For example, the following query returns a table with two columns (the title and the duration in hours) containing all movies with a G rating:

```
SELECT title,minutes/60 FROM movies WHERE rating='G';
```

You can also compute summary data use SQL. The following query computes the number of movies of each rating:

```
SELECT COUNT(*),rating FROM movies GROUP BY rating='G';
```

Chapter 10

Examples of Database Servlets

In this section we present some examples of servlets that interact with databases.

10.0.4 A webpage incorporating database content

Our first example is an expanded version of the survey demo in the previous example. The code appears in Figure 10.1.

In this example, the survey asks several questions and then provides various types of summary data. The overall structure of this first servlet is

```
{<html>...GENERATE HTML HEADER...
 <body style="color:black; background:white">
[(begin ... LOAD LIBRARIES...
  (let ((command (.getParameter request "command")))
    (if (equal? #null command)
       {... DISPLAY ALL MESSAGE IN DB,
            GENERATE LINK TO ADD A NEW MESSAGE:
            <a href="db1.servlet?command=newmsg">
               Add a new message</a>}
       (case command
        (("newmsg")
         {... GENERATE FORM GETTING MESSAGE INPUT FROM USER
              AND SENDING IT To db1.servlet with "command=addmsg"})
        (("addmsg")
         {... STORE USER'S MESSAGE IN THE DATABASE
           GENERATE AN ACKNOWLEDGEMENT, AND PROVIDE
           A LINK BACK TO THE TOP})
         (else {unknown command: [command]}))))))]
 </body></html>...GENERATE BOTTOM OF HTML PAGE... }
```

```
{<html><head><title>Simple DB Frontend Demo </title></head>
 <!--  this is a simple front end to a database
   it assumes the database contains a table created by
   CREATE TABLE news(d DATE, t TIME, subj VARCHAR, msg VARCHAR);
  -->
 <body style="color:black; background:white">

 [(begin ;; we need these libraries for file I/O and table making
   (load "webapps/scheme/lib/db.scm")
   (load "webapps/scheme/lib/table.scm")
   (let ((command (.getParameter request "command")))
   (if (equal? #null command)
    {<h1>Simple Frontend for the NEWS database</h1>
       <!-- display the links stored in the "msgs" file -->
       <table border=5>
        <tr><th>date</th><th>time</th>
            <th>subject</th><th>message</th></tr>
        [ (make-trs
           (rest
            (runquery "jdbc:hsqldb:hsql://localhost"
                      "sa" "mynewpassword"
              {SELECT * FROM news})))]
       </table>
        <br><a href="db1.servlet?command=newmsg">
            Add a new message</a>}

   (case command
    (("newmsg")
      {<form method="post" action="db1.servlet">
        <input type="hidden" name="command" value="addmsg">
        Subject:<br><input type="text" name="subject">
        <br>Message:
        <br><textarea name="msg" rows=10 cols=60></textarea>
        <br><input type="submit">
       </form>})
   (("addmsg")
    (let ((subject (.getParameter request "subject"))
          (msg     (.getParameter request "msg")))
      (runquery "jdbc:hsqldb:hsql://localhost"
                "sa" "mynewpassword"
        {INSERT INTO news
           VALUES(CURDATE(),CURTIME(),'[subject]','[msg]')})

      {Posted message<br>Subject:<br>[subject]<br>Message:
       <br>[msg]<br><a href="db1.servlet">back to top</a>}))

   (else {unknown command: [command]})))))]
 </body> </html>}
```

Figure 10.1: The database-frontend servlet

Thus, this servlet generates three different pages depending on the value of the parameter "command." If it is "#null" then the servlet shows the "front page" which displays all the messages in the database and provides a link to the "newmsg" page. The "newmsg" page is just an HTML form soliciting the desired info from the user and sending the data back to itself, but with "command=addmsg" This last page stores the user data in the database and generates an "thank you" page with a link back to the front page.

The front page is generated by the following brls code:

```
{<h1>Simple Frontend for the NEWS database</h1>
   <!-- display the links stored in the "msgs" file -->
   <table border=5>
    <tr><th>date</th><th>time</th>
        <th>subject</th><th>message</th></tr>
    [ (make-trs
       (rest
        (runquery "jdbc:hsqldb:hsql://localhost"
          "sa" "mynewpassword"
          {SELECT * FROM news;})))]
   </table>
    <br><a href="db1.servlet?command=newmsg">
        Add a new message</a>  }
```

which is mostly static HTML except for the table. The SQL query to retrieve all rows in the table is

```
SELECT * FROM news;
```

The `rest` procedure removes the first row from the result (the first row is always a list of the column names).

The "newmsg" page generates an HTML form requesting the subject and message from the user:

```
{<form method="post" action="db1.servlet">
  <input type="hidden" name="command" value="addmsg">
  Subject:<br><input type="text" name="subject">
  <br>Message:
  <br><textarea name="msg" rows=10 cols=60></textarea>
  <br><input type="submit">
 </form>}
```

There is nothing new here except for the use of the "hidden" input element. This input element sets the value of the "command" parameter to "addmsg", but it does not appear on the user's page. It's sole purpose is to tell the servlet which subpage the data should be processed by.

The "addmsg" page stores the user's data in the database, shows the user what has been stored, and provides a link back to the front page:

```
(let ((subject (.getParameter request "subject"))
      (msg     (.getParameter request "msg")))
  (runquery "jdbc:hsqldb:hsql://localhost"
    "sa" "mynewpassword"
    {INSERT INTO news
        VALUES(CURDATE(),CURTIME(),'[subject]','[msg]');})

  {Posted message<br>Subject:<br>[subject]<br>Message:
    <br>[msg]<br><a href="db1.servlet">back to top</a>})
```

The SQL query for inserting the data show the use of two SQL procedures `CURDATE()` and `CURTIME()` which return the current data and the current time respectively.

10.0.5 Extensions

There are many ways that this example can be extended. We discuss a few of them here.

10.0.6 Deleting rows

One useful extension is to allow users to delete as well as add messages. To make this easy we would like to have each row in the database have a unique number, so we could just say delete row 37 or delete all rows numbered less than 19. The best way to do this is to create the table with an additional column that will be automatically assigned an "identity" number each time a new row is inserted. This is done using the "INTEGER IDENTITY" type. So, for our example, the CREATE TABLE statement would need to be rewritten as

```
CREATE TABLE news
 (d DATE, t TIME, subj VARCHAR, msg VARCHAR, n INTEGER IDENTITY);
```

This also affects the other SQL queries. Since we now have an additional column, the INSERT commands have to include a NULL in the last position. This will cause the database engine to automatically insert the correct value in the table.

```
{INSERT INTO news
  VALUES(CURDATE(),CURTIME(),'[subject]','[msg]',NULL);})
```

The SQL for deleting the Nth message is then

```
{DELETE FROM news WHERE n=[N];}
```

10.0.7 Adding Password Protection

It would also be a good idea to require the user to present a password before letting them modify the database. This can be easily done using the HTML form. Just add a "password" input element and then modify the "runquery" procedure call to use the value of the password parameter

```
(runquery "jdbc:hsqldb:hsql://localhost"
   "sa" (.getParameter request "password")
    {......})
```

10.1 The database-backed webpage

Our next example is a webpage that uses the "news" table in the "webdb" database to add database content to a web page. The code for this servlet is in Figure 10.2. This page is quite similar to the previous one in that it generates a table from the database data. What is different about this demo is that it only shows the first row of the database and provides links which allow the user to access the first 10 or first 1000 messages. The procedure to return the first N elements of a list is defined, right after the libraries are loaded, by the following code:

```
(define (firstN N L)
  (if (or (< N 1) (null? L)) ()
      (cons (first L) (firstN (- N 1) (rest L)))))
```

This uses all four of the major "list" operations of Scheme:

- `(cons X L)` – creates a new list by putting the element X at the beginning of the list L, so `(cons 'a '(b c d))` returns the list `'(a b c d)`
- `(first L)` – returns the first element of a list so `(first '(a b c d)` returns the element `a`.
- `(rest L)` – returns a copy of the list L, but with the first element removed, so so `(rest'(a b c d)` returns the list `'(b c d)`
- `(null? L)` – returns the true value (`#t`) if `L` is the empty list, and returns the false value `#f`, otherwise.

The servlet also demonstrates some error checking when it is trying to compute the value of the variable `nummsg` which will be used to determine how many rows of the table to display. If the parameter `nummsgs` does not have a value (for example upon visiting the page for the first time), then `nummsgs1` will have the value `#null`, and trying to convert this into a number using `Double.` will generate an exception. The servlet handles this case, by trying to catch that exception, and returning 1 if indeed there is any error converting `nummsgs1` into a number. This is done with the

```
(tryCatch EXPR1 (lambda(e) EXPR2))
```

expression which first tries to evaluate EXPR1 to get the return value. If there is an exception, then it evaluates EXPR2 to either get the return value, or throw an exception.

```
[(begin ;; we need these libraries for file I/O and table making
   (load "webapps/scheme/lib/db.scm")
   (load "webapps/scheme/lib/table.scm")
   (define (firstN N L)
     (if (< N 1) ()
         (cons (first L) (firstN (- N 1) (rest L)))))

  (let* (
         (nummsgs1 (.getParameter request "nummsgs"))
         (nummsg (tryCatch (Double. nummsgs1) (lambda(e) 1)))
        )
 {<html><head><title>Simple DB-backed webpage Demo
       </title></head>
 <!--  this is a simple webpage
       getting some of its data from a database -->
 <body style="color:black; background:white">
    <h1>The DBDEMO project</h1>
     This is the main page of the DBDEMO project:
     <table border=10 width=80%>
       <tr>
         <td style="font:bold 40pt serif;text-align:center">
        STATIC CONTENT GOES HERE
       </td></tr></table>
     <br>
     <h2>Latest News</h2>
       <table border=5>
        [(if (> nummsg 1)
           {<tr><th>date</th><th>time</th>
                <th>subject</th><th>message</th></tr>}
           "")]
        [(make-trs
          (firstN nummsg
             (rest
               (runquery "jdbc:hsqldb:hsql://localhost"
                  "sa" "mynewpassword"
                   {SELECT * FROM news ORDER BY d,t desc;}))))]
       </table>
      <br>
      <a href="db2.servlet?nummsgs=1000">View all messages</a><br>
      <a href="db2.servlet?nummsgs=10">View last 10 messages</a>
 </body>
 </html>
}))]
```

Figure 10.2: The database-backed webpage

10.2 A Network Database Front-end

In this section, we discuss a simple graphical user interface to a database. The code is in Figure 10.3.

Observe that most of the program is concerned with setting up the interface (creating and naming the components, specifying how they should be laid out on the screen, etc.). The only place where the database is involved is in the "submitquery" procedure.

```
(define (submitquery query)
 (tryCatch
      (runquery (readstring (t "host/db"))
                        (readstring (t "user"))
                        (readstring (t "pw")) query)
      (lambda(e) (list "ERROR: " e))))
```

This procedure reads the appropriate textfields to determine the URL of the database and the user/password information. It then creates sends the query to that database and returns the result. If there is an error then it returns a list containing the error.

This program also shows a way to get access to the a component without using a tagger. In the action for the "choice" component, the component itself is obtained using the ".getSource" procedure applied to the event "e". This is useful when one only needs to refer to a component in the component's action code.

Exercise 23 *Revise this database example so that it provides a custom front end to a particular database, i.e. so that it allows the user to add, delete, and search in the database just by pushing buttons and filling out fields. All the SQL querying should go on in the actions that you create.*

Exercise 24 *Write a general purpose GUI that allows the user to interact with a general database (specified in the "host/db" field) without having to know any SQL. These DB clients are quite popular and are not too hard to create.*

```
(jlib.JLIB.load)
(load "db.scm")
;; this is webapps/scheme/lib/db.scm file(define tag (maketagger))

(define tag (maketagger))
(define win
 (window "DB DEMO"
     (border
        (north (col 'horizontal
           (label "JDBC Demo Page" (HelveticaBold 24))
           (table 3 2
                 (label "DatabaseURL")
                 (tag "host/db"
                    (textfield "jdbc:hsqldb:hsql://localhost" 50))
                 (label "User")
                     (tag "user" (textfield "sa" 50))
                 (label "Password")
                     (tag "pw" (textfield "mynewpassword" 50)))
           (row
             (choice "#sample queries"
               "create table demo(name VARCHAR, price DECIMAL);"
                   "drop table demo;"
                   "insert into demo values('abc','1123');"
                   "select * from demo where price > 500;"
                   "delete from demo where name='abc';"
                   (action (lambda(e)
                     (writestring (tag "query")
                     (readstring (.getSource e)))))))
           (tag "query" (textarea 5 50
           "create table demo(name VARCHAR, price DECIMAL);"))
             (row 'none
                 (button " SEND QUERY "
                         java.awt.Color.white$ (HelveticaBold 18)
                   (action (lambda (e)
                    (showresults
                    (submitquery (readstring (tag "query")))))))
                 (button "clear" (action (lambda(e)
                      (writestring (tag "results") ""))))))
       (center
           (tag "results" (textarea 10 50))))))
(define (showresults results)
   (for-each (lambda(x) (appendlnexpr (tag "results") x))
                    results))
(define (submitquery query)
(display (list 'runquery (readstring (tag "host/db"))
                         (readstring (tag "user"))
                         (readstring (tag "pw")) query)) (newline))

 (tryCatch
      (runquery (readstring (tag "host/db"))
                (readstring (tag "user"))
                (readstring (tag "pw"))
                query)
      (lambda(e) (list "ERROR: " e))))
(.pack win) (.show win)
```

Chapter 11

Peer to Peer programming

In this Chapter we provide an introduction to peer-to-peer computing. The first main example we will discuss is a chat room. The second is an enhanced chat program that manages multiple rooms and lets you join multiple rooms with multiple aliases.

11.1 Group-servers and Group-clients

The chat room examples will be implemented using a library (jlib.Networking) that allows us to create group-servers and group-clients. The group-clients are simple objects that allow you to

- join a group server, by selecting a name and group and specifying the host and port of the group server,
- send messages to everyone in the group (yourself included)
- specify actions that should be performed whenever a message arrives from the group

11.2 Starting a group server

You can start a group server by opening a command window and giving the following commands:

```
 java -cp jscheme.jar jscheme.REPL
> (jlib.Networking.load)
> (define S (make-group-server 23232))
>
```

This will start a group server running on port 23232 of your computer. (As usual, this assumes that you are in the directory containing jscheme.jar). You can shutdown the server by giving the command

```
 > (exit)
```

or by shutting down your computer!

You can test your group-server by opening another command window (possible on a different computer) and giving the following commands:

```
java -cp jscheme.jar jscheme.REPL
> (jlib.Networking.load)
> (define H "127.0.0.1")
;; use the IP address of your group server here

> (define P 23232)
 ;; use the port of your group server here

> (define G (make-group-client "me" "test" H P))
  ; connect to server

> (G 'add-listener 'mykey (lambda R (display R) (newline)))
   ;; this causes group-client to display msgs with key= 'mykey

> (G 'send 'mykey "Hello World" (* 1111 1111))
;;;; a response should appear here ....

> (G 'send 'otherkey "Hello World Again" 'a 'b 'c)
;;;; no response should appear here
;;;; as there is no listener on 'otherkey
```

Exercise 25 *Startup a server on one machine and two clients on two other machines and use the two clients to chat. Do this by setting up the listener as shown above and then making G calls of the form*

```
 (G 'send 'mykey "joe:.....")
```

to send messages.

11.3 A Simple Chatroom applet

Next we assume that you have started a group-server on port 23232 on your computer and you have also started a tomcat server with a jscheme webapp. If you insert the program in Figure 11.1 in the webapps/scheme folder, then it will provide users access to a single chatroom. The first part of the applet contains the documentation used to construct the applet page. The second part of the applet contains a procedure which creates a chat window given the username, chatgroup, host, and port of the group server. Most of this code is standard GUI code like that we have already seen. The three interesting parts of the program are where the group client is created

```
   (define S (make-group-client UserName ChatGroup Host Port))
```

```
"chat1a.applet"
"Tim Hickey"
""
"This is a simple example of a chat applet"
""

(begin
  (jlib.JLIB.load)
  (jlib.Networking.load)
  (let ()
    (define (chatwin UserName ChatGroup Host Port)
      (define t (maketagger))
      (define S (make-group-client UserName ChatGroup Host Port))
      (define w (window "test"
        (col
          (button "quit" (action (lambda (e)
                                   (S 'logout) (.hide w))))
            (t "chatarea" (textarea 20 50))
            (t "chatline" (textfield "" 50 (action (lambda(e)
               (S 'send "chat"
      (string-append UserName ": " (readstring (t "chatline"))))
               (writeexpr (t "chatline") "")
        )))))))
      (S 'add-listener "chat" (lambda R
       (appendlnexpr (t "chatarea") R)))
      (.pack w) (.show w)
      w
     )

   (define (rand N) (Math.round (* N (Math.random))))

   (chatwin
      (string-append "user-" (rand 1000))
      "chat"
      (.getHost (.getDocumentBase thisApplet))
      23456)
))
```

Figure 11.1: A multi-room chat program

and the action of the textfield, which reads what the user has written in the textfield and sends it to the group server with the keyword "chat"

```
(S 'send "chat"
 (string-append UserName ": " (readstring (t "chatline"))))
```

and finally the part of the program in which we add a listener to handle the messages that come in from the group server (including our own messages). In this case, we just append the messages with keyword "chat" to the end of the chatarea.

```
(S 'add-listener "chat" (lambda R
 (appendlnexpr (t "chatarea") R)))
```

Observe that the use of the keyword allows us to carry on several simultaneous group conversations on different topics with the same group client. Finally when the user presses the quit button, we logout of the group-client which causes the group-client to close its connection to the group-server

```
(S 'logout) (.hide w)
```

You might try to modify this applet so that it censors any message containing any of a small list of unsavory words (e.g. SH*T, but note that this might unfairly single out message about shitake mushrooms.)

You might also add a password feature, or look into requiring each user to have a unique user name. You could hook up this applet to access a database of usernames and passwords of registered clients if you want a little more security.

11.4 A Multiroom Chat applet

In this section we show how to extend the previous example to allow for multiple rooms and to allow for some querying as to who is online.

The key idea here is to create a registration window that allows the user to ask the group-server questions about the user, groups, etc. currently being served. This is done using the "send-line" procedure from the "Networking.scm" library. The group-server will respond to three queries (as shown below) with a list of the users, groups, and users-in-group. For example, the action on the following button queries the server for a list of the active groups and then displays that list in the textarea:

```
(button "show groups" (action (lambda(e)
 (writeexpr (t "serverinfo")
     (send-line Host Port "groups\n")))))
```

This window also allows the user to startup a new chat window using the specified user/group/host/port to create a chatwindow. The code for the chat window is exactly as shown in the previous example, so we don't repeat it here.

The "send-line" procedure sends a string of characters to the specified host/port. You can use it to get the current local time on a server by sending an empty message to port 13

```
(send-line "129.64.2.3" 13 "\n")
```

Here we have sent a carriage return, but using the empty string "" would probably suffice for most servers anyway.

```
"chat2.applet"
"Tim Hickey"
""
"multi-room chat GUI"
""

(begin
  (jlib.JLIB.load) (jlib.Networking.load)

  (define (rand N) (Math.round (* N (Math.random))))
  (define (chatwin ....) ...) ;; as in previous example

  (define (make-register-window Host Port)
    (define t (maketagger))
    (define regwin (window "register"
      (border
        (center (t "serverinfo" (textarea 10 40)))
        (north
          (border
            (north (col
               (t "group" (textfield "test" 20))
               (t "user" (textfield "guest" 20))))
            (south
             (row
               (button "show users" (action (lambda(e)
                (writeexpr (t "serverinfo")
                    (send-line Host Port "users\n")))))
               (button "show groups" (action (lambda(e)
                (writeexpr (t "serverinfo")
                    (send-line Host Port "groups\n")))))
               (button "show users in group" (action (lambda(e)
                (writeexpr (t "serverinfo")
                     (send-line Host Port (string-append
                          "users-in-group\n"
                         (readstring (t "group"))))))))
               (button "join group" (action (lambda(e)
                   (make-chat (readstring (t "user"))
                       (readstring (t "group")) Host Port)
                 )))
            )))))
      (.pack regwin) (.show regwin) regwin   )

    (register-window
       (tryCatch
         (.getHost (.getDocumentBase thisApplet))
         (lambda(e) "127.0.0.1"))
      23456))
```

Figure 11.2: A multi-room chat program

Chapter 12

Examples of P2P Programming

This chapter is not yet ready.

12.1 Servents

12.2 Network-aware GUI components

12.3 Running a chat registrar

12.4 Whiteboards

12.5 Shared Textareas

Part VI

Appendices

Appendix A

The Tomcat server and Jscheme

A.1 Installation instructions for Mac/Linux

The Tomcat server will only run on your Mac if you have the Mac OS 10.1 operating system or higher. It should run under any Linux distribution. We assume from now on that you do have one of these operating systems and that you know how to run a browswer and open a terminal window

A.1.1 Installing tomcat

The installation process is fairly easy. First you need to open a browser and download the jscheme-tomcat.tgz file from the website

```
http://www.cs.brandeis.edu/~tim/Downloads/jscheme-tomcat.tgz
```

and store it somewhere on your system. Lets say your username is Joe and you store it in

```
/Users/joe/jscheme-tomcat.tgz
```

You can then close the browser.

Next you unpack it by opening a terminal window, go to the directory where you downloaded the tgz file and giving the unpack command as follows:

```
% tar xvzf jscheme-tomcat.tgz
```

This will create a folder called `tomcat` in that directory.

A.1.2 Starting/Stopping the Tomcat Server

Next, go into the tomcat directory and set the JAVA_HOME variable to be the path to the 1.3+ version of the Java Development Kit.

On the Mac, this is currently done by the following command:

```
% cd tomcat
% setenv JAVA_HOME /Systems/Library/Frameworks/JavaVM.framework/Versions/1.3.1/Home
```

Under linux its the same, but the JAVA_HOME is usually "/usr/java/jdk1.3". You can run the "locate javac" command to find where your Java files are located.

Now you are ready to start the server by issuing the command:

```
% bin/startup.sh
```

You can stop the server at any time by issuing the command

```
% bin/shutdown.sh
```

Or my shutting down your computer.

A.1.3 Adding content to the server

Finally, to add webcontent to your server, you should change directory into the `tomcat/webapps` folder. This shows all of the webapps that your server is currently delivering. Of particular interest for us is the `scheme` webapp. Change directory into the `scheme` folder and any files you create in here (or in subfolders) will be served on the web. Moreover, files ending in sssp, sxml, brls, applet, snlp will be processed as described in the main body of this text.

A.1.4 Accessing the server

Your server is now running on your machine on port 8080. If your machine has a registered domain name, say /tt mymachine.com, then you can access the scheme files on your server at the web address:

```
http://mymachine.com:8080/scheme
```

If you don't have a registered domain name (which is most often the case) then you must use the IP address itself as the address.

To find out your IP address on a MAC, you can select the "System Preferences" item in the "apple menu," then select Network, and have it show "Built-in Ethernet" or "Airport," whichever you are using. The screen will then show your IP address.

On a Mac or under Linux, you can also give the command

```
% ifconfig -a
```

and it will give you your IP address (along with lots of other info!)

A.2 Installation instructions for the PC

The Tomcat server will only run on your PC if you have also installed the Java JDK version 1.2 or higher. Older versions of windows have this preinstalled, but Microsoft decided not to preinstall Java in XP. You can download an install Java by visiting

```
http://java.sun.com/j2se/1.3
```

and selecting the "Microsoft Windows" download option.

We assume from now on that you do have this operating system and that you know how to run a browswer and open a dos window (start-¿programs-¿accessories-¿commandprompt).

A.2.1 Installing tomcat

The installation process is fairly easy. First you need to open a browser and download the jscheme-tomcat.zip file from the website

```
http://www.cs.brandeis.edu/~tim/Downloads/jscheme-tomcat.zip
```

and store it somewhere on your system. Lets say you store it at the toplevel in

```
C:jscheme-tomcat.zip
```

You can then close the browser.

Next you unpack it by opening a terminal window, go to the directory where you downloaded the tgz file and giving the unpack command as follows:

```
% cd C:
% unzip jscheme-tomcat.zip
```

This will create a folder called `C:tomcat`.

A.2.2 Starting/Stopping the Tomcat Server

Next, go into the tomcat directory and set the JAVA_HOME variable to be the path to the 1.3+ version of the Java Development Kit. This is currently done by the following command:

```
% cd C:tomcat
% set JAVA_HOME=C:\jdk1.3.1_02
```

where you will need to specify where Java is on your system. It usually installs in the top level of the C drive and has a name that begins with "jdk" Now you are ready to start the server by issuing the command:

```
% bin\startup.sh
```

You can stop the server at any time by issuing the command

```
% bin\shutdown.sh
```

Or my shutting down your computer.

If you get an "Out of environment memory error," when trying to start up the server, then you will need to increase the default size of the environment in the MSDOS window. Do this by right clicking on the MSDOS icon in the upper left corner of the DOS window. Then click on the memory tab, and then select the largest value for the initial environment (e.g. around 4000). Shutting down the MSDOS window (by typing "exit") and opening a new MSDOS window should eliminate that error. Just reenter the JAVA_HOME and startup the server.

A.2.3 Adding content to the server

Finally, to add webcontent to your server, you should change directory into the `webapps` folder. This shows all of the webapps that your server is currently delivering. Of particular interest for us is the `scheme` webapp. Change directory into the `scheme` folder and any files you create in here (or in subfolders) will be served on the web. Moreover, files ending in sssp, sxml, brls, applet, snlp will be processed as described in the main body of this text. Note that you must store these files as plain text (not as HTML, or Microsoft Word)

A.2.4 Accessing the server

Your server is now running on your machine on port 8080. If your machine has a registered domain name, say /tt mymachine.com, then you can access the scheme files on your server at the web address:

```
http://mymachine.com:8080/scheme
```

If you don't have a registered domain name (which is most often the case) then you must use the IP address itself as the address. You get the IP address by going to (start-¿.....)

A.3 Creating a certificate for secure web pages

THIS SECTION IS IN PREPARATION.

A.4 Configuring for email

THIS SECTION IS IN PREPARATION.

Appendix B

Installing the HSQL Database

Download instructions.

Starting the Server.

Starting the DatabaseManager

Changing the System administrator password.

Appendix C

Code for the servlet and applet libraries

C.1 Files.scm

```
;; files.scm
(define (write-to-file filename data)
   (let ((f (java.io.PrintWriter. (java.io.FileWriter. filename #f))))
      (.println f (string-append  data))
      (.close f)))

(define (append-to-file filename data)
  (let ((f (java.io.PrintWriter. (java.io.FileWriter. filename #t))))
      (.println f  (string-append data))
      (.close f)))

(define (read-from-file name default)
   (tryCatch
     (let* (
         (f (open-input-file name))
         (data (read f)))
       (.close f)
       data)
     (lambda(e) default)))

(define (read-all-from-file name default)
   (tryCatch
     (let* ((f (open-input-file name))
            (data
              (let loop ((x (read f)))
                (if (eof-object? x) ()
                 (cons x (loop (read f)))))))
       (.close f)
       data)
     (lambda(e) default)))

(define (read-string-from-file name default)
   (tryCatch
     (let* ((f (java.io.BufferedReader. (java.io.FileReader. name)))
            (data
              (let loop ((x (.readLine f)))
                 (if (equal? x #null) ()
                     (cons (string-append x "\n")
                           (loop (.readLine f)))))))
       (.close f)
       (apply string-append data))
     (lambda(e) default)))

(define (servlet-file request name)
  {webapps[(.getRequestURI request)]_[name]}  )
```

C.2 mail.scm

```
(define (send-mail request to from subj text)
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   (define smtp-host "smtp.cs.brandeis.edu")
        ;;; You must change this to your smtp host !
   (define smtp-protocol "imap")
        ;;; and change this to your protocol
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (define (getRealPath servlet File)
    (.getRealPath
       (.getServletContext (.getServletConfig servlet)) File))

  (define (send to from subj text)
   (define (isnull? x)  (or (equal? x "") (eq? x #null)))

   (define smtp-mbox "INBOX")
   (define props
    (let ((x (java.lang.System.getProperties)))
       (.put x "mail.smtp.host" smtp-host)
       x))
   (define ses (javax.mail.Session.getDefaultInstance props #null))

   (define (sendmail to from subj text msg)
(begin
  (if (not (isnull? to))
      (begin
(let ((toAddrs
               (javax.mail.internet.InternetAddress.parse to #f))
      (myto javax.mail.Message$RecipientType.TO$))
   (.setRecipients msg myto toAddrs))))
  (if (not (isnull? from))
      (begin
(let ((fromAddr
               (javax.mail.internet.InternetAddress. from)))
   (.setFrom msg fromAddr))))
         (if (not (isnull? subj))
      (.setSubject msg subj))
  (if (not (isnull? text))
      (.setText msg text))
  (javax.mail.Transport.send msg)))
   (sendmail to from subj text (javax.mail.internet.MimeMessage. ses)))

  (send to from subj (.toString text))
)
```

Bibliography

[1] World Wide Web Consortium webpages.