

CGI Scripting in Scheme

- The Basic Steps
 - Stage One: Prepare a Scheme Procedure
 - Stage Two: Extend Scheme File
 - Stage Three: Build CGI Wrapper
 - Stage Four: Build HTML Interface
 - Stage Five: Put it all together
- A Sample Procedure
- HTML Forms
- Sample Code

As you may know, it is possible to write programs that generate Web pages in response to user input (often entered on other Web pages). The most common mechanism for writing such programs is CGI, which stands for *common gateway interface*. It is possible to write CGI programs (usually called "scripts") in almost any programming language, including Scheme.

Here's how CGI typically works. The user enters information on a form (simply an HTML page with some special tags). When the user clicks on the **Submit** button (or something similar) the browsers packages up all the information and sends it to the server. The server then unpackages the information and passes it on to the CGI script. The CGI script looks at the information and generates a page and passes the page back to the server. The server passes the page back to the browser, and everyone is happy. So, what do you need to learn? You already know how to write Scheme to generate an HTML page (or you should if you did the strings lab). So, you have to learn how to get information from the server, tell the server to run your program from a CGI script, and build the HTML page that lets someone enter information.

I've set up some helper programs that make it easier (but not necessarily easy) for you to write CGI scripts in Scheme. I also recommend that you follow a standard process when building your scripts.

The Basic Steps

Here are the steps that I recommend that you undertake when writing a CGI script.

Stage One: Prepare a Scheme Procedure

Let's start with what you should already mostly know how to do.

1. Decide the *purpose* of the script: what you want the script to do. Someone (i.e., me) might tell you what to do or you might choose on your own. For example, you might want to write a program that generates a simple greeting page.
2. Determine what *input* your script will take. For example, your greeting script might take a user's name as input we'll call that input `user`
3. Write a Scheme procedure that takes as a parameter the input decided upon in the previous step and

generates an appropriate HTML page. See below for a sample procedure.

4. Save
5. Test that procedure! that procedure in a `.ss` file. For example, `greetings.ss`. The file should be in your `public_html` directory.
6. Make that file world-readable with

```
% share filename
```

But don't type the percent sign!

Stage Two: Extend Scheme File

Now you need to extend the file so that it works with the CGI system (in terms of getting input from the CGI system and working when called from the CGI system). The way I've set up my CGI gateway, it always calls a procedure named `page` that has no parameters.

1. Add the following line to the top of your Scheme file (after your introductory comments):

```
(require-library "cgi.ss" "net")
```

This line tells Scheme that you need the special utility procedures for dealing with CGI scripts.

2. Add a line to the top of your Scheme file of the following form:

```
(define testing #t)
```

This line helps you test your program (as long as you follow all my instructions).

3. Add lines to your Scheme file of the following form

```
(define desired-variable
  (if testing "default"
    (extract-binding/single 'name (get-bindings))))
```

For example,

```
(define user
  (if testing "Sam"
    (extract-binding/single 'user (get-bindings))))
```

4. Add another procedure to your Scheme file that is named `page` and takes no parameters. This procedure should call your first procedure using the values extracted above. For example,

```
(define page
  (lambda ()
    (greeting-page user)))
```

In order for my helper stuff to work correctly, this *must* be named `page`!

5. Test your stuff by loading this into DrScheme and typing `(page)`.
6. Once you've verified that things work correctly, change the value of `testing` to `#f`.

Stage Three: Build CGI Wrapper

Now it's time to build something that ties your Scheme program to the CGI system.

1. Build a file of the following form. Make sure that there are no spaces at the beginning of any line:

```
#!/bin/bash
/home/rebelsky/bin/schemeweb your-scheme-file
```

For example,

```
#!/bin/bash
/home/rebelsky/bin/schemeweb greeting.ss
```

Note that you should always use the `/home/rebelsky/bin` no matter who you are!

2. Save that file in your `public_html` directory as something that ends in `.cgi`. For example, `greetings.cgi`.
3. Tell Unix that you've written a program with

```
% chmod a+x filename.cgi
% chmod a+r filename.cgi
```

Once again, don't type the percent sign.

4. Test the CGI interface by using a URL like the following:

```
http://www.cs.grinnell.edu/~username/file.cgi?name=value
```

For example,

```
http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2001S/Examples/greetings.cgi?user=SamR
```

Only use letters and numbers for testing!

Stage Four: Build HTML Interface

Now you're ready to build the HTML page that people will use.

1. Write an HTML page that can provide input to your page. The input page is typically called a form.

Stage Five: Put it all together

1. Test, test, and test again.

A Sample Procedure

Here is a Scheme procedure that generates a greeting page based on an input name. It assumes that you've written some helper procedures similar to those from the strings lab.

```

;;; Procedure
;;;   greeting-page
;;; Parameters:
;;;   user, a string that represents the name of a user
;;; Purpose:
;;;   Generates an HTML page that can greet the named person.
;;; Produces:
;;;   A string that corresponds to that page.
;;; Preconditions:
;;;   The string must be nonempty.
;;; Postconditions:
;;;   The returned page is valid HTML.
(define greeting-page
  (lambda (user)
    (make-page
     (head "Greetings")
     (body (string-append
            (heading 1 "Welcome")
            (string #\newline)
            (paragraph (string-append "Hi " user "!")))))))

```

HTML Forms

As mentioned above, a form is simply some HTML that lets the user enter some input and ties that input to a CGI script. (Forms can do other things, too, but this purpose is enough for now).

As you might expect, you surround a form with `form` tags. The opening tag must also include two parameters:

- `method`, which can be `put` or `get`. I recommend that you use `get` while testing. However, if you expect the user to enter a lot of input, use `put`.
- `action`, which should be the URL of your CGI script. It can just be the name of your script if the HTML file is in the same directory.

For example,

```

<form method="get" action="greeting.cgi">
...
</form>

```

You can include normal HTML in a form. Normal HTML is displayed normally and is not uploaded to the server when the user submits values. Rather, it is used to give some information to the user about where things might go.

When you want the user to type a value, you use an `input` tag with the following parameters:

- `type`, which must be `text`
- `name`, which is the name you want assigned to the value in the field. For our example, this will be `user`.
- `value`, which is the default value.

For example,

```
<input type="text" name="user" value="">
```

You should also provide a handy-dandy "Click Here" button. Once again, you use an `input` tag. This time, you use slightly different parameters:

- `type`, which must be `submit`
- `value`, which provides the text in the button

For example

```
<input type="submit" value="Enter your name and click here!">
```

That's it, you know the basics. If you want things other than fields in your forms, ask me about them in class or look them up on the Web.

Sample Code

`greeting.ss`

```
;;; File:
;;;   greeting.ss
;;; Version:
;;;   1.0 of February 2001
;;; Author:
;;;   Samuel A. Rebelsky
;;; Contents:
;;;   A simple example of some Scheme procedures for CGI
;;; Organization:
;;;   Preparation - Stuff I need to do to get started
;;;   Sam's HTML Helpers - Procedures for generating HTML
;;;   Page Generators - Procedures for generating specific kinds of
;;;     pages (when given particular values)
;;;   Interface - Stuff to extract important values and then
;;;     call the page generators.  You'll find (page) here.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Preparation

;;; Look!  We're going to use the CGI library.  What fun.
(require-library "cgi.ss" "net")

;;; Are we testing?
(define testing #f)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Sam's HTML Helpers

;;; Procedure:
;;;   markup
;;; Parameters:
;;;   text, some text to mark up
;;;   tag, the role that the text plays
```

```

;;; Purpose:
;;; Generates some nice happy HTML for me.
;;; Produces:
;;; A string for HTML that represents the marked text.
;;; Preconditions:
;;; Both parameters are strings.
;;; The text is valid HTML.
;;; The tag contains only alphanumeric characters and
;;; corresponds to a valid HTML tag.
;;; Postconditions:
;;; You get some nice HTML.
(define markup
  (lambda (text tag)
    (string-append "<" tag ">" text "</" tag ">")))

;;; Procedure:
;;; heading
;;; Parameters:
;;; level, a heading level (1-6)
;;; text, some text to mark up
;;; Purpose:
;;; Generates some nice happy HTML that represents the heading of
;;; a section of text.
;;; Produces:
;;; A string for HTML that represents the heading.
;;; Preconditions:
;;; Level is an exact integer between 1 and 6, inclusive.
;;; text is a string.
;;; text is valid HTML.
;;; Postconditions:
;;; You get some nice HTML which a normal browser displays
;;; as a heading.
(define heading
  (lambda (level text)
    (markup text (string-append "h" (number->string level)))))

;;; Procedure:
;;; head
;;; Parameters:
;;; title, the title of the page
;;; Purpose:
;;; Generates some nice happy HTML that represents the head of
;;; a page with a particular title.
;;; Produces:
;;; A string for HTML that represents the head.
;;; Preconditions:
;;; The title is a string with no internal HTML.
;;; Postconditions:
;;; You get some nice HTML which a normal browser displays
;;; as the titlebar of a window.
(define head
  (lambda (title)
    (markup (markup title "title") "head")))

;;; Procedure:
;;; body
;;; Parameters:

```

```

;;; contents, the intended contents of a page
;;; Purpose:
;;; Generates some nice happy HTML that represents the body of
;;; a page with a particular contents.
;;; Produces:
;;; A string for HTML that represents the body.
;;; Preconditions:
;;; contents is a string
;;; contents represents valid, though partial, HTML
;;; Postconditions:
;;; You get some nice HTML which you can join with a
;;; head and some html tags to make a full page.
;;; The result may contain carriage returns (newlines).
(define body
  (lambda (contents)
    (markup (string-append (string #\newline)
                           contents
                           (string #\newline))
            "body"))))

;;; Procedure:
;;; paragraph
;;; Parameters:
;;; contents, the contents of the paragraph
;;; Purpose:
;;; Generates some nice happy HTML that represents on paragraph.
;;; Also to annoy the world with introductory comments that are
;;; significantly longer than the procedure needs.
;;; Produces:
;;; A string for HTML that represents the paragraph.
;;; Preconditions:
;;; The contents is (are?) valid HTML with no block-level elements.
;;; Postconditions:
;;; You get some nice HTML which a normal browser displays
;;; appropriately.
(define paragraph
  (lambda (contents)
    (markup contents "p"))))

;;; Procedure:
;;; paragraph
;;; Parameters:
;;; head, the head of a page
;;; body, the body of a page
;;; Purpose:
;;; Shoves the head and the body together to make a page.
;;; Produces:
;;; HTML for a full page. Wow!
;;; Preconditions:
;;; head was created by the head procedure
;;; body was created by the body procedure
;;; Postconditions:
;;; You get some nice HTML which a normal browser displays
;;; appropriately.
(define make-page
  (lambda (head body)
    (markup (string-append (string #\newline)
                           head
                           (string #\newline)
                           body
                           (string #\newline))
            "html"))))

```

```

        head
        (string #\newline)
        body
        (string #\newline))
    "html"))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Page Generators

;;; Procedure
;;;   greeting-page
;;; Parameters:
;;;   name, a string
;;; Purpose:
;;;   Generates an HTML page that can greet the named person.
;;; Produces:
;;;   A string that corresponds to that page.
;;; Preconditions:
;;;   The string must be nonempty.
;;; Postconditions:
;;;   The returned page is valid HTML.
(define greeting-page
  (lambda (name)
    (make-page
     (head "Greetings")
     (body (string-append
            (heading 1 "Welcome")
            (string #\newline)
            (paragraph (string-append "Hi " name " !"))))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Interface

;;; Value:
;;;   user
;;; Purpose:
;;;   Names the user.  Intended as input for greeting-page.
(define user
  (if testing "Sam"
    (extract-binding/single 'user (get-bindings))))

;;; Procedure:
;;;   page
;;; Parameters:
;;;   None
;;; Purpose:
;;;   Builds an HTML page according to specifications.
;;; Produces:
;;;   A string that corresponds to the HTML page.
;;; Preconditions:
;;;   Variable "user" must be defined.
;;;   greeting-page must be definedk, take one parameter, and generate a page.
;;; Postconditions:

```

```
;;; The returned page is as valid as anything greeting-page produces.
(define page
  (lambda ()
    (greeting-page user)))
```

greeting.cgi:

```
#!/bin/bash
/home/rebelsky/bin/schemeweb greeting.ss
```

greeting.html

```
<html>
<head>
<title>Testing Scheme CGI</title>
</head>
<body>
<form method="get" action="greeting.cgi">
<input type="text" name="user" value=""> <br>
<input type="submit" value="Enter Your Name and Click Me!">
</form>
</body>
</html>
```