CSE399: Advanced Programming

Handout 16

Web Scripting in Haskell

CGI Scripting

CGI Scripts are a popular means of providing dynamic functionality for web sites.

- Web server (e.g. Apache) recognizes certain “magic URLs” as dynamically generated. E.g., `http://fling-l.seas.upenn.edu/~bcpierce/cgi-bin/ex1.cgi`
- When one of these URLs is requested, the server runs the corresponding program (`ex1.cgi`) as an external process.

Demo: ex1

CGI Scripting

- If the URL being requested comes from a `FORM` in an HTML page, the values in the form are also passed to the CGI script.
  - The details of how this happens depend on whether the form uses the `GET` or `POST` method.
    - If `GET`, then the parameters are passed in an enviroment variable called `QUERY_STRING`.
    - If `POST`, then the parameters are sent to the CGI program on `stdin`.

    In either case, a bunch of other information is passed in environment variables.
- The server takes whatever this program prints on its stdout (generally an HTML page) and sends it back to the requesting client.

CGI Scripting on SEAS Machines

- Because badly written CGI scripts can open security holes, CETS does not allow CGI scripts to be run on the regular SEAS web server.
- Instead, a special machine, `fling-lseas`, is provided for this purpose.
- `fling-l` runs the same version of linux as the lab machines
- However, it runs the same version of linux as the lab machines and has access to the same filesystem, so you can compile things using GHC on `minus` or wherever and put the binary in your `~/html/cgi-bin` directory
- Caveat: make sure your executable program has the extension `.cgi` — otherwise the server won't recognize it.

WASH/CGI

CGI Scripting in Haskell

Most languages these days have libraries that handle low-level details like parsing the information from forms.

However, writing CGI scripts that present complex functionality to the user is a harder problem. In particular, the "one-shot" request/response model provided by the HTTP and CGI protocols doesn't directly support extended conversations between a browser and a server.

For this, we need a higher-level notion of sessions.

WASH/CGI

WASH is a collection of Haskell libraries that provides such a session abstraction (along with numerous other goodies).

Demo: `Adventure.cgi` (via browser and text)

```
bcpierce@minus:~/html/cgi-bin> ./Adventure.cgi
Content-Type: text/html; charset=utf-8

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-trans
><!-- generated by WASH/HTML 0.11
--><html xmlns="http://www.w3.org/1999/xhtml"><head><title>Adventures in Haskell...</title
></head
><body><h1>Adventures in Haskell...</h1
><script type="text/javascript"><!--
var SubmitAction=[];function OnSubmit(){var r=true;for(var i=0;i<SubmitAction.length;i++){r=r&&SubmitAction
//
--></script
><form enctype="application/x-www-form-urlencoded" target="_self" onsubmit="return OnSubmit();" method="pos
/><img name="if0x0" align="center" title="non empty string expected" alt="non empty string expected" src="?
/><input onclick="this.form.WASHsub.value=this.name; return true" name="s0x1" value="Go" type="submit"
/><p>Shortcuts: <input onclick="this.form.WASHsub.value=this.name; return true" name="s0x2" value="n" type=
/>   <input onclick="this.form.WASHsub.value=this.name; return true" name="s0x3" value="s" type="submit"
/>   <input onclick="this.form.WASHsub.value=this.name; return true" name="s0x4" value="e" type="submit"
/>   <input onclick="this.form.WASHsub.value=this.name; return true" name="s0x5" value="w" type="submit"
/></p
><p style="color: red; background: #bbbbbb; "></p
><hr width="95%"
/><p>Welcome... </p
><p>Enter a command (such as n, s, e, or w) to move to the next node.</p
><input value="" name="WASHsub" type="hidden"
/><input value="W10=" name="=CGI=parm=" type="hidden"
/></form
><script type="text/javascript"><!--
document.forms[0].f0x0.focus();
document.forms[0].f0x0.select();
// --></script
></body
></html
>
bcpierce@minus:~/html/cgi-bin>
```

The Document Sublanguage

Composing Documents

Wash provides functions corresponding to all the HTML tags. So we can write, for example,

```
html (body (p (ul (li (text "Hello world")))))
```

Composing Documents

Each html constructor yields a (singleton) list of html nodes, and these lists can be concatenated using the sequencing combinator `>>`.

```
ul (li (text "a") >> li (text "b") >> li (text "c"))
```

Also, these sequences of HTML nodes are an instance of the `Monad` class, so we can use the `do` syntax for composing documents.

```
do p (text "This is a very")
   p (text "complicated way")
   p (do text "of saying"
         ul (do li (text "nothing")
                li (text "very")
                li (text "important")))
```

Composing Documents

This makes it easy to write parameterized documents:

```
standardPage ttl nodes =
  html (do head (title (text ttl))
           body (do h1 (text ttl)
                    nodes))
```

HTML Quasi-Validation

Wash includes some sophisticated (and interesting) trickery using Haskell's type classes to perform "quasi-validation" of generated HTML.

We'll return to this on Wednesday. For now, just think of

```
    WithHTML x y m a
```

as "the type of `HTML`".

The Session Language

Sessions

The ask and tell functions are used to send responses to the client browser.

```
ask :: WithHTML x CGI a -> CGI ()
tell :: (CGIOutput a) => a -> CGI ()
```

tell is lower-level: it just takes soem content (like HTML) and ships it out. It is seldom used.

ask is implemented in terms of tell. It takes some HTML with embedded forms, makes the necessary arrangements (filling in hidden fields, etc.) for restarting the session at the right place when these forms are activated, and uses tell to ship out the resulting HTML.

Whole Programs

To actually execute a CGI action, we need to turn it into an IO action. This is accomplished by the run function.

```
run :: CGI () -> IO ()
```

There is also a function

```
io :: (Read a, Show a) => IO a -> CGI a
```

that embeds an IO action in a CGI action. The result of the IO must be Readable and Showable, so that it can be recorded in the session log.

The Widget Sublanguage

Wash-Style Input

Input forms play a critical role in many interactive web sites. Wash provides very powerful (but initially somewhat tricky and puzzling!) facilities for dealing with forms in a high-level way.

HTML Forms

```
<form method=POST action="http://www.kumquat.com/demo">
  Name:
     <input type=text name=name size=32 maxlength=80>
  <p>
  Sex:
     <input type=radio name=sex value="M"> Male
     <input type=radio name=sex value="F"> Female
  <p>
  <input type=submit>
</form>
```

WASH Forms

Wraps an HTML form around its arguments. All standard attributes are computed and need not be supplied explicitly.

```
makeForm :: WithHTML x CGI a -> WithHTML y CGI ()
```

Convenient workhorse. Takes the title of a page and a monadic HTML value for the contents of the page. Wraps the contents in a form so that input fields and buttons may be used inside.

```
standardQuery :: String -> H.WithHTML x CGI a -> CGI ()
standardQuery ttl elems =
  ask (standardPage ttl (makeForm elems))
```

Continuation Buttons

Standard "wrapper type" for all kinds of input elements. The argument conveys the attributes that determine precisely how this INPUT behaves.

```
type HTMLField x y a = WithHTML x CGI () -> WithHTML y CGI
```

For example, `submit0` creates a continuation button that takes no parameters.

```
submit0 :: CGI () -> HTMLField x y ()
```