University of Arizona, Department of Computer Science

CSc 372 — Assignment 8 — Due 08:00, Tue Dec 6 — 5%

Christian Collberg

November 16, 2011

1 Introduction

In this assignment you will write your own Facebook/Orkut/Myspace/Friendster-clone, albeit a very minimalistic one. Your system will run on the department's cgi server `http://cgi.cs.arizona.edu` and will allow anyone to create a profile, edit it, browse others' profiles, and add and delete friends.

- You are encouraged to work in pairs on this assignment.
- You should hand in your code as usual using `d2l` but we will grade this assignment interactively.
- **Unlike previous assignments, 08:00, Tue Dec 6 is a hard deadline. There will be *no* extensions.**
- The assignment will be graded Tue, Dec 6 and Wed, Dec 7. We will pass around a signup sheet in class where you can make an appointment with the graders to show your system.

2 CGI Programming

When you put a program (it can be written in C, Perl, Ruby, or whatever) in `/cs/cgi/people/YOURLOGIN/public_html/` you make it possible for anyone to connect to it and run it with a browser. There should already be a script `clock.cgi` in your directory: try to run it by connecting to `http://cgi.cs.arizona.edu/~YOURLOGIN/clock.cgi`!

Programs which can be run like this are called *cgi-scripts*. We will, of course, write ours in Ruby. Ruby has a cgi-library that helps with (some of) the programming. Read about it in the text book, Chapter 18. The cgi library is documented at `http://www.ruby-doc.org/stdlib/libdoc/cgi/rdoc/index.html`.

Conceptually, cgi programming is very easy. When you click on a `SUBMIT` button on a web page the script is started up, any data that you have entered on the web site (name, password, credit card number, etc.) is transfered in the form of *name=value* pairs to the script, the script does whatever processing it needs, and then prints HTML on standard output which is then transfered back to the browser for display.

Cgi scripts are *state-less*, i.e. they don't have any memory of what has happened previously. Every time you hit `SUBMIT` the script is started up from the beginning. Any state information you need will have to be stored in cookies, databases, etc.

Every web page has to be designed by writing HTML code. Special tags are used to create a *form* where you can fill in data to be transfered to the cgi script. The `<form>`-tag begins and ends a form:

```
<form target="web.cgi">
   username: <input type="text" name="username"><br>
   password: <input type="password" name="password"><br>
   <input type="submit" name="action" value="login">
   <input type="hidden" name="username" value="">
</form>
```

You use the `input`-tag to get a box where you can type text:

```
<input type="text" name="username">
```

When the Ruby script gets invoked the text that the user entered can be extracted using

```
$cgi =  CGI.new("html3")
username = $cgi['username']
```

I.e. the variable `$cgi` is a hashtable that holds all the data that the user has entered. This HTML creates a box where we can enter a password:

```
 <input type="password" name="password">
```

and this creates a button **login** :

```
<input type="submit" name="action" value="login">
```

When the user clicks on the **login** button, the script will start up and the `action` variable will hold the string `"login"`:

```
$cgi =  CGI.new("html3")
action = $cgi['action']
if action == "login" then
    ...
```

Radio buttons are created like this:

```
  sex: <input type="radio" name="sex" value="Male"> Male
       <input type="radio" name="sex" value="Female"> Female
```

We can get the value (either `Male` or `Female`) like this:

```
$cgi =  CGI.new("html3")
sex = $cgi['sex']
if sex == "Male" then
    ...
```

The final form element that we will be using is the *pull-down-menu*. It's created using the `select` and `option` tags:

```
<select name="nopants-how">
   <option value="nohow">How did you meet?</option>
   <option value="dated">dated</option>
   <option value="random">met randomly</option>
   <option value="friend">met through a friend</option>
   <option value="family" selected>in my family</option>
   <option value="group">in a group together</option>
   <option value="work">worked together</option>
</select>
```

In the same way as previously, we can get the data that the user entered (`"random"` if the user chose `"met randomly"`, etc.) through the `$cgi` variable:

```
$cgi =  CGI.new("html3")
how = $cgi['nopants-how']
if how == "random" then
    ...
```

Adding `selected` to one of the `option`-tags makes that the initial value.

The `hidden` tag is special — it doesn't actually produce anything visible on the web page, it is simply used to pass information from one invocation of the script to the next:

```
<input type="hidden" name="username" value="redhat">
```

In our implementation we use `hidden` to keep track of which user is logged in.

2.1 A Simple Example

Consider the Ruby cgi script in Figure 1. The first time it is run the user hasn't entered any data, so the `left` and `right` fields are empty. The generated HTML looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><BODY>
<center><h1>Welcome to the calculator!</h1></center><br>
   <form target="calc.cgi">
      <center>
         <input type="text" name="left" value="">
         <select name="op">
            <option value="add" selected>+</option>
            <option value="mul" >*</option>
         </select>
         <input type="text" name="right" value="">
         =
         <br>
         <input type="submit" name="action" value="calculate">
      </center>
   </form>
</BODY></HTML>
```

The user then enters data, and hits the **calculate** button. The screen looks like this:

This time, the `left` and `right` cgi variables have values so we go ahead and compute the result and generate new HTML, this time with `left`, `right`, `op` and `value` being filled in:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><BODY>
<center><h1>Welcome to the calculator!</h1></center><br>
  <form target="calc.cgi">
    <center>
      <input type="text" name="left" value="6">
      <select name="op">
        <option value="add" >+</option>
        <option value="mul" selected>*</option>
      </select>
      <input type="text" name="right" value="7">
      =
      42
      <br>
      <input type="submit" name="action" value="calculate">
    </center>
  </form>
</BODY></HTML>
```

The headers (`DOCTYPE`, etc.) are automatically generated by the `$cgi.out` method.

2.2 The department cgi server

`calc.cgi` is available on the assignment web page. You can upload it to your area on the cgi server and try to run it from a browser. Be careful to set the ownership, permissions, and group to the correct values, though, or the cgi server will refuse to run the script! This is what I have to do (replace `collberg` with your own unix login):

```
cp calc.cgi /cs/cgi/people/collberg/public_html
chgrp wmark /cs/cgi/people/collberg/public_html
chgrp wmark /cs/cgi/people/collberg/public_html/calc.cgi
chmod u=rwx,g=xr,o=x /cs/cgi/people/collberg/public_html
chmod u=rwx,g=,o= /cs/cgi/people/collberg/public_html/calc.cgi
```

Similarly, this is what I do to set permissions for the facebook server:

```
#!/usr/bin/ruby

require 'cgi'

$cgi =  CGI.new("html3")

def calcForm(left, op, right, value)
   addselected = (op=="add")?"selected":""
   mulselected = (op=="mul")?"selected":""
   return <<-END
   <center><h1>Welcome to the calculator!</h1></center><br>
   <form target="calc.cgi">
      <center>
         <input type="text" name="left" value="#{left}">
         <select name="op">
            <option value="add" #{addselected}>+</option>
            <option value="mul" #{mulselected}>*</option>
         </select>
         <input type="text" name="right" value="#{right}">
         =
         #{value}
         <br>
         <input type="submit" name="action" value="calculate">
      </center>
   </form>
END
end

if $cgi.has_key?('left') &&  $cgi.has_key?('right') then
   left  = $cgi['left'].to_i()
   right = $cgi['right'].to_i()
   op    = $cgi['op']
   if op == "add" then
     value = (left + right).to_s
   else
     value = (left * right).to_s
   end
else
   left  = ""
   right = ""
   op    = "add"
   value = ""
end

$cgi.out {
  $cgi.html {
    $cgi.body {
         calcForm(left,op,right,value)
    }
  }
}
```

Figure 1: A simple web calculator cgi script, written in Ruby.

```
cp web.cgi /cs/cgi/people/collberg/public_html
chmod u=rwx,g=xr,o=x /cs/cgi/people/collberg/public_html
chmod u=rwx,g=,o= /cs/cgi/people/collberg/public_html/web.cgi
chgrp wmark /cs/cgi/people/collberg/public_html
chgrp wmark /cs/cgi/people/collberg/public_html/web.cgi
cp database.txt /cs/cgi/people/collberg/
chmod a+rwx /cs/cgi/people/collberg//database.txt
```

Your group should be whatever the unix `groups` command returns *first* — for me it's `wmark`, for you probably your login name.

- **For this assignment, be particularly careful with your code when you store it on the cgi server — always set permissions such that no one else can read the script!**
- **First of all, add the command `umask 077` to your `.login` file.**
- **Secondly, set the group and permissions as described above.**

The makefile is set up to take care of this. Ideally, you should be able to just run

```
> make install
> make installcalc
```

and everything will just work! You can read more about the department's cgi server and how to run scripts on it here: `http://www.cs.arizona.edu/computing/web/cgi.html`

3 Overview

Have a look at Figure 2. No, wait… 

…it's really not that bad! The figure shows the five different pages that our system can display, and how we can go from one to the other.

When we want to first access the 372book website we point our browser to

`http://cgi.cs.arizona.edu/~YOURLOGIN/web.cgi`

and we're presented with the *welcome*-page. It has two buttons that will either take us to the *login*-page or the *create new account*-page.

From the *login*-page we can go to the *profile*-page or, if the user enters an invalid username or password, back to the *login*-page. The *create new account*-page functions similarly: we keep returning to this page until the user enters a valid username (one that isn't already in the database), password, etc.

The *profile*-page displays all the information about the logged in user. We can see his/her real name, sex, a list of friends, and how many friends they have, directly or through one or two levels of indirection. There is

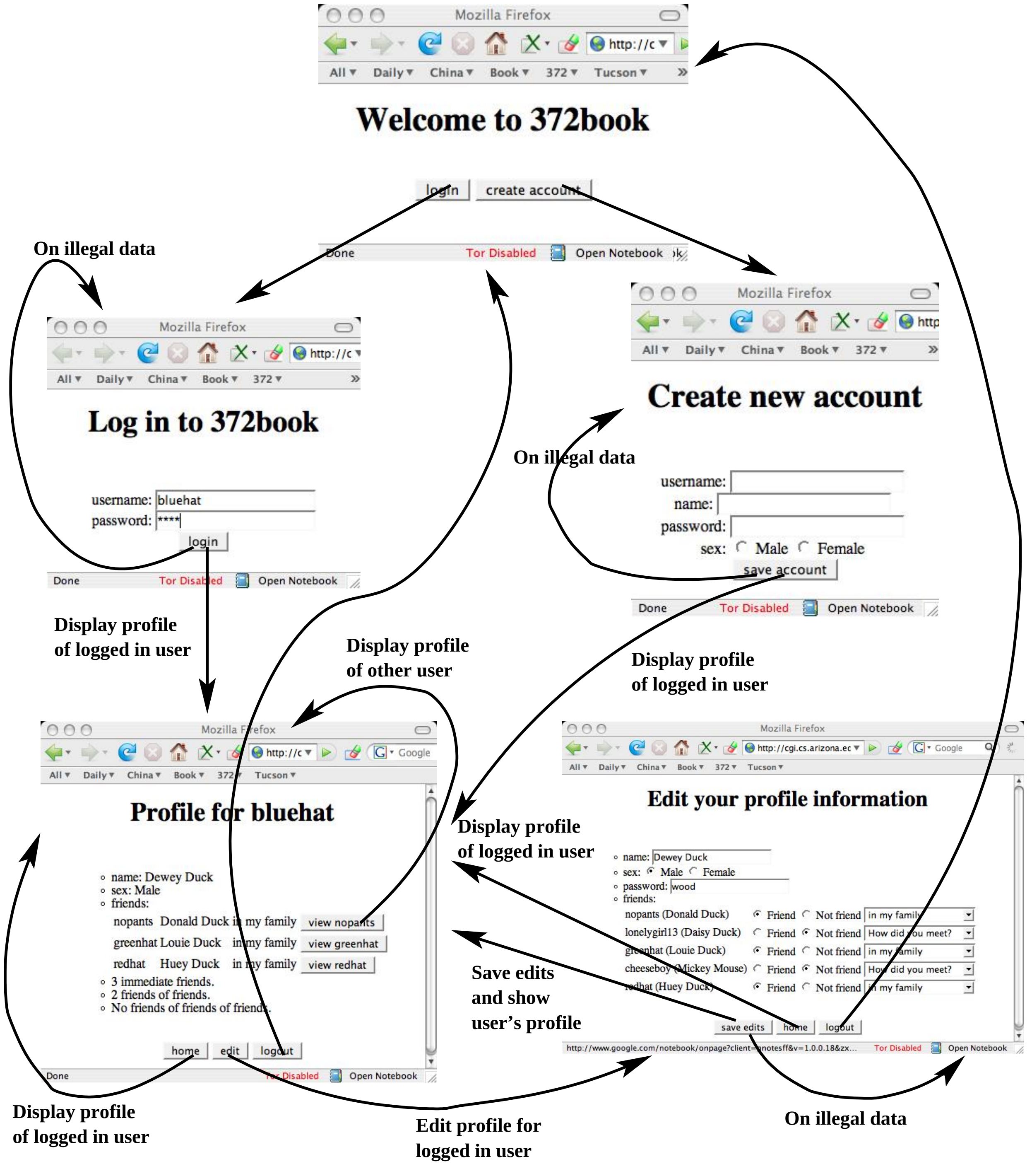


Figure 2: Overview.

one button for each friend, and clicking on it will take us back to the *profile*-page, *but* we will be displaying one of the friends instead of the logged in user. Note that clicking on a friend button doesn't change the logged in user. The **home** -button also takes us back to the *profile*-page, but this time to the logged in user's page. The **logout** -button takes us back to the login page.

Clicking on the **edit** -button takes us to the *edit page*. It is similar to the *profile*-page, except that it allows all information to be edited. All the current information about the user is preselected. Clicking on the **save edits** -button saves all the data in the database file `database.txt` whereas the **home** -button cancels the edits. In either case, we're taken back to the user's *profile*-page where the current information is now displayed.

4 Getting Started

To get started, download the files `web.cgi`, `database.txt`, and `makefile` from the class web site.

Figure 3 shows the `makefile`. If you have never used a makefile before, don't worry, it's very simple — it's simply a way to store unix commands in one place so that you can use them later. Before you get started, edit the file and change the `USER` from `collberg` to your own login.

It's best to do as much debugging as possible *off-line* before you start running on the cgi server. The Ruby cgi library has an offline mode where it reads the input to the script (in the form of `name=value`-pairs) from standard input or from the command line and writes the generated HTML to standard output. I've coded a couple of test-cases into the `makefile` — try typing `make test1`, `make test2`, etc.

When you are ready to try out your code on the department cgi server, simply type `make install` and the files will be copied over. The cgi server is *very* picky about file naming (the script has to end in `.cgi` not `.rb`), ownership, group, and permission, but (hopefully) the `makefile` sets this correctly for you! Once the files are installed, you can connect to `http://cgi.cs.arizona.edu/~YOURLOGIN/web.cgi`. When something goes wrong, unfortunately, you're likely to get a cryptic and uninformative message. There is an error-log that sometimes helps: type `make errors` to see it. Keep in mind, though, that the log is global for all cgi-scripts running on the server!

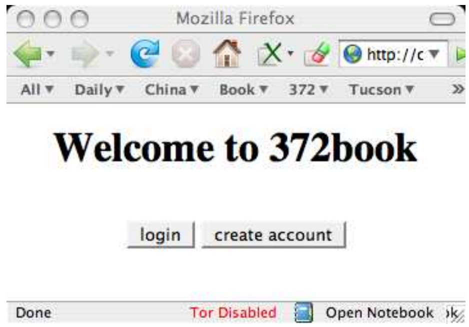
We will not grade the individual parts of your code, only the program as a whole. So, feel free to modify the template as you see fit. If you don't like my coding "style," go ahead and use your own. Add methods and classes as you wish. You like commenting your code? Go right ahead! The only thing that must remain constant is the external interface, so that we can grade your code by pointing a browser to it.

5 The Welcome Screen [10 points]

In the next few sections of this handout I will describe each of the five screens, the HTML code that should be generated for each of them, and what you need to code up. The *welcome screen* looks like this:

```
WEB      = web.cgi
CALC     = calc.cgi
DATABASE = database.txt

USER     = collberg
CGI      = /cs/cgi/people/${USER}
SCRIPTS  = ${CGI}/public_html
DATA     = ${CGI}/
GROUP    = `groups | gawk '{print $$1}'`
LOCALSITE = http://localhost/~${USER}
DIRPERMS      = u=rwx,g=xr,o=x
SCRIPTPERMS   = u=rwx,g=,o=

########################################################################
# Installing scripts on the CGI server
########################################################################

install: database.txt
cp ${WEB} ${SCRIPTS}
chmod ${DIRPERMS} ${SCRIPTS}
chmod ${SCRIPTPERMS} ${SCRIPTS}/${WEB}
chgrp ${GROUP} ${SCRIPTS}
chgrp ${GROUP} ${SCRIPTS}/${WEB}
cp database.txt ${DATA}
chmod a+rwx ${DATA}/${DATABASE}

errors:
tail /cs/cgi/logs/error_log

access:
tail /cs/cgi/logs/access_log

########################################################################
# Test scripts
########################################################################

# welcome page
test1: database.txt
ruby ${WEB} -w action=""

# OK login
test2: database.txt
ruby ${WEB} -w action=login username=nopants password=feathers

# wrong user
test3: database.txt
ruby ${WEB} -w action=login username=scrooge password=feathers
```

Figure 3: The `makefile`. The actual file contains more tests.



The HTML is also uncomplicated, just two submit buttons:

```
<!DOCTYPE HTML PUBLIC ”−//W3C//DTD HTML 3.2 Final//EN”><HTML><BODY>

   <center><h1>Welcome to 372book</h1></center><br>
   <form target=”web.cgi”>
      <center>
         <input type=”submit” name=”action” value=”login”>
         <input type=”submit” name=”action” value=”create account”>
         <input type=”hidden” name=”username” value=””>
      </center>
   </form>

</BODY></HTML>
```

You actually don't have to do any coding to get this to run, the code has already been provided for you in the `web.cgi` template you can pick up from the assignment web site. The code looks like this:

```
def Facebook.welcomeForm()
   ...
end

def Facebook.welcomeScreen()
   writeHTML(welcomeForm())
end
```

The `writeHTML()` method tells the server to transfer the generated HTML to the user's browser.

At the very bottom of `web.cgi` is the code that gets executed every time the user hits a **submit** -button:

```
action = $cgi["action"]
$username = $cgi['username']
case action
   when ""              then Facebook.welcomeScreen()
   when "login"         then Facebook.loginScreen()
   ...
end
```

We look up *which* submit button the user pressed and invoke the appropriate method.

6 The database

It's time now to talk about the database itself! For simplicity, we're just using a plain text file, `database.txt`, which you can see in Figure 4. The database file is manipulated by the two classes `User` (which allows us to create a user object storing all information about a single user) and `Database` which stores a hashtable of all the known users:

```
class User
   attr_reader :username, :name, :friends, :sex, :password
   attr_writer :username, :name, :friends, :sex, :password
   def initialize(username,name,friends,sex,password)
   def to_out
end

class Database
   def initialize
   def [] (name)
   def []= (name,user)
   def users()
   def exists_user?(name)
   def load()
   def save()
end
```

`Database` has methods for loading the text file and saving it again. The `users()` method yields all the users in the database. The `to_out()` method returns a user's data as a string, in the format of Figure 4.

7 The Login Screen [10 points]

The *login*-screen looks like this:

```
USERNAME  bluehat
NAME      "Dewey Duck"
SEX       Male
PASSWORD  wood
FRIENDS   nopants/family greenhat/family redhat/family
END

##############################################
USERNAME  lonelygirl13
NAME      "Daisy Duck"
SEX       Female
PASSWORD  sailorboy
FRIENDS   nopants/dated cheeseboy/random
END

##############################################
USERNAME  nopants
NAME      "Donald Duck"
SEX       Male
PASSWORD  feathers
FRIENDS   bluehat/family lonelygirl13/dated greenhat/family cheeseboy/friend redhat/family
END

##############################################
USERNAME  greenhat
NAME      "Louie Duck"
SEX       Male
PASSWORD  chuck
FRIENDS   bluehat/family nopants/family redhat/family
END

##############################################
USERNAME  redhat
NAME      "Huey Duck"
SEX       Male
PASSWORD  junior
FRIENDS   bluehat/family nopants/family greenhat/family
END

##############################################
USERNAME  cheeseboy
NAME      "Mickey Mouse"
SEX       Male
PASSWORD  cheddar
FRIENDS   lonelygirl13/random nopants/friend
END

##############################################
```

Figure 4: The file `database.txt`.

and the generated HTML like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><BODY>

   <center>
       <h1>Log in to 372book</h1><br>
       <font color="red"></font><br>
       <form target="web.cgi">
          username: <input type="text" name="username"><br>
          password: <input type="password" name="password"><br>
          <input type="submit" name="action" value="login">
          <input type="hidden" name="username" value="">
   </form>
   </center>

</BODY></HTML>
```

We keep returning to this page until the user enters a valid username and password:

Below is the template code. You only need to supply the `Facebook.login` method, the rest has been given to you:

```
# This method returns two values, the first one is true
# if login was successful, false otherwise. The second
# argument is an error message useful when the login was
# unsuccessful. (Yes, using exceptions might be better,
# but I hate exceptions.)
# Error messages returned:
#    "cannot log in #{username}: no such user"
#    "cannot log in #{username}: wrong password"
def Facebook.login(username,password)
   # YOUR CODE HERE
end

# Return the HTML for the login screen. "message"
# is an error message (in case the user has made
# an unsuccessful login attempt). Leave it blank
# the first time.
def Facebook.loginForm(message)

def Facebook.loginScreen()
   username =  $cgi["username"]
   password =  $cgi["password"]
   if username != "" then
```

```
      Facebook.load()
      ok, message = Facebook.login(username,password)
      if ok then
         Facebook.profileScreen(username)
      else
         writeHTML(loginForm(message))
      end
   else
      writeHTML(loginForm(""))
   end
end
```

Note how we're loading the database using `Facebook.load()`. We have to do this *every time* the script executes. Why is that? Because, as we noted earlier, the scripts don't remember anything between executions.

8 The Create Account Screen [20 points]

The *create-account*-screen is very similar to the *login*-screen:

There are several error conditions we must check for:

On any error we return to the *create-account*-screen.

Here's the generated HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><BODY>

  <center>
      <h1>Create new account</h1><br>
      <font color="red"></font><br>
      <form target="web.cgi">
         username: <input type="text" name="username"> <br>
         name: <input type="text" name="realname"> <br>
         password: <input type="password" name="password"> <br>
         sex: <input type="radio" name="sex" value="Male"> Male
              <input type="radio" name="sex" value="Female"> Female <br>
       <input type="submit" name="action" value="save account">
       <input type="hidden" name="username" value="">
  </form>

</BODY></HTML>
```

Here's the code you have to write:

```
# A username consists of a letter (upper or lower case)
# followed by at least one or more letters or digits.
def Facebook.okUsername(un)

# A password consists of four or moour letters or digits.
def Facebook.okPassword(pw)

# Sex is one of Male or Female.
def Facebook.okSex(sex)

# user is an obejct of type User, containing all the
# information gathered about the new user. We return
# two values, a boolean which is true if the
# user has been successfully added to the Database
# and false otherwise. On a failed attempt, one of
# these error messages is generated:
#    "cannot create #{username}: user already exists"
#    "cannot create #{username}: illegal username"
#    "cannot create #{username}: illegal password"
#    "cannot create #{username}: choose Male or Female"
def Facebook.createUser(user)
```

```
# Return the HTML for the create user screen. "message"
# is an error message. Leave it blank the first time.
def Facebook.createForm(message)
   ...
   <input type="submit" name="action" value="save account">
   ...
end

# This is where we arrive the first time the user hits the
# create account button:
def Facebook.createScreen()

# After the user has filled in data into the account screen
# and hits "create account", we'll arrive here. We check
# for valid data and either go to the profile screen (if
# all was OK) or back to the create account screen if there
# was an error.
def Facebook.saveCreatedUser()
```

Note the `Facebook.createForm(message)`-code. The value of the `action` cgi variable is set to `save account`. This is so that we'll go to `Facebook.saveCreatedUser()` to save the values that user has filled into the form.

9 The Profile Screen [30 points]

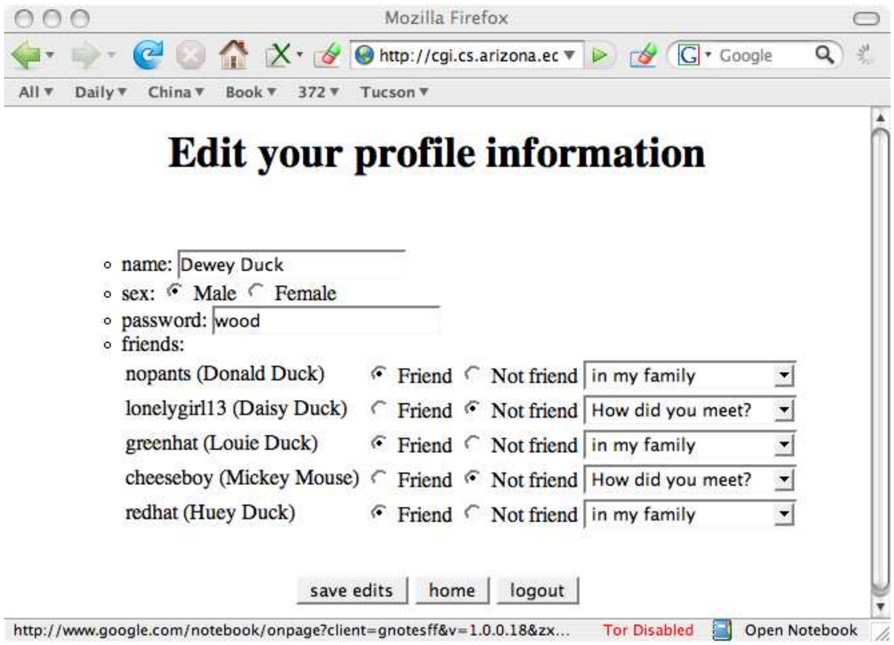
This is what the *profile*-screen looks like, followed by the corresponding HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><BODY>

  <center><h1>Profile for bluehat</h1></center><br>
  <form target="web.cgi">
  <ul>
    <ul>
      <li> name:      Dewey Duck
      <li> sex:       Male
      <li> friends:
    <table>
    <tr>
      <td> nopants </td>
      <td> Donald Duck </td>
      <td> in my family </td>
      <td> <input type="submit" name="action" value="view nopants"> </td>
    </tr>
        ..............
    </table>
      <li> 3 immediate friends.
      <li> 2 friends of friends.
      <li> No friends of friends of friends.
    </ul>
  </ul>
  <br>
  <center>
    <input type="submit" name="action" value="home">
    <input type="submit" name="action" value="edit">
    <input type="submit" name="action" value="logout">
    <input type="hidden" name="username" value="bluehat">
 </center>
  </form>

</BODY></HTML>
```

Note that I want you to be grammatical when you print out the number of friends! I.e., get the singular/plural correct!

10 The Edit Screen [30 points]

This is what the *edit*-screen looks like:



and the generated HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><HTML><BODY>

   <center><h1>Edit your profile information</h1></center><br>
   <form target="web.cgi">
   <ul>
      <ul>
         <li> name:      <input type="text" name="name" value="Dewey Duck">
         <li> sex:       <input type="radio" name="sex" value="Male" checked> Male
                         <input type="radio" name="sex" value="Female" > Female <br>
         <li> password: <input type="text" name="password" value="wood">
         <li> friends:
      <table>
      <tr>
         <td> nopants (Donald Duck)</td>
         <td> <input type="radio" name="nopants-isfriend" value="yes" checked> Friend</td>
         <td> <input type="radio" name="nopants-isfriend" value="no" > Not friend</td>
         <td>       <select name="nopants-how">
         <option value="nohow" >How did you meet?</option>
         <option value="dated" >dated</option>
         <option value="random" >met randomly</option>
         <option value="friend" >met through a friend</option>
         <option value="family" selected>in my family</option>
         <option value="group" >in a group together</option>
         <option value="work" >worked together</option>
      </select></td>
      </tr>
      <tr>
         <td> lonelygirl13 (Daisy Duck)</td>
          ..................
      </tr>
```

```
      ......................
    </table>

    </ul>
  </ul>
  <br>
  <center>
    <input type="submit" name="action" value="save edits">
    <input type="submit" name="action" value="home">
    <input type="submit" name="action" value="logout">
    <input type="hidden" name="username" value="bluehat">
  </center>

</BODY></HTML>
```

11 Submission and Assessment

The deadline for this assignment is 08:00, Tue Dec 6. It is worth 5% of your final grade.

You should submit the assignment using `d2l.arizona.edu`. The `README` file should give the members of your team.

The assignment will be graded Tue, Dec 6 and Wed, Dec 7. You should set up an appointment with the TA for a slot to show your program.

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.