Computer Science 50
Fall 2009
Scribe Notes

Week 9 Monday: November 2, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 9 Monday: November 2, 2009
Andrew Sellergren

1 Announcements (0:00–5:00, 30:00–32:00)

- This is CS 50.
- 1 new handout.
- Dinner with David (and faculty) Wednesday at 6 PM. Also Lunch with David Friday at 1:15 PM. RSVP here.
- Beware of floating point imprecision in the almighty Google's calculator. Also read about Why computers suck at maths.
- Check out The Big Board for Problem Set 7. Charles Li has jumped out to an early lead thanks to a bit of exploitation. That's fine, though, that just makes him the man to beat. The gauntlet has been thrown down! You'll also notice that The Big Board is implemented with Ajax, which allows it to update without the page needing to be refreshed. We'll talk more about Ajax in the context of Problem Set 8. You're welcome to browse the HTML and CSS source code, but you won't have access to the PHP source code for the staff solution. This is true of most websites you'll find on the internet.
- Lolcats are the greatest thing that's ever happened to the internet. Period. If you think otherwise, then you are officially banned from the VTR. Yes, I have that authority.
- Apply now to be a TF or CA for next year![1] TFs are responsible for leading sections, holding office hours, grading problem sets. CAs are generally CS 50 alumni who volunteer for two hours a week to help with answer students' questions via e-mail, bulletin board, and office hours.

2 Problem Set 6 in PHP (5:00–30:00)

- In a program named `speller`, we're going to re-implement Problem Set 6 in PHP. This might prove somewhat demoralizing, as you'll see that we can do so with considerably less hassle than we experienced while implementing it in C. The trade-off, however, is performance. Our PHP program will run much slower than our C program.
- We saw last week that PHP files can be run from the command line by invoking the PHP interpreter like so: `php <filename>`. However, they can also be executed by typing just the filename if the file itself contains at its top the path to the PHP interpreter in what's called a shebang:

  ```
  #!/usr/bin/php
  ```

  We also need to make sure that the program has executable permissions by running `chmod 700 speller`.

[1] Link will be posted on the course website shortly.

- Let's take a look at the source code for `speller`:

```
#!/usr/local/bin/php
<?php

    /**************************************************************************
     * speller.php
     *
     * Computer Science 50
     * David J. Malan
     *
     * Implements a spell-checker.
     **************************************************************************/

    require("dictionary.php");

    // suppress notices and warnings
    error_reporting(E_ALL ^ E_NOTICE ^ E_WARNING);


    // maximum length for a word
    // (e.g., pneumonoultramicroscopicsilicovolcanoconiosis)
    define("LENGTH", 45);

    // default dictionary
    define("WORDS", "/home/cs50/pub/share/pset6/dict/words");

    // check for correct number of args
    if ($argc != 2 && $argc != 3)
    {
        print("Usage: speller.php [dict] file\n");
        return 1;
    }

    // benchmarks
    $ti_load = 0.; $ti_check = 0.; $ti_size = 0.; $ti_unload = 0.;

    // determine dictionary to use
    $dict = ($argc == 3) ? $argv[1] : WORDS;

    // load dictionary
    $before = microtime(TRUE);
    $loaded = load($dict);
    $after = microtime(TRUE);

    // abort if dictionary not loaded
```

```
if (!$loaded)
{
    print("Could not load $dict.\n");
    return 2;
}

// calculate time to load dictionary
$ti_load = $after - $before;

// try to open file
$file = ($argc == 3) ? $argv[2] : $argv[1];
$fp = fopen($file, "r");
if ($fp === FALSE)
{
    print("Could not open $file.\n");
    return 3;
}

// prepare to report misspellings
printf("\nMISSPELLED WORDS\n\n");

// prepare to spell-check
$word  = "";
$index = 0; $misspellings = 0; $words = 0;

// spell-check each word in file
for ($c = fgetc($fp); $c !== FALSE; $c = fgetc($fp))
{
    // allow alphabetical characters and apostrophes (for possessives)
    if (preg_match("/[a-zA-Z]/", $c) || ($c == "'" && $index > 0))
    {
        // append character to word
        $word .= $c;
        $index++;

        // ignore alphabetical strings too long to be words
        if ($index >= LENGTH)
        {
            // consume remainder of alphabetical string
            while (($c = fgetc($fp)) !== FALSE && preg_match("/[a-zA-Z]/", $c));

            // prepare for new word
            $index = 0; $word = "";
        }
    }
```

```
        // ignore words with numbers (like MS Word)
        else if (ctype_digit($c))
        {
            // consume remainder of alphabetical string
            while (($c = fgetc($fp)) !== FALSE && preg_match("/[a-zA-z0-9]", $c));

            // prepare for new word
            $index = 0; $word = "";
        }

        // we must have found a whole word
        else if ($index > 0)
        {
            // update counter
            $words++;

            // check word's spelling
            $before = microtime(TRUE);
            $misspelled = !check($word);
            $after = microtime(TRUE);

            // update benchmark
            $ti_check += $after - $before;

            // print word if misspelled
            if ($misspelled)
            {
                print("$word\n");
                $misspellings++;
            }

            // prepare for next word
            $index = 0; $word = "";
        }
    }

    // close file
    fclose($fp);

    // determine dictionary's size
    $before = microtime(TRUE);
    $n = size();
    $after = microtime(TRUE);

    // calculate time to determine dictionary's size
    $ti_size = $after - $before;
```

```
    // unload dictionary
    $before = microtime(TRUE);
    $unloaded = unload();
    $after = microtime(TRUE);

    // abort if dictionary not unloaded
    if (!$unloaded)
    {
        print("Could not load $dict.\n");
        return 5;
    }
    // calculate time to determine dictionary's size
    $ti_unload = $after - $before;

    // report benchmarks
    printf("\nWORDS MISSPELLED:     %d\n", $misspellings);
    printf("WORDS IN DICTIONARY:  %d\n", $n);
    printf("WORDS IN FILE:        %d\n", $words);
    printf("TIME IN load:         %f\n", $ti_load);
    printf("TIME IN check:        %f\n", $ti_check);
    printf("TIME IN size:         %f\n", $ti_size);
    printf("TIME IN unload:       %f\n", $ti_unload);
    printf("TOTAL TIME:           %f\n\n", $ti_load + $ti_check + $ti_size + $ti_unload

?>
```

In the first line, we're simply including a helper file. The call to `error_reporting()` is PHP-specific—it suppresses notices and warnings, the less severe of the three types of messages, but allows errors, the most severe, to be displayed. Next we're defining constants for the maximum length of a word and the default dictionary, just as we did in C, albeit with slightly different syntax. Then, as we saw in `quote.php`, we can work with command-line arguments using the `$argc` and `$argv` variables.

- When we initialize the benchmark variables, we write `0.` so that they will be stored under the hood as floating points. Recall that because PHP is loosely typed, you the programmer don't usually need to worry about explicit casting between variables, but in this case we don't want the benchmarks to be stored as integers.
- The rest of the code implements the same benchmarking we saw in C. Walk through it yourself when you have a chance.
- But what Problem Set 6 was really about was implementing four functions: `load()`, `check()`, `size()`, and `unload()`. So let's walk through that

step by step in a file called `dictionary.php`, which will automatically be included when we run `speller`:

```
<?

    //hash table
    $dictionary = array();

    //has table's size
    $size = 0;

    function load($dict)
    {

    }

    function check($word)
    {

    }

    function size()
    {
        return $size;
    }

    function unload()
    {

    }

?>
```

Although it's possible to implement your own data structures in PHP, for the most part it's unnecessary. Arrays come built in to PHP and, in fact, are implementations of hash tables. The superglobals `$_POST` and `$_GET` are associative arrays, which means they can have not only numbers, but also strings, as indices. So to implement our dictionary data structure as a hash table, we need only initialize it as an empty array.

- The second global variable we'll need is one to keep track of the size of our hash table. We can very quickly implement the `size()` function simply by returning this variable.
- So let's start implementing `load()`:

```
<?
```

```
    function load($dict)
    {
        $lines = file($dict);
        print_r($lines);
        exit;
    }
?>
```

The `file()` function returns an array containing each word in the file. Let's test its functionality just by recursively printing out this array and then exiting. If we do this, we see that when we run `speller`, we'll get an array printed out with each numeric index corresponding to one word in the dictionary, the 143,091th being "zymurgy," the last word in the dictionary. We do notice that each element of the array is separated by two lines because the newline character is being printed along with each word. We'll deal with that in a moment.

- Instead of assigning the array to any variable, let's skip straight to iterating over its elements:

```
<?
    function load($dict)
    {
        foreach (file($dict) as $word)
        {
            $dictionary[trim($word)] = true;
        }
    }
?>
```

The `foreach` syntax is one which doesn't exist in C, but does in PHP and allows us to iterate over ever element in an array or object, assigning each to a temporary variable whose name we specify after the `as` keyword—`$word` in this case. We're first passing `$word` to the function `trim()` in order to remove whitespace from the beginning and end, since we know that there is a newline character at the end of each word in the dictionary.

- So for every word in the dictionary, we're going to create an index into our hash table which has the value true. Thus, our `check()` function will simply take a word as input and check if the value at that index is equal to true. In fact, let's go ahead and quickly implement `check()`:

```
<?
    function check($word)
    {
        if ($dictionary[$word])
            return true;
```

```
        else
            return false;
    }
?>
```

One last sanity check we'll add to `load()` is checking whether the dictionary file has been opened properly:

```
<?
    function load($dict)
    {
        if (!file_exists($dict) || !is_readable($dict))
            return false;
        foreach (file($dict) as $word)
        {
            $dictionary[trim($word)] = true;
        }
    }
?>
```

One last nuisance of PHP is that global variables can't actually be accessed within functions unless they're declared as `global` within that function. So, finally, we have:

```
<?

     //hash table
    $dictionary = array();

    //has table's size
    $size = 0;

    function check($word)
    {
        global $dictionary;
        if ($dictionary[$word])
            return true;
        else
            return false;
    }

    function load($dict)
    {
        global $dictionary;
        global $size;
        if (!file_exists($dict) || !is_readable($dict))
```

```
            return false;
        foreach (file($dict) as $word)
        {
            $dictionary[trim($word)] = true;
            $size++;
        }
    }

    function size()
    {
        global $size;
        return $size;
    }

    function unload()
    {
        return true;
    }
?>
```

Since we don't manipulate memory directly in PHP, we don't need to free any of it at the end, so our `unload()` function simply returns true. We also need to actually make sure we're incrementing `$size` every time we add a word to the hash table. And we're done!

- But when we run `speller` on `ralph.txt`, we get a count of 8 misspelled words, which isn't correct. One last change we need to make is handling capitalization. Let's just convert all words to lowercase before we check if they're in the hash table:

```
<?
    function check($word)
    {
        global $dictionary;
        if ($dictionary[strtolower($word)])
            return true;
        else
            return false;
    }
?>
```

- It took a matter of minutes to implement Problem Set 6 in PHP, but we'll pay for it in performance. If we run our PHP implementation on a longer text like `holmes.txt`, we'll find that our running time exceeds 2 seconds. For the same text, the C implementation runs in less than half a second.[2]

[2] Actually, there's also a bug in the PHP implementation that's causing the number of misspelled words to be wrong. Can you find it?

- Question: indexing into our hash table with words that don't exist in the dictionary might cause a segfault in C, but in PHP it will simply trigger a notice. Recall that we suppressed notices in `speller` with our call to `error_reporting()`. We could fix this by calling the `isset()` function to see if the index actually exists before trying to access its value.
- So what was the point of Problem Set 6? First, we wanted to show you that there are different tools for the job. Second, we wanted to make sure that you understand exactly how something like an array or hash table is implemented under the hood. Once you've done the hard work of learning C, you've really earned the privilege to program in higher-level languages like PHP. Know that programming isn't all about banging your head against the wall to fix compiler errors.

3 Web Development Tools (32:00–39:00)

- For the purposes of Problem Set 7, we'll recommend that you use Firefox for development, simply because of the wealth of tools it offers for debugging. Of course, we'll still expect that your site displays similarly in two major browsers, only one of which can be Firefox.
- Firebug is an invaluable add-on for Firefox. To begin with, it allows you to view the source of a website—whether yours or someone else's—as an expandable hierarchical tree. In contrast, often if you simply opt to View Source for a webpage, the XHTML will almost be unreadable because it is dynamically generated with no attention being paid to pretty printing.
- If we click the bug icon at the bottom right of our Firefox browser, a window will pop up on the lower half of the page. We click on the HTML tab to show the aforementioned expandable tree. As we click the plus signs next to each element to reveal its children, we can also click on an element to see its content highlighted on the webpage itself.
- You can also click on the CSS tab, select an HTML element, and view the styles that apply to it on the right half of the Firebug window.
- One other tool we'll discuss on Wednesday is JavaScript Debugger, another Firefox add-on. We'll now briefly mention the Web Developer add-on for Firefox, which allows you to easily manipulate the size of the browser window—so that you might test your site on 800x600 as well as 1024x768 resolutions.

4 JavaScript (39:00–68:00)

- So far in our foray into web development, the only functional programming language we've worked with has been PHP, which executes server-side. As we've already seen, this can present a problem when we perform validation

and then redirect the user back to a form where all his previous input has been deleted. To help deal with this scenario, we can implement client-side validation using JavaScript. In this way, we'll prevent the user's form from ever submitting if his inputs aren't valid.

- What we mean by *client-side* is that the source code will actually be downloaded and executed by the browser (the client) when a user visits the site. You can see this client-side execution in action when you visit Google Maps and click and drag the map. If you do it fast enough, you'll see that gray squares are quickly being replaced by map squares, which are being downloaded as needed. In fact, this dynamic downloading is the result of Ajax (which formerly stood for Asynchronous JavaScript and XML), which can update a site's content without the need for a page refresh. We saw this in the context of The Big Board for Problem Set 7.
- Let's take a look at a very simple HTML form, `form1.html`, that **doesn't** implement client-side validation:

```
<!--

form1.html

A form without client-side validation.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title></title>
  </head>
  <body>
    <form action="dump.php" method="get">
      Email: <input name="email" type="text" />

      <br />
      Password: <input name="password1" type="password" />
      <br />
      Password (again): <input name="password2" type="password" />
      <br />
      I agree to the terms and conditions:
```

```
      <input name="agreement" type="checkbox" />
      <br /><br />

      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

Notice that this form submits to `dump.php`, which doesn't actually do anything except spit out the contents of `$_GET`. Recall that `$_GET` is useful for non-sensitive information of shorter length (since it is passed in the URL), whereas `$_POST` is useful for sensitive information of longer length (since it is passed in the headers). `$_REQUEST` holds the contents of both `$_POST` and `$_GET`, but it's generally better design to refer to one or the other specifically, whenever possible.

- Now let's move on to `form2.html`, which implements rudimentary client-side validation:

```
<!--

form2.html

A form with client-side validation.

Computer Science 50
David J. Malan

-->

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <script type="text/javascript">
    // <![CDATA[

        function validate()
        {
            if (document.forms.registration.email.value == "")
            {
                alert("You must provide an email adddress.");
                return false;
            }
```

```
            else if (document.forms.registration.password1.value == "")
            {
                alert("You must provide a password.");
                return false;
            }
            else if (document.forms.registration.password1.value !=
                      document.forms.registration.password2.value)
            {
                alert("You must provide the same password twice.");
                return false;
            }
            else if (!document.forms.registration.agreement.checked)
            {
                alert("You must agree to our terms and conditions.");
                return false;
            }
            return true;
        }

    // ]]>
    </script>
    <title></title>
  </head>
  <body>

    <form action="dump.php" method="get" name="registration"
          onsubmit="return validate();">
      Email: <input name="email" type="text" />
      <br />
      Password: <input name="password1" type="password" />
      <br />
      Password (again): <input name="password2" type="password" />
      <br />

      I agree to the terms and conditions:
      <input name="agreement" type="checkbox" />
      <br /><br />
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

Here we see that a new attribute of the form is defined: `onsubmit`. As you might guess, it controls what happens when the form is actually submitted. In this case, we're calling a JavaScript function called `validate()`.

- At the top, in the `head` element, we have a new tag named `script`. Although there are other types of scripts that can be embedded, generally speaking this tag is only used to embed JavaScript. Right after the open tag, we have the following sequence of characters:

  ```
  // <![CDATA[
  ```

  The `<![CDATA[` tells the browser is that the lines which follow are *character data* and shouldn't be parsed as XHTML. In this way, characters like `<` which are used in the JavaScript won't prevent our webpage from validating. The double slash in front tells the JavaScript interpreter not to interpret that line as JavaScript.

- Before we begin walking through the `validate()` function, we can key in on what it might be checking. We'll want to check that the Email field isn't blank, that the Password fields match, and that the checkbox is checked.

- The first if condition in `validate()` checks for a blank Email field. We do this by walking through the DOM hierarchy beginning with the root node, `document`. From there we access the `forms` followed by the `registration` form in particular. Notice that `registration` corresponds to the value we gave to the `name` attribute of our form. Finally we access the `value` attribute of the `email` field. If this value is the empty string, then we call a built-in function called `alert()` which pops up a window. After this window pops up, we return false, which is important to ensure that the form doesn't actually submit. In our `onsubmit` tag, we are returning whatever is returned by `validate()`. If false, then the form won't be submitted, else it will.

- In the next conditions, we check for a blank Password field and for non-matching Password fields. Then we access the `checked` property of the checkbox field to make sure that it has been clicked.

- So why bother with server-side validation if client-side validation is so simple and elegant? As it turns out, users can disable JavaScript in almost every major browser with a few clicks of the mouse. If a malicious user were to do this, he could get past all your client-side validation. Thus if you have no server-side validation, he could have a field day with your form. Just as importantly, not every browser fully supports JavaScript—the BlackBerry browser being a good example. When you're developing a website, then, you need to consider what users you might be alienating if you choose to implement functionality which absolutely requires JavaScript.

- When you navigate to Facebook's home page, what actually gets sent to the server are HTTP headers which might look something like this:

```
GET /home.php HTTP/1.1
```

If we send some variables as well, say, to Google, the headers might look like so:

```
GET /search?q=foo HTTP/1.1
```

We can see the same, albeit slightly more complicated, using the Live HTTP Headers add-on for Firefox when we submit our own form. What this shows us is that even if JavaScript is enabled and client-side validation is in place, we can circumvent it and submit the form anyway simply by sending properly formed HTTP headers.

- We can boil down web browsing to a simple exchange of HTTP headers using the `telnet` command-line program. If we execute

  ```
  telnet www.facebook.com 80
  ```

  we can connect to Facebook's servers on port 80. Then we only need to run `GET /home.php HTTP/1.1` to mimic a browser's request for the homepage. In response, we'll get a HTTP 302 message, so we can follow this as well and make a request for a different page. In response, we'll get a whole slew of HTML with embedded JavaScript. The JavaScript will actually be *obfuscated*, meaning it is purposefully made difficult to read to discourage intellectual property from being stolen. Obfuscation usually entails removing whitespace and shortening variable and function names.

- One last JavaScript trick we'll introduce is in `form4.html`:

  ```
  <!--

  form4.html

  A form with client-side validation demonstrating disabled property.

  Computer Science 50
  David J. Malan

  -->

  <!DOCTYPE html PUBLIC
       "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <script type="text/javascript">
  ```

```
    // <![CDATA[

        function toggle()
        {
            if (document.forms.registration.button.disabled)
                document.forms.registration.button.disabled = false;
            else
                document.forms.registration.button.disabled = true;
        }

        function validate()
        {
            if (document.forms.registration.email.value == "")
            {
                alert("You must provide an email adddress.");
                return false;
            }
            else if (document.forms.registration.password1.value == "")
            {
                alert("You must provide a password.");
                return false;
            }
            else if (document.forms.registration.password1.value !=
                    document.forms.registration.password2.value)
            {
                alert("You must provide the same password twice.");
                return false;
            }
            else if (!document.forms.registration.agreement.checked)
            {
                alert("You must agree to our terms and conditions.");
                return false;
            }
            return true;
        }

    // ]]>
    </script>
    <title></title>
</head>
<body>

    <form action="dump.php" method="get" name="registration"
    onsubmit="return validate();">
      Email: <input name="email" type="text" />
      <br />
```

```
      Password: <input name="password1" type="password" />
      <br />
      Password (again): <input name="password2" type="password" />
      <br />

      I agree to the terms and conditions:
      <input name="agreement" onclick="toggle();" type="checkbox" />
      <br /><br />
      <input disabled="disabled" name="button" type="submit" value="Submit" />
    </form>
  </body>
</html>
```

This form has the functionality of disabling the Submit button so long as the user hasn't checked the terms and conditions checkbox. To accomplish this, we implement the `toggle()` function, which is called whenever the checkbox is clicked thanks to its `onclick` attribute. `toggle()` then checks if the Submit button is disabled (via the `disabled` attribute), in which case it enables it (by setting this attribute to false).

- As a teaser for Wednesday's discussion of Ajax, check out `ajax2.html`, which retrieves a stock quote without a page refresh.