CSC 372, Spring 2015
Assignment 2 Supplement—Using the tester

The syllabus says,

> *For programming problems great emphasis will be placed on the ability to deliver code whose output exactly matches the specification. Failing to achieve that will typically result in large point deductions, sometimes the full value of the problem.*

Whenever possible I'll use an automated testing tool to test your solutions for programming problems. For each assignment I'll make available a "student set" of tests that you can run yourself. The set of tests used when grading (the "grading set") will often have additional tests. Passing all the tests in the student set will guarantee at least 75% of the points for a given problem, but that percentage will typically be higher, around 80-90%. It some cases it will be 100%, with the student set used as the grading set.

Test cases in the grading set are weighted. Those weights are not supplied in the student set but the rule of thumb is that common cases are weighted more than unusual cases.

Sometimes I'll give you a break for bonehead mistakes but there's no excuse for not using the tester. If a student says, "I spent many hours on this and it works great but it failed every test because in one place I used a comma instead of a semicolon."; I'll ask, "Did you use the tester?"

The tester is on lectura

The tester is a collection of bash scripts, Ruby programs, and assorted files that are all on lectura. All the files are world-readable and the whole works could be hauled over to your own machine and run there, no matter whether it's a Mac, a Linux machine, or a Windows machine with Cygwin installed. However, there's a lot to the tester and it's poorly documented. There are also potential issues with staying in sync with the per-assignment configuration files and test cases.

You're welcome to try getting the tester machinery working on your own machine but I can't promise that I'll have much time to help you with it.

The instructions in this document assume that you're working on lectura.

Use a symbolic link for easy access to the tester and data files

An easy way to access data files and the tester itself is to put a symlink (symbolic link) to `/cs/www/classes/cs372/spring15/a2` in your assignment 2 directory. I describe a symbolic link as a Windows shortcut done right. Imagining that you've got your Haskell files for assignment 2 in an `assn2` directory that's a subdirectory of `372`, which is in your home directory on lectura. Here's what to do:

```
$ cd ~/372/assn2
$ ln -s /cs/www/classes/cs372/spring15/a2 .
```

Then take a look at what `ls` shows:

```
$ ls -l a2
lrwxrwxrwx 1 whm whm 33 Feb  1 13:50 a2 -> /cs/www/classes/cs372/spring15/a2
```

That lowercase "L" at the start of the line and the `->` after `a2` indicate a symbolic link. If you "`ls a2`", "`cd a2`", etc., you'll actually be operating on `/cs/www/classes/cs372/spring15/a2`.

Using "`.`" as the last argument for `ln` causes the link to have the same name as the target (`a2`) but you could name it something else. Omitting the last argument has the same effect.

Quick summary of usage, for the TLDR crowd

With the `a2` symlink in place and your solution for `warmup.hs` in the current directory, you can test it like this:

```
$ a2/tester warmup.hs
```

To stop it early, use `^C` (control+C). It might take more than one `^C`, too.

You'll probably want to test problems one at a time, as you develop them, but to test all problems in sequence, you can run the tester with no arguments. Here we combine that with `grep` to simply look for failures:

```
$ a2/tester | grep FAIL
```

Be sure to capitalize `FAIL`, or use `grep -i fail`, to ignore case.

You can also name multiple problems to be tested:

```
 $ a2/tester join cpfx warmup
```

The above case also shows that the `.hs` suffix is not required on the file names.

More detail on running the tester

Let's work with a simple "hello" function:

```
    > :type hello
    hello :: [Char] -> [Char]

    > hello "whm"
    "Hello, whm!"
```

The code is in the file `hello.hs`:

```
$ cat hello.hs
hello s = "Hello, " ++ s ++ "!"
```

Here's a test run with no failures:

```
$ a2/tester hello.hs

------------------------------------------------------------
|                                                          |
|                        .: hello                          |
|                                                          |
------------------------------------------------------------

------------------------------------------------------------
|                                                          |
|                     Test Execution                       |
|                                                          |
------------------------------------------------------------

Test: 'ulimit -t 2; ./tesths hello.hs ':type hello'': PASSED

Test: 'ulimit -t 2; ./tesths hello.hs 'hello "world"'': PASSED
```

Those two lines starting with "`Test: `" indicate that two tests were run. Both passed. I've underlined the output that shows what's actually being tested. The first test, `:type hello`, checks the type of the function. The second test runs the function, with `hello "world"`.

The `Test:` lines start with `ulimit -t 2;`. That limits CPU time for the test for two seconds, which should be plenty for these two. The text that follows that semicolon, up to the final apostrophe, is the exact command that was run. You can do a copy/paste to run it yourself. Let's try them:

```
$ ./tesths hello.hs ':type hello'
*Main> "TESTING START"
*Main> hello :: [Char] -> [Char]
*Main> Leaving GHCi.

$ ./tesths hello.hs 'hello "world"'
*Main> "TESTING START"
*Main> "Hello world!"
*Main> Leaving GHCi.
```

The portion `./tesths` causes a bash script named `tesths` in the current directory to be run.
The balance of the line is arguments for the script that specify the source file and the input to send to `ghci`.

Note that the script `tesths` was copied into the current directory as a side-effect of running `a2/tester`. If there was already a file named `tesths`, it would have been overwritten! Another side-effect is that the directory `tester.out` was created, to hold various files created by the testing process.

Understanding differences reported by the tester

If a test fails, the `diff` command is to used to show you the differences found between the expected output and the actual output. "diffs" can sometimes be hard to understand. Googling for "understanding diffs" or "deciphering diffs" turns up a lot of stuff, but here are a couple of examples, too.

Let's intentionally break `hello` by removing the comma in `"Hello, "`. Here's a failure, with line numbers added for reference. Line 5 is long and has wrapped around.

```
$ a2/tester hello.hs
[...header lines not shown...]

1.  Test: 'ulimit -t 2; ./tesths hello.hs ':type hello'': PASSED
2.
3.  Test: 'ulimit -t 2; ./tesths hello.hs 'hello "world"'': FAILED
4.  Differences (expected/actual):
5.  *** /cs/www/classes/cs372/spring15/a2/master/tester.out/
    hello.out.1     2015-02-01 14:01:29.589017038
6.  --- tester.out/hello.out.1      2015-02-01 15:40:11.491556742
7.  ***************
8.  *** 1,3 ****
9.    *Main> "TESTING START"
10. ! *Main> "Hello, world!"
11.   *Main> Leaving GHCi.
12. --- 1,3 ----
13.   *Main> "TESTING START"
14. ! *Main> "Hello world!"
15.   *Main> Leaving GHCi.
```

The type of `hello` is unchanged but the output differs, so the first test still passes but the second now fails.

Lines 5 and 6 name the two files that are being "diffed" (compared). The first is the file that contains the expected output, `.../a2/master/tester.out/hello.out.1`. The second, `tester.out/hello.out.1`, contains the output produced by running `./tesths hello.hs 'hello "world"'` in the current directory. You can look at both files with `cat`, `more`, editors, or any other tool. Those file names are preceded by `***` and `---` , which are used later, on lines 8 and 12, to identify the files those blocks of text came from.

Line 8's "`*** 1,3 ****`" means that what follows are lines 1-3 from the expected output. Line 12's "`--- 1,3 ----`" means that what follows are lines 1-3 from the actual output. Diffs in tester output always follow the convention of showing the expected output first and the actual output second.

The exclamation marks on lines 10 and 14 indicate that those lines differ between the expected and actual output. If we didn't already know what we did to break it, we might need to look close to see that the lines differ by only a single comma.

For a more interesting "diff", lets work with a function that prints the first N lower-case letters, one per line. Like the `street` problem, this function directly produces printed output using `putStr` rather than producing a value that is in turn displayed by `ghci`. Here's an example of expected behavior.

```
> letters 4
a
b
c
d
```

Here's what we get with our buggy version, with line numbers added to aid explanation:

```
$ a2/tester letters.hs
[...header lines not shown...]

1.  Test: 'ulimit -t 2; ./tesths letters.hs 'letters 4'': FAILED
2.  Differences (expected/actual):
3.  *** /cs/www/classes/cs372/spring15/a2/master
    /tester.out/letters.out.0   2015-02-01 15:27:28.521369342
4.  --- tester.out/letters.out.0    2015-02-01 15:34:02.502277020
5.  ***************
6.  *** 1,6 ****
7.    *Main> "TESTING START"
8.  ! *Main> a
9.    b
10. - c
11.   d
12.   *Main> Leaving GHCi.
13. --- 1,8 ----
14.   *Main> "TESTING START"
15. ! *Main>
16. ! a
17. ! x
18.   b
19.   d
20. + Done!
21.   *Main> Leaving GHCi.
```

We see in line 1 that `letters 4` was the Haskell expression that was tested.

Lines 3 and 4 identify the two files being diffed. Blocks from the expected output will be identified with `***`; blocks from the actual output will be identified with `---`.

The exclamation marks on line 8 and lines 15-17 show that those sections apparently correspond to each other but their content differs.

The minus sign on line 10 shows that there's a line in the expected output, with just a "`c`", that doesn't seem to appear in the actual output.

The plus sign on line 20 shows that there's a line in the actual output, "`Done!`", that doesn't seem to appear in the expected output.

If we have trouble understanding a diff we might first look at the expected output. We can take advantage of the `a2` symlink that we created early on to type or copy/paste less:

```
$ cat a2/master/tester.out/letters.out.0
*Main> "TESTING START"
*Main> a
b
c
d
*Main> Leaving GHCi.
```

Here's the actual output. Remember that the tester creates a `tester.out` directory in the directory it's run in.

```
$ cat tester.out/letters.out.0
*Main> "TESTING START"
*Main>
a
x
b
d
Done!
*Main> Leaving GHCi.
```

Alternatively, we could try manually running the exact command the tester ran:

```
$ ./tesths letters.hs 'letters 4'
*Main> "TESTING START"
*Main>
a
x
b
d
Done!
*Main> Leaving GHCi.
```