Using the Tester
CSC 352
September 3, 2015

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, "the Tester", to test your solutions for programming problems. For each assignment I'll make available a "student set" of tests that you can run yourself using the Tester. The set of tests used when grading (the "grading set") will often have additional tests. Unless otherwise specified for a problem or an entire assignment, passing all the tests in the student set will guarantee at least 75% of the points for a given problem. It some cases it will be higher, or even 100%, with the student set used as the grading set. I'll call that 75% minimum the "student set guarantee" and ensure that searching the write-up for "student set" will find any exceptions.

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 I had an extra space at the of a line."; 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.

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

You'll use the Tester by running `aN/tester`, where `N` is the assignment number. For assignment 2 you'll be using `a2/tester`.

Quick summary of usage, for the TL;DR crowd

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

```
% a2/tester lengths.java
```

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

You can also name multiple problems to be tested:

```
$ a2/tester iota revnum
```

The above case also shows that the `.java` suffix is not required for a Java problem.

I recommend testing problems one at a time, as you develop them, but when you are ready test all problems in sequence, you can run `a2/tester` with no arguments. Here we combine that with `grep` to look for failures:

```
% a2/tester | grep FAIL
```

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

GREAT IDEA: Do `a2/tester | grep FAIL` as a final double-check before running `a2/turnin`.

Another handy symlink

Along with the `a2` the symbolic link that the assignment 2 write-up asks you to create in your assignment 2 directory, let's make one more symlink:

```
% ln -s a2/tester t
```

Remember: I like short names for things that are used a lot.

With that symlink, `t`, you can say, for example, `./t revnum`.

More detail on running the tester

To avoid tying this write-up to specific problems in this semester's assignment we're going to imagine a new problem: *Write a version of `args.java` that also prints an argument count.* Example:

```
% java args one 2 III
3 arguments:
|one|
|2|
|III|
```

For this example problem, I'm using a different testing script: `a2/tester-ex`.

Here's a test run with no failures:

```
% a2/tester-ex args

------------------------------------------------------------
|                                                          |
|                           args                           |
|                                                          |
------------------------------------------------------------


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


Test: 'ulimit -t 1; java args': PASSED

Test: 'ulimit -t 1; java args one 2 III': PASSED
```

Those two lines starting with "`Test: `" indicate that two tests were run. Both passed. I've underlined

the output that shows the cases that are being tested. The first case, `java args`, checks behavior with no arguments. The second test, `java args one 2 III`, checks behavior with three arguments.

The `Test:` lines start with `ulimit -t 1;`. That limits CPU time for the test to one second, which is way more than these should ever use. (Try `time java args`.) The text that follows that semicolon, up to the final apostrophe, is the exact command that was run. Let's try each of them. These commands are simple but in general I recommend a copy/paste to avoid typos.

```
% java args
0 arguments:

% java args one 2 III
3 arguments:
|one|
|2|
|III|
```

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 `args` by removing the colon that's output after "`arguments`". Both tests fail but the second is more interesting, so we'll look at it.

I'll start a new page to get the output and explanation together on one page.

Here's the failure, with line numbers added for reference. Note that line 3 is long and has wrapped around.

```
% a2/tester-ex args
[...header lines and first test not shown...]

1.  Test: 'ulimit -t 1; java args one 2 III': FAILED
2.  Differences (expected/actual):
3.  *** /cs/www/classes/cs352/fall15/a2/master/tester.out/args.out.02
         2015-09-03 01:57:22.990919630 -0700
4.  --- tester.out/args.out.02    2015-09-03 02:40:18.539495345 -0700
5.  ***************
6.  *** 1,4 ****
7.  ! 3 arguments:
8.    |one|
9.    |2|
10.   |III|
11. --- 1,4 ----
12. ! 3 arguments
13.   |one|
14.   |2|
15.   |III|
```

Lines 3 and 4 name the two files that are being "diffed" (compared). The first is the file that contains the expected output, `...fall15/a2/master/tester.out/args.out.02`. The second, `tester.out/args.out.02`, contains the output produced by running `java args one 2 III`. You can look at both files with `cat`, `less`, editors (local or remote), etc.

Note that the Tester creates a directory, `tester.out`, to hold various files created during testing.

The file name on line 3 is preceded by `***`. Circle those three asterisks, and the three asterisks on line 6, and connect them with a line. The three asterisks on line 6, matching the three asterisks on line 3, and the " `1,4` " that follows, indicates that the following four lines (numbered 7-10) are the first four lines in `.../fall15/a2/master/tester.out/args.out.02`. Repeat that circling and connecting with the three dashes (`---`) on lines 4 and 11, which show that lines 12-15 are the first four lines in `tester.out/args.out.02`.

Diffs in tester output always follow the convention of showing the expected output first and the actual output second.

The exclamation marks on lines 7 and 12 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 colon.

For a more interesting "diff", let's further foul-up `args.java`. We'll skip the second argument and then print a couple of chatty lines at the end:

```
public class args {
    public static void main(String args[]) {
        System.out.println(args.length + " arguments");
        for (int i = 0; i < args.length; i++)
            if (i != 1)
                System.out.println("|" + args[i] + "|");
```

```
        System.out.println("Done!\nThanks for running me!");
        }
    }
```

Here's what we get with the new version, with line numbers added to aid explanation. Again, we'll look at the second diff.

```
% a2/tester-ex args
[...header lines and first diff not shown...]
1.  Test: 'ulimit -t 1; java args one 2 III': FAILED
2.  Differences (expected/actual):
3.  *** /cs/www/classes/cs352/fall15/a2/master/tester.out/args.out.02
         2015-09-03 01:57:22.990919630 -0700
4.  --- tester.out/args.out.02    2015-09-03 03:06:56.463577195 -0700
5.  ***************
6.  *** 1,4 ****
7.  ! 3 arguments:
8.    |one|
9.  - |2|
10.   |III|
11. --- 1,5 ----
12. ! 3 arguments
13.   |one|
14.   |III|
15. + Done!
16. + Thanks for running me!
```

We see in line 1 that `java args one 2 III` was run for this case.

Lines 3 and 4 identify the two files being diffed. The asterisks on line 6 show that lines 7-10 come from the file, named on line 3, that holds the expected output. Similarly, the dashes on line 11 show that lines 12-16 come from the file holding the actual output, named on line 4.

First, we still see the missing colon evidenced by lines 7 and 12.

The minus sign on line 9 indicates that the line "`|2|`" only appears in the expected output. I think of the minus sign as indicating the line has been subtracted from the actual output.

The plus signs on lines 15-16 indicates that those lines only appear in the actual output. I think of the plus signs as indicating those lines have been added to the actual output.

If we have trouble understanding a diff we might next look at the expected and actual files themselves.

```
% cat a2/master/tester.out/args.out.02
3 arguments:
|one|
|2|
|III|
```

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

```
% cat tester.out/args.out.02
```

```
3 arguments
|one|
|III|
Done!
Thanks for running me!
```

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

```
% java args one 2 III
3 arguments
|one|
|III|
Done!
Thanks for running me!
```

For complex differences you might open the expected and actual files in side-by-side windows in an editor. A simple form of that is provided by `vimdiff`: (type `:q<ENTER>` TWICE to get out!)

```
% vimdiff a2/master/tester.out/args.out.02 tester.out/args.out.02
```

It's not shown in the examples above but following the `diff` output is a line showing the names of the files that were diffed:

```
...
Test: 'ulimit -t 1; java args one 2 III': FAILED
Differences (expected/actual):
...
+ Done!
+ Thanks for running me!

Files diffed:
a2/master/tester.out/args.out.02 tester.out/args.out.02
```

That's provided so you can select the whole line with multiple clicks, type `vimdiff` or some other command and then paste both file names onto that line.

`a2/args-goofy.java` is a copy of the hacked-up `args.java` above.

"Killed"

The tester doesn't do a very good job of handling programs that go into an infinite loop. Below is an example with a version of `args.java` that has an infinite loop. I see two shortfalls: (1) The line "`Killed`" is the only indication we have of an infinite loop. (2) That "`Killed`" appears before the "`Test: ...`" line for the case that actually went into a loop.

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

```
Killed

Test: 'ulimit -t 1; java args': FAILED
Differences (expected/actual): ...
```