# Using the Tester
# (and some policies on testing)
# CSC 372
# Last updated: September 10, 2022

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".

Unless otherwise specified or extenuating circumstances arise (such as a bonehead error on my part), the student set will "freeze" 72 hours before the assignment's deadline. By "freeze" I mean that no additional tests will be added—if you pass all tests for a problem at T-72 hours, you're guaranteed at least 75% of the points for that problem.

Test cases in the grading set are weighted. Those weights are not supplied in the student set but a rule of thumb is that cases for basic functionality are weighted more than edge 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, "Why didn't you use the Tester?"

## The Tester is on lectura

The Tester is a collection of bash scripts, Ruby programs, an Icon program, and assorted files that are all on lectura. The instructions in this document assume that you're working on lectura. (The tester will run as-is on macOS and Cygwin but there are logistical issues to deal with, like staying in sync with the testing configuration on lectura. See Help Wanted, below.)

You'll use the Tester by running `aN/tester`, where `N` is the assignment number. For assignment 3 you'll be using `a3/tester`. Note that the path, `a3/tester`, assumes that you've made an `a3` symlink, as described in the `a3` write-up. For convenience, `a3/t` is symlinked to `a3/tester` but you might do `ln -s a3/tester t` to be able to run it with `./t`, for example. (Remember that we're happy to help you with UNIX stuff.)

## When to use the Tester

If you're doing TDD with an xUnit tool like JUnit, it's appropriate to run tests frequently, typically after almost every change, but that's not the intention with the Tester. I recommend that you first manually test your code with examples from the write-up and other cases that come to mind. When things are look right to the naked eye, then that's the time to run the Tester.

There is an HUnit for Haskell, as well as Hspec, QuickCheck and more, but I haven't seen anything that

looks likely to provide more benefit than trouble for our modest needs in 372.

**Quick summary of usage, for the TL;DR crowd**

With the `a3` symlink in place and your solution for `warmup.hs` in the current directory, you can test it like this:

```
% a3/tester warmup.hs
```

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

`warmup.hs` is unusual because it contains many functions that are tested. You'll see that running `a3/tester warmup.hs` tests *all* of them. That can produce a lot of useless output, especially if you've written only one of those functions. Testing can be limited to a single function (or several) by following `warmup.hs` with a `-t`:

```
a3/tester warmup -t lst
```

You can also name multiple problems to be tested:

```
% a3/tester join cpfx warmup
```

The above also shows that the `.hs` suffix is not required.

It's a tedious time-killer to scroll back looking for the start of the Tester's output. One approach is to pipe into `less`:

```
% a3/tester warmup | less
```

An alternative on macOS with Terminal is to do `cmd-K` before running the Tester—that clears the scrollback, so when you scroll back, you'll be at the start of the Tester output. I don't know of a keyboard shortcut to clear the scrollback with PuTTY but `fall22/bin/clr` is a two-line script that clears PuTTY's scrollback. (Let me know if you know a simpler way to do that.)

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

```
% a3/tester | grep FAIL
```

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

**GREAT IDEA: Do `a3/tester | grep FAIL` as a final double-check before your final `a3/turnin`.**

An interesting twist, an Original Thought by Chioke Aarhus, 372 Spring '16, is to discard the `PASSED` lines:

```
% a3/tester | grep -v PASSED
```

**More detail on running the tester**

For the purpose of this example we'll imagine that there are two more problems on `a3`: `hello.hs` and `letters.hs`.

Let's work with a simple "hello" function, whose code is correct and is in the file `hello.hs`:

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

% ghci hello.hs
...
> :type hello
hello :: [Char] -> [Char]

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

Here's a test run with no failures:

```
% a3/tester hello

------------------------------------------------------------
|                                                          |
|                          hello                           |
|                                                          |
------------------------------------------------------------


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


Test: 'ulimit -t 2; a3/tesths hello.hs ':type hello'': PASSED

Test: 'ulimit -t 2; a3/tesths hello.hs 'hello "world"'': PASSED
```

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

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

```
% a3/tesths hello.hs ':type hello'
*Main> > > "TESTING START"
> hello :: [Char] -> [Char]
> Leaving GHCi.

% a3/tesths hello.hs 'hello "world"'
*Main> > > "TESTING START"
> "Hello, world!\n"
> Leaving GHCi.
```

`a3/tesths` is a bash script that loads the named file with `ghci` and then feeds the third argument, such as "`:type hello`", into `ghci`.

Note: If you include the `ulimit -t 2;` when trying a test, like this,

```
% ulimit -t 2; a3/tesths hello.hs 'hello "world"'
```

you'll set the CPU time limit to two seconds for all future commands in that instance of bash. If you inadvertently do that, you'll need to log out and log in again to clear it. (An ordinary user can decrease their CPU time limit, but cannot raise it.)

**Understanding differences reported by the Tester**

If a test fails, the `diff` command is to used to show the differences 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 Tester-specific examples.

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

```
% a3/tester hello.hs
[...header lines not shown...]

1.  Test: 'ulimit -t 2; a3/tesths hello.hs ':type hello'': PASSED
2.
3.  Test: 'ulimit -t 2; a3/tesths hello.hs 'hello "world"'': FAILED
4.  Differences (expected/actual):
5.  *** a3/master/tester.out/hello.out.02   2016-01-28
    12:52:52.292586244 -0700
6.  --- tester.out/hello.out.02   2016-01-28 23:22:41.190616251
    -0700
7.  ***************
8.  *** 1,3 ****
9.    *Main> > > "TESTING START"
10. ! > "Hello, world!"
11.   > Leaving GHCi.
12. --- 1,3 ----
13.   *Main> > > "TESTING START"
14. ! > "Hello world!"
15.   > Leaving GHCi.
```

The type of `hello` is unaffected by removing that comma but the output differs, so the first test still passes but the second test now fails.

Lines 5 and 6 name the two files that are being "diffed" (compared). I've underlined and bolded the file names. The first is the file that contains the expected output, `a3/master/tester.out/hello.out.02`. The second, `tester.out/hello.out.02`, contains the output produced by running `a3/tesths hello.hs 'hello "world"'` in the current directory. That directory, `tester.out`, was created in the current directory by the Tester, to hold various files created by the testing process. Needless to say, you can look at both files with `cat`, `less`, editors, or any other tool.

The names of the files being compared are preceded by `***` and `---` , which are used later, on lines 8 and 12, to identify the files those blocks of text come 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", let's work with a function named `letters` that prints the first N lower-case letters, one per line. Like the examples in the section *A little output* in the slides, 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 the Tester shows with our buggy version, with line numbers added to aid explanation:

```
% a3/tester letters.hs
[...header lines not shown...]

1.  Test: 'ulimit -t 2; a3/tesths letters.hs 'letters 4'': FAILED
2.  Differences (expected/actual):
3.  *** a3/master/tester.out/letters.out.01 2016-01-28
    12:58:44.406350815 -0700
4.  --- tester.out/letters.out.01 2016-01-28 23:25:53.772537760
    -0700
5.  ***************
6.  *** 1,6 ****
7.    *Main> > > "TESTING START"
8.  ! > a
9.    b
10. - c
11.   d
12.   > Leaving GHCi.
13. --- 1,8 ----
14.   *Main> > > "TESTING START"
15. ! >
16. ! a
17. ! x
18.   b
19.   d
20. + Done!
21.   > Leaving GHCi.
```

We see in line 1 that `letters 4` is the Haskell expression that's being tested.

Lines 3 and 4 identify the two files being diffed. Note that line 3 wraps around. 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 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 appear in the expected output.

If we have trouble understanding a diff, it often helps to directly examine the files being diffed. Here's the file with the expected output:

```
% cat a3/master/tester.out/letters.out.01
*Main> > > "TESTING START"
> a
b
c
d
> Leaving GHCi.
```

Here's what was actually output when tested:

```
% cat tester.out/letters.out.01
*Main> > > "TESTING START"
>
a
x
b
d
Done!
> Leaving GHCi.
```

Of course, instead of looking at the file with the actual output, we could try manually running the exact command the tester ran:

```
% a3/tesths letters.hs 'letters 4'
*Main> > > "TESTING START"
>
a
x
b
d
Done!
> Leaving GHCi.
```

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 a3/master/tester.out/letters.out.01 tester.out/letters.out.01
```

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 2; a3/tesths letters.hs 'letters 4'': FAILED
Differences (expected/actual):
...
```

```
+ Done!
  > Leaving GHCi.

Files diffed:
a3/master/tester.out/letters.out.01 tester.out/letters.out.01
```

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.

`pr -mT` provides a simple side-by-side display of two files:

```
% pr -mT a3/master/tester.out/letters.out.01 tester.out/letters.out.01
*Main> > > "TESTING START"          *Main> > > "TESTING START"
> a                                 >
b                                   a
c                                   x
d                                   b
> Leaving GHCi.                     d
                                    Done!
                                    > Leaving GHCi.
```

`diff -y`, which produces a side-by-side diff, is sometimes useful.

If `diff` is claiming a difference but the text looks identical, the problem might be trailing whitespace or embedded non-printable characters, like NULs (ASCII code 0). Problems like that can be turned up by piping into `cat -A`:

```
% a3/tester hello | cat -A
...
```

**Exceeding the time limit—handled poorly...**

A bug in a recursive function can produce infinite recursion. Infinite recursion will cause the test's time limit to be exceeded and the test will be killed. Sadly, the Tester doesn't provide any clear evidence of the time limit being exceeded. Here's what we see for a diff with a version of `hello` that infinitely recurses:

```
Test: 'ulimit -t 2; a3/tesths hello.hs 'hello "world"'': FAILED
Differences (expected/actual):
*** a3/master/tester.out/hello.out.02   2016-01-28
12:52:52.292586244 -0700
--- tester.out/hello.out.02     2016-01-28 23:36:23.020876709 -0700
***************
*** 1,3 ****
  *Main> > > "TESTING START"
! > "Hello, world!\n"
! > Leaving GHCi.
--- 1,2 ----
  *Main> > > "TESTING START"
! >
\ No newline at end of file
```

Note two things: (1) The actual output doesn't end with "`Leaving GHCi.`" (2) `diff` says there's no newline at the end of the file. The combination of those two things typically indicates the time limit was exceeded.

A good next step is to try the command yourself, without a time limit. Let's try it outside the Tester:

```
 % a3/tesths hello.hs 'hello "world"'
...wait a while...give up...hit ^C
```

The time limits set for tests are usually far more than what's needed but in rare cases you may find that your solution is simply slow, and that it does complete when run outside the Tester. If so, let us know. If it's not outrageously slow, we might just bump up the time limit on the test.

**Help Wanted**

If you know Ruby and would like to help make some improvements on the Tester, let me know. The Ruby code is in `a3/rtester`, if you'd like to see what you might be dealing with, which ain't pretty, but works.