Lab Due Date: Friday, June 6th, 5:00pm

⚠️ **This is a difficult lab.** For some students, it is the most difficult lab in the course. This is also a lab where students are frequently caught cheating.

Our suggestions:

- Start the lab early and come to the in-person lab sessions.
- Review the Academic Integrity policy for the course.

Pass Threshold:
Question 1: at least **3 tests** must be passed.
Question 2: at least **4 tests** must be passed.
Question 3: at least **5 tests** must be passed.
Question 4: at least **6 tests** must be passed.

Completion Threshold:
Question 1: at least **6 tests** must be passed.
Question 2: at least **6 tests** must be passed.
Question 3: at least **10 tests** must be passed.
Question 4: at least **10 tests** must be passed.

Question 1: Arithmetic Test Suite

In this question you are asked to write a **test suite** for a script named `arithmetic.sh`.

- The `arithmetic.sh` script expects exactly one command line argument; the character `a` (for add) or `s` (for subtract).
- The script expects to read three integers from standard input, **separated by newline characters**.
- The script outputs `Yes` if the operation (add or subtract, as determined by the command line argument) applied to the first two numbers read from standard input equals the third number read from standard input. Otherwise, the script outputs `No`.

For example, if the command line argument is `a`, then the three inputs 2, 3, and 5 (**with a newline after each input**) will result in `Yes` since adding 2 and 3 produces 5.
On the other hand, if the command line argument is `s`, then the three inputs 3, 4, and 7 (**with a newline after each input**) will result in `No` since 3 minus 4 produces -1, which is not 7.

If the script is not called with exactly one argument that is either the character `a` or `s`, the script should output an appropriate usage message **on the standard error stream** and exit with an **exit code of 4**.

The script may assume that the user always inputs three integers, i.e., the script does not need to perform error checking on the input from the standard input stream.

See below for an environment where you can explore the script.

What to submit: As in Lab 2, you are creating a **test suite** for the `arithmetic.sh` script. You are not writing the actual script yet (you will do this in Question 2).

Since the script expects a command line argument as well as input from standard input, each test will consist of three files. For a test with stem `mytest`, you would create a file `mytest.args` with the command line argument you want to give, a file `mytest.in` with the input you want to give on standard input, and a file `mytest.expect` with the output you expect the test to produce on standard output.

Creating `.expect` files: Unlike in Lab 2, we do not provide a "correct implementation" of `arithmetic.sh` in the Linux Student Environment that can be used to automatically generate the `.expect` files. You will need to create the `.expect` files manually. Since the output is just `Yes` or `No`, the easiest way to do this is probably to use the `echo` command with output redirection:

```
echo Yes > mytest.expect
```

Once you have created all your tests, create a file **arithmetic.txt** containing the stems (names without extensions) of all your tests, just like the `change.txt` and `gas.txt` files from Lab 2.

Error test cases

Note that the script's specification sometimes mentions producing output on the **standard error stream** instead of the standard output stream. The `.expect` files in our testing setup are really *expected standard output* files and we do not have a way to test what is produced on the standard error stream (though our testing setup could be extended to do this).

However, we do know that when an error occurs, *no output* is expected on the standard output stream. This means you can create tests that check for error conditions by creating `.expect` files that are **empty**.

A word of warning: an empty file is a file that contains nothing; a text file that contains a *single blank line* is not an empty file. You can create empty files in Linux by using the `touch` command. For example: `touch mytest.expect` creates an empty file named `mytest.expect`. It will contain nothing (to confirm run `wc mytest.expect`).

Submitting to Marmoset

The following command will submit your arithmetic.txt file as well as all the .args, .in, and .expect files in the current directory.

```
/u/cs_build/bin/marmoset submit cs136l lab3q1 arithmetic.txt *.args *.in
*.expect
```

If you only want to submit certain specific test cases, rather than all the test cases in the current directory, you will need to list the files you want to submit manually.

```
/u/cs_build/bin/marmoset submit cs136l lab3q1 arithmetic.txt test1.args
test1.in test1.expect ...
```

Exploring the arithmetic script

Unfortunately, our edX environment does not support interactive sessions on the command line and is limited to one request and one response. As a compromise, we have designed the following strategy. In the environment below, you have access to our version of the `arithmetic.sh` script. To use it, you will first need to create a text file which contains the three integer values you want to test. Then, you would run the script and redirect input from the text file you just created. The environment below is pre-populated with one example to get you started.

Feel free to use this environment to reverse engineer the requirements if they are unclear. Please do not attempt to reverse engineer the source (although if you do, we would be very curious to know how you did it).

Explore the expected behaviour of arithmetic.sh

```
echo 1 > test.in # notice how > will overwrite any previous test.in
echo 2 >> test.in # notice the use of >> to append to file
echo 3 >> test.in # notice the use of >> to append to file
echo s > test.args
./arithmetic.sh $(cat test.args) < test.in
```

Code Output

```
No
```

Question 2: Arithmetic Script

Implement the `arithmetic.sh` script using the specifications from the previous question.

That is, as discussed in the module, you should create a file (using `vim` or another text editor) called `arithmetic.sh` that begins with the line `#!/bin/bash` and is followed by a sequence of Bash commands that implements the functionality described in Question 1.

⚠️ To pass all the tests, your script must have proper **error handling**, which includes *printing error messages to standard error* (not to standard output) and *exiting with the specified exit code*. The exact content of your error messages does not matter but error messages must be present and printed to the correct stream.

We recommend creating this file **in the same directory as Question 1** so that you can use your test cases from Question 1 to test your script, as described below.

Testing your script

It is very important to test your `arithmetic.sh` script yourself, as Marmoset often does not give useful feedback if there is an error in your script, particularly for syntax errors.

After writing your script, use the `chmod` command to give the executable permission to the user (yourself).

```
chmod u+x arithmetic.sh
```

Then you can run your test cases from Question 1 to see if the script works correctly. For example, to run the test case with stem `mytest` you would write:

```
./arithmetic.sh $(cat mytest.args) < mytest.in
```

The output will appear on the terminal and you can visually confirm that there are no errors and that output matches the content of the `.expect` file.

It is also possible to programmatically check whether the output matches the `.expect` file using the `diff` command (something we will explore further in Question 4). To do this, first redirect the output of `arithmetic.sh` to a separate file, such as `mytest.out`, then use the `diff` command to compare `mytest.out` with `mytest.expect`:

```
./arithmetic.sh $(cat mytest.args) < mytest.in > mytest.out
diff mytest.out mytest.expect
```

If the `diff` command produces **no output** that means there is no difference between the files (the test passed). Otherwise the test failed, and the `diff` command will produce some output showing the difference between the `mytest.out` and `mytest.expect` files.

You can also check the exit code of `diff` by using the command `echo $?` which must be run immediately after `diff`. If it prints 0 that means there was no difference (test passed), if it prints 1 that means there was a difference (test failed). Other values mean an error occured with the `diff` command itself.

Once you have confirmed your script is working, submit your implementation to Marmoset:

```
/u/cs_build/bin/marmoset submit cs136l lab3q2 arithmetic.sh
```

Question 3: Wrapper

In this question, you will write a wrapper script named `wrapper.sh` that takes the stem (name without extension) of a test case as a command line argument, and runs `arithmetic.sh` with that test case.

- The script takes exactly one command line argument, a string which is the `stem` for a test case.

- The script looks for files in the current directory called `stem.args` and `stem.in`.
- If exactly one argument is not provided, the script exits with **exit code 4** and prints an appropriate error message to **standard error**.
- If either of the files `stem.args` or `stem.in` does not exist or is not readable, the script exits with **exit code 5** and prints an appropriate error message to **standard error**.
- Otherwise, if all preconditions are satisfied, the script calls the `arithmetic.sh` script with arguments supplied from `stem.args` and standard input redirected from `stem.in`.

We assume that `arithmetic.sh` is in the same directory as `wrapper.sh`, and that when executing `wrapper.sh`, our current directory is the same directory where these two scripts are located.

In the environment below you can explore the behaviour of `wrapper.sh` in more detail.

Once you have finished and tested your `wrapper.sh` script, submit it to Marmoset:

```
/u/cs_build/bin/marmoset submit cs136l lab3q3 wrapper.sh
```

Explore the behaviour of the wrapper program

```
echo 1 > mytest.in # create input file
echo 2 >> mytest.in # append to input file
echo 3 >> mytest.in # append to input file
echo s > mytest.args # create arguments file
./wrapper.sh mytest # execute wrapper giving it the stem of our input a
```

Let's automate I/O Testing

In the previous module on Testing and Debugging we had mentioned that we will help you create your own automated script for regression testing; during development, as you fix bugs and refine your code, you will often need to rerun old tests, to check that existing bugs have been fixed, and to ensure that no new bugs have been introduced. This task is greatly simplified if you take the time to create a formal test suite, and build tools to automate your testing. In the following problem, you will implement such a tool as a Bash script. The script will be useful every time you write a program that takes input/arguments and generates output (not only in CS136 and CS246 but even for non-course related development). Be sure to complete it!

Question 4: Test Suite Runner

Create a Bash script called `runSuite.sh` that is invoked as follows:

```
./runSuite.sh suite-file program
```

The argument `suite-file` is the name of a file containing a list of test stems, and the argument `program` is the name of the program to be tested.

The `runSuite.sh` script runs `program` on each test in the test suite (as specified by `suite-file`) and reports on any tests whose output does not match the expected output.

Example

The `suite-file` contains a list of test stems, with each stem separated by whitespace (which may be spaces *or* newlines). For example, suppose our suite file is called `suite.txt` and contains the following data:

suite.txt

```
test1 test2
reallyBigTest
```

Then our test suite consists of three test cases named `test1`, `test2` and `reallyBigTest`.

The `runSuite.sh` script will run the `program` with each of the three tests, much like the `wrapper.sh` script did with `arithmetic.sh`.

For example, suppose `runSuite.sh` is invoked as follows:

```
./runSuite.sh suite.txt ./myprogram
```

Then `runSuite.sh` will call `./myprogram` three times, one for each of the test cases.

Let's assume `./myprogram` takes input from both command line arguments and standard input. This means each test case will have three associated files, which are assumed to be in the current directory:

- For `test1` we have `test1.args`, `test1.in` and `test1.expect`.
- For `test2` we have `test2.args`, `test2.in` and `test2.expect`.
- For `reallyBigTest` we have `reallyBigTest.args`, `reallyBigTest.in` and `reallyBigTest.expect`.

For each test case, `runSuite.sh` will run `./myprogram` one time, providing it with command line arguments from the `.args` file and providing it with standard input redirected from the `.in` file.

Missing `.args` and `.in` files

Some programs do not take command line arguments, or do not take input from standard input, and some programs do not take input at all. This means test cases for those programs might not include a `.args` file, or not include a `.in` file, or not include either file.

If the `.args` or `.in` file for a test does not exist (or exists but is not readable), this is not considered an error. Instead, the `program` is simply executed without providing command line arguments (if the `.args` file is unavailable), without providing standard input (if the `.in` file is unavailable), or with neither (if neither file is available).

The standard output from running the program is saved to a temporary file. Then, `runSuite.sh` compares the standard output to the contents of the `.expect` file. If the contents **do not match**, this means the test failed, and `runSuite.sh` outputs information about the test that failed.

Missing .expect files

While the `.args` and `.in` files are optional, every test must include a `.expect` file, or else `runSuite.sh` must report an error.

For example, if the test called `test2` failed, `runSuite.sh` would output the following:

```
Test failed: test2
Args:
(contents of test2.args, if it exists and is readable)
Input:
(contents of test2.in, if it exists and is readable)
Expected:
(contents of test2.expect)
Actual:
(contents of actual standard output from the program)
```

with the `(contents ...)` lines replaced with the actual file contents, as described.

Note that the lines `Args:` and `Input:` are still printed even if the `.args` or `.in` files do not exist or are not readable. For clarity, the following output would be produced if `test2.args` and `test2.in` did not exist or were not readable:

```
Test failed: test2
Args:
Input:
Expected:
(contents of test2.expect)
Actual:
(contents of actual standard output from the program)
```

Specifications

- The script is to be invoked with two command line arguments: `./runSuite.sh suite-`

`file program`

- The first argument, `suite-file`, is a text file containing the list of test stems.
  - You may assume this file exists and is readable (you do not need to check this in your script).
  - Test stems are separated by whitespace. Whitespace can be newlines *or* simply spaces, so multiple stems might appear on the same line.
- The second argument, `program`, is the program to run the tests against.
  - You may assume this program exists, is readable, and is executable (you do not need to check this in your script).
  - You may assume the command line argument corresponds to how you would run the program. For example, if the program is named `myprogram` and you need to type `./myprogram` to run it, then the provided command line argument will be `./myprogram` rather than `myprogram`. You do not need to (and should not) manually prepend `./` to the command line argument.
- If an incorrect number of arguments is provided, the script should produce an appropriate error message to **standard error** and exit with a **non-zero exit code**.
- For each `stem` in the `suite-file`, the program should perform the following actions:
  - If `stem.expect` does not exist or is not readable, the script should produce an appropriate error message to **standard error** and exit with a **non-zero exit code**.
  - Run `program`, providing command line arguments from `stem.args` (if it exists and is readable), redirecting `stem.in` into standard input (if it exists and is readable), and redirecting standard output from the program into a temporary file (see the *Hints* section below for how to create temporary files).
  - Compare the standard output from the program to the `stem.expect` file. If there is no difference, the test passed; output nothing and move on to the next stem. If there is a difference, then the test **failed** and you must output the following information (to standard output) in the exact format given below, with the `(contents ...)` parts replaced with the actual contents of the relevant file.

    ```
    Test failed: stem
    Args:
    (contents of stem.args, if it exists and is readable)
    Input:
    (contents of stem.in, if it exists and is readable)
    Expected:
    (contents of stem.expect)
    Actual:
    (contents of actual standard output from the program)
    ```

    Follow these output specifications very carefully. You will fail most tests if your output does not exactly match the specification, including for minor issues like incorrect capitalization, missing punctuation, or extra whitespace.

Hints

Temporary files: To create a temporary file, use the `mktemp` command, which avoids filename duplication. You can use the following code:

```
TEMPFILE=$(mktemp)
```

After the above command runs, the `TEMPFILE` variable will contain the name of a file, which you can access using `$TEMPFILE`. In the Linux Student Environment, the `mktemp` command defaults to creating the temporary file in the directory at the absolute path `/tmp/`, which is great because then the file will not clutter our working directory.

You must still ensure that you delete any temporary files that you create:

```
rm $TEMPFILE
```

Note: `mktemp` returns the absolute path to the temporary file, and not just the name of the temporary file.

Comparing outputs with `diff`: Do not attempt to compare outputs by storing them in a shell variable and then using string comparison. This does not scale well to programs that produce large outputs. Instead, use the `diff` command to determine if there are differences between the expected and actual output.

When two files are compared with `diff`, the exit code is 0 if no differences are found, and non-zero if there are differences. The following code shows an example of using `diff` to compare the files `test.out` and `test.expect`:

```
diff test.out test.expect > /dev/null
if [ $? -ne 0 ]; then
  echo "Files are different"
fi
```

Note that if the files are different, then `diff` will produce output showing the differences between the files. We do not want this output in our script, which is why we redirect standard output from `diff` to `/dev/null` to suppress it.

Testing your implementation of `runSuite.sh`: At the path `/u2/cs136l/pub/lab3/q4` you will find a directory containing example programs with test suites that you can use to test your implementation. Begin by reading the file named `README.txt` which explains how the `/u2/cs136l/pub/lab3/q4/zigzag` directory is organized.

You can also test with the test suites you created in Lab 2 and Lab 3 Question 1.

Submitting to Marmoset

Once you have confirmed your `runSuite.sh` script is working, submit it to Marmoset:

```
/u/cs_build/bin/marmoset submit cs136l lab3q4 runSuite.sh
```