Assignment 1: shrug, Shrug

<div align="right">version: 0.4 last updated: 2020-07-06 16:00</div>

Aims

This assignment aims to give you

- practice in Shell programming generally
- a clear concrete understanding of Git's core semantics

Note: the material in the lecture notes will not be sufficient by itself to allow you to complete this assignment. You may need to search on-line documentation for Shell, Git, etc. Being able to search documentation efficiently for the information you need is a *very* useful skill for any kind of computing work.

Introduction

Your task in this assignment is to implement **Shrug**, a subset of the version control system [Git](#).

Git is a *very* complex program which has many individual commands. You will implement only a few of the most important commands. You will also be given a number of simplifying assumptions, which make your task easier.

Shrug is a contraction of **Sh**ell-**ru**nning-**g**it.

You must implement **Shrug** in Shell.

Interestingly, early versions of Git made heavy use of Shell and Perl.

Reference implementation

Many aspects of this assignment are not fully specified in this document; instead, you must match the behaviour of a reference implementation.

For example, your script `shrug-add` should match the behaviour of `2041 shrug-add` exactly, including producing the same error messages.

Provision of a reference implementation is a common method to provide or define an operational specification, and it's something you will likely need to do after you leave UNSW.

Discovering and matching the reference implementation's behaviour is deliberately part of the assignment.

While the code in the reference implementation is fairly straight forward, reverse-engineering its behaviour is obviously not so simple, and is a nice example of how coming to grips with the precise semantics of an apparently obvious task can still be challenging.

If you discover what you believe to be a bug in the reference implementation, report it in the class forum. Andrew may fix the bug, or indicate that you do not need to match the reference implementation's behaviour in this case.

Shrug Commands

Subset 0

Subset 0 commands must be implemented in POSIX-compatible Shell. See the **Permitted Languages** section for more information.

shrug-init

The `shrug-init` command creates an empty Shrug repository.

`shrug-init` should create a directory named `.shrug`, which it will use to store the repository. It should produce an error message if this directory already exists.

You should match this, and other error messages exactly. For example:

```
$ ls -d .shrug
ls: cannot access '.shrug': No such file or directory
$ shrug-init
Initialized empty shrug repository in .shrug
$ ls -d .shrug
.shrug
$ shrug-init
shrug-init: error: .shrug already exists
```

`shrug-init` may create initial files or directories inside `.shrug`.

You do not have to use a particular representation to store the repository.

You do not have to create the same files or directories inside `shrug-init` as the reference implementation.

shrug-add *filenames*...

The `shrug-add` command adds the contents of one or more files to the "**index**".

Files are added to the repository in a two step process. The first step is adding them to the index.

You will need to store files in the index somehow in the `.shrug` sub-directory. For example, you might choose store them in a sub-directory of `.shrug`.

Only ordinary files in the current directory can be added. You can assume filenames start with an alphanumeric character ([a-zA-Z0-9]) and will only contain alpha-numeric characters, plus '.', '-' and '_' characters.

The `shrug-add` command, and other Shrug commands, will not be given pathnames with slashes.

shrug-commit -m *message*

The `shrug-commit` command saves a copy of all files in the index to the repository.

A message describing the commit must be included as part of the commit command.

Shrug commits are numbered sequentially: they are not hashes, like Git. You must match the numbering scheme.

You can assume the commit message is ASCII, does not contain new-line characters, and does not start with a '-' character.

shrug-log

The `shrug-log` command prints a line for every commit made to the repository: each line should contain the commit number, and the commit message.

shrug-show *[commit]:filename*

The `shrug-show` should print the contents of the specified *filename* as of the specified *commit*. If *commit* is omitted, the contents of the file in the index should be printed.

You can assume the commit if specified will be a non-negative integer.

For example:

```
$ ./shrug-init
Initialized empty shrug repository in .shrug
$ echo line 1 > a
$ echo hello world >b
$ ./shrug-add a b
$ ./shrug-commit -m 'first commit'
Committed as commit 0
$ echo  line 2 >>a
$ ./shrug-add a
$ ./shrug-commit -m 'second commit'
Committed as commit 1
$ ./shrug-log
1 second commit
0 first commit
$ echo line 3 >>a
$ ./shrug-add a
$ echo line 4 >>a
$ ./shrug-show 0:a
line 1
$ ./shrug-show 1:a
line 1
```

Subset 1

Subset 1 is more difficult. You will need spend some time understanding the semantics (meaning) of these operations, by running the reference implementation, or researching the equivalent Git operations.

Note the assessment scheme recognises this difficulty.

Subset 1 commands must be implemented in POSIX-compatible Shell. See the **Permitted Languages** section for more information.

shrug-commit *[-a]* -m *message*

`shrug-commit` can have a -a option, which causes all files already in the index to have their contents from the current directory added to the index before the commit.

shrug-rm *[--force] [--cached] filenames*...

`shrug-rm` removes a file from the index, or from the current directory and the index.

If the `--cached` option is specified, the file is removed only from the index, and not from the current directory.

`shrug-rm`, like `git rm`, should stop the user accidentally losing work, and should give an error message instead if the removal would cause the user to lose work. You will need to experiment with the reference implementation to discover these error messages. Researching `git rm`'s behaviour may also help.

The `--force` option overrides this, and will carry out the removal even if the user will lose work.

shrug-status

`shrug-status` shows the status of files in the current directory, the index, and the repository.

```
$ ./shrug-init
Initialized empty shrug repository in .shrug
$ touch a b c d e f g h
$ ./shrug-add a b c d e f
$ ./shrug-commit -m 'first commit'
Committed as commit 0
$ echo hello >a
$ echo hello >b
$ echo hello >c
$ ./shrug-add a b
$ echo world >a
$ rm d
$ ./shrug-rm e
$ ./shrug-add g
$ ./shrug-status
a - file changed, different changes staged for commit
b - file changed, changes staged for commit
c - file changed, changes not staged for commit
d - file deleted
e - deleted
f - same as repo
```

Subset 2

Subset 2 is extremely difficult. You will need spend considerable time understanding the semantics of these operations, by running the reference implementation, and/or researching the equivalent Git operations.

Note the assessment scheme recognises this difficulty.

Subset 2 commands must be implemented in POSIX-compatible Shell. See the **Permitted Languages** section for more information.

shrug-branch *[-d] [branch-name]*

`shrug-branch` either creates a branch, deletes a branch, or lists current branch names.

shrug-checkout *branch-name*

`shrug-checkout` switches branches.

Note that, unlike Git, you can not specify a commit or a file: you can only specify a branch.

shrug-merge *branch-name|commit* -m *message*

`shrug-merge` adds the changes that have been made to the specified branch or commit to the index, and commits them.

```
$ ./shrug-init
Initialized empty shrug repository in .shrug
$ seq 1 7 >7.txt
$ ./shrug-add 7.txt
$ ./shrug-commit -m commit-1
Committed as commit 0
$ ./shrug-branch b1
$ ./shrug-checkout b1
Switched to branch 'b1'
$ perl -pi -e 's/2/42/' 7.txt
$ cat 7.txt
1
42
3
4
5
6
7
$ ./shrug-commit -a -m commit-2
Committed as commit 1
$ ./shrug-checkout master
```

If a file contains conflicting changes, `shrug-merge` produces an error message.

```
$ ./shrug-init
Initialized empty shrug repository in .shrug
$ seq 1 7 >7.txt
$ ./shrug-add 7.txt
$ ./shrug-commit -m commit-1
Committed as commit 0
$ ./shrug-branch b1
$ ./shrug-checkout b1
Switched to branch 'b1'
$ perl -pi -e 's/2/42/' 7.txt
$ cat 7.txt
1
42
3
4
5
6
7
$ ./shrug-commit -a -m commit-2
Committed as commit 1
$ ./shrug-checkout master
```

Diary

You must keep notes on each piece of work you make on this assignment. The notes should include date, starting and finishing time, and a brief description of the work carried out. For example:

Date	Start	Stop	Activity	Comments
19/06/19	16:00	17:30	coding	implemented basic commit functionality
20/06/19	20:00	10:30	debugging	found bug in command-line arguments

Include these notes in the files you submit as an ASCII file named `diary.txt`.

Testing

Autotests

As usual, some autotests will be available:

```
$ 2041 autotest shrug shrug-*
...
```

You can also run only tests for a particular subset or an individual test:

```
$ 2041 autotest shrug subset1 shrug-*
...
$ 2041 autotest shrug subset1_13 shrug-*
...
```

If you are using extra Shell files, include them on the autotest command line.

Autotest and automarking will run your scripts with a current working directory different to the directory containing the script. The directory containing your submission will be in `$PATH`.

You will need to do most of the testing yourself.

Test Scripts

You should submit ten Shell scripts, named `test00.sh` to `test09.sh`, which run shrug commands that test an aspect of Shrug.

The `test??.sh` scripts do not have to be examples that your program implements successfully.

You may share your test examples with your friends, but the ones you submit must be your own creation.

The test scripts should show how you've thought about testing carefully.

Permitted Languages

Your programs must be written entirely in POSIX-compatible shell.

Your programs will be run with *dash(1)*, in `/bin/dash`. You can assume anything that works with the version of `/bin/dash` on CSE systems is POSIX compatible.

Start your programs with:

```
#!/bin/dash
```

If you want to run these scripts on your own machine — for example, one running macOS — which has *dash(1)* installed somewhere other than `/bin`, use:

```
#!/usr/bin/env dash
```

You are permitted to use any feature `/bin/dash` provides.

On CSE systems, `/bin/sh` is the Bash (Bourne-again shell) shell: `/bin/sh` is a symlink to `/bin/bash`. Bash implements many non-POSIX extensions, including regular expressions and arrays. These *will not work* with `/bin/dash`, and you are not permitted to use these for the assignment.

You are not permitted to use Perl, Python or any language other than POSIX-compatible shell.

You are permitted to use only these external programs:

basename(1)	diff(1)	gzip(1)	pwd(1)	tac(1)	unxz(1)
bunzip2(1)	dirname(1)	head(1)	readlink(1)	tail(1)	unzip(1)
bzcat(1)	du(1)	hostname(1)	realpath(1)	tar(1)	wc(1)
bzip2(1)	echo(1)	less(1)	rev(1)	tee(1)	wget(1)
cat(1)	egrep(1)	ln(1)	rm(1)	test(1)	which(1)
chmod(1)	env(1)	ls(1)	rmdir(1)	time(1)	who(1)
cmp(1)	expand(1)	lzcat(1)	sed(1)	top(1)	xargs(1)
cp(1)	expr(1)	lzma(1)	seq(1)	touch(1)	xz(1)
cpio(1)	false(1)	md5sum(1)	sha1sum(1)	tr(1)	xzcat(1)
csplit(1)	fgrep(1)	mkdir(1)	sha256sum(1)	true(1)	yes(1)
cut(1)	find(1)	mktemp(1)	sha512sum(1)	uname(1)	zcat(1)
date(1)	fold(1)	more(1)	sleep(1)	uncompress(1)	
dc(1)	getopt(1)	mv(1)	sort(1)	unexpand(1)	
dd(1)	grep(1)	patch(1)	stat(1)	uniq(1)	
df(1)	gunzip(1)	printf(1)	strings(1)	unlzma(1)	

Only a few of the programs in the above list are likely to be useful for the assignment.

Note you are permitted to use built-in shell features including: `cd`, `exit`, `for`, `if`, `read`, `shift` and `while`.

If you wish to use an external program which is not in the above list, please ask in the class forum for it to be added.

You may submit extra Shell files.

Assumptions/Clarifications

Like all good programmers, you should make as few assumptions as possible.

You can assume `shrug` commands are always run in the same directory as the repository, and only files from that directory are added to the repository.

You can assume the directory in which `shrug` commands are run will not contain sub-directories apart from `.shrug`.

You can assume where a branch name is expected a string will be supplied starting with an alphanumeric character ([a-zA-Z0-9]), and only containing alphanumeric characters plus '-' and '_'. In additon a branch name will not be supplied which is entirely numeric. This allows brnach names to be distinguished from commits when merging.

You can assume where a filenames is expected a string will be supplied starting with an alphanumeric character ([a-zA-Z0-9]) and only containing alpha-numeric characters, plus '.', '-' and '_' characters.

You can assume where a commit number is expected a string will be supplied which is a non-negative integer with no leading zeros. It will not contain white space or any other charcters except digits.

You can assume that `shrug-add`, `shrug-show`, and `shrug-rm` will be given just a filename, not pathnames with slashes.

You do not have to consider file permissions or other file metadata. For example, you do not have to ensure files created by a checkout command have the same permissions as when they were added.

You do not have to handle concurrency. You can assume only one instance of any `shrug` command is running at any time.

You can assume that only the arguments described above are supplied to `shrug` commands. You do not have to handle other arguments.

You should match the output streams used by the reference implementations. It writes error messages to stderr: so should you.

You should match the exit status used by the reference implementation. It exits with status 1 after an error: so should you.

Autotest and automarking will run your scripts with a current working directory different to the directory containing the script. This may break Shell with code in extra files: if so, ask for help in the forum. The directory containing your submission will be in `$PATH`.

You can assume arguments will be in the position and order shown in the usage message from the reference implementation. Other orders and positions will not be tested. For example, here is the usage message for `shrug-rm`:

```
$ 2041 shrug-rm
usage: shrug-rm [--force] [--cached] <filenames>
```

So, you assume that if the `--force` or `--cached` options are present, they come before all filenames, and if they are both present the `--force` option will come first.

Change Log

Version 0.1	• Initial release
(2020-06-26 14:00)	
Version 0.2	• bug in reference implementation fixed for shrug-status when file changed or removed after being added
(2020-07-06 21:00)	
Version 0.3	• bug in reference implementation fixed for shrug-show for file in index before first commit
(2020-07-06 16:00)	
Version 0.4	• add clarification about what can be supplied as arguments - fix minor bug in reference implementation for certain commit messages
(2020-07-06 16:00)	

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 2041 autotest shrug
```

Assessment

Submission

When you are finished working on the assignment, you must submit your work by running `give`:

```
$ give cs2041 ass1_shrug shrug-* diary.txt test??.sh [any-other-files]
```

You must run `give` before **Sunday July 15 21:59 2020** to obtain the marks for this assignment. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

You can run `give` multiple times. Only your last submission will be marked.

If you are working at home, you may find it more convenient to upload your work via [give's web interface](#).

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted [here](#).

Automarking will be run by the lecturer after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

Manual marking will be done by your tutor, who will mark for style and readability, as described in the **Assessment** section below. After your tutor has assessed your work, you can [view your results here](#); The resulting mark will also be available [via give's web interface](#).

Due Date

This assignment is tentatively due **Sunday July 15 21:59 2020**.

If your assignment is submitted after this date, each hour it is late reduces the maximum mark it can achieve by 2%. For example, if an assignment worth 74% was submitted 10 hours late, the late submission would have no effect. If the same assignment was submitted 15 hours late, it would be awarded 70%, the maximum mark it can achieve at that time.

Assessment Scheme

This assignment will contribute 15 marks to your final COMP(2041|9044) mark

15% of the marks for assignment 1 will come from hand-marking. These marks will be awarded on the basis of clarity, commenting, elegance and style: in other words, you will be assessed on how easy it is for a human to read and understand your program.

5% of the marks for assignment 1 will be based on the test suite you submit.

80% of the marks for assignment 1 will come from the performance of your code on a large series of tests.

An indicative assessment scheme follows. The lecturer may vary the assessment scheme after inspecting the assignment submissions, but it is likely to be broadly similar to the following:

| **HD (85+)** | All subsets working; code is beautiful; great test suite and diary |

DN (75+)	Subset 1 working; good clear code; good test suite and diary
CR (65+)	Subset 0 working; good clear code; good test suite and diary
PS (55+)	Subset 0 passing some tests; code is reasonably readable; reasonable test suite and diary
PS (50+)	Good progress on assignment, but not passing autotests
0%	knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 FL for COMP(2041|9044)	submitting any other person's work; this includes joint work.
academic misconduct	submitting another person's work without their consent; paying another person to do work for you.

Intermediate Versions of Work

You are required to submit intermediate versions of your assignment.

Every time you work on the assignment and make some progress you should copy your work to your CSE account and submit it using the `give` command below. It is fine if intermediate versions do not compile or otherwise fail submission tests. Only the final submitted version of your assignment will be marked.

All these intermediate versions of your work will be placed in a Git repository and made available to you via a web interface at `https://gitlab.cse.unsw.edu.au/`*z5555555*`/20T2-comp2041-ass1_shrug` (replacing *z5555555* with your own zID). This will allow you to retrieve earlier versions of your code if needed.

Attribution of Work

This is an individual assignment.

The work you submit must be entirely your own work, apart from any exceptions explicitly included in the assignment specification above. Submission of work partially or completely derived from any other person or jointly written with any other person is not permitted.

You are only permitted to request help with the assignment in the course forum, help sessions, or from the teaching staff (the lecturer(s) and tutors) of COMP(2041|9044).

Do not provide or show your assignment work to any other person (including by posting it on the forum), apart from the teaching staff of COMP(2041|9044). If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted, you may be penalized, even if that work was submitted without your knowledge or consent; this may apply even if your work is submitted by a third party unknown to you. You will not be penalized if your work is taken without your consent or knowledge.

Submissions that violate these conditions will be penalised. Penalties may include negative marks, automatic failure of the course, and possibly other academic discipline. We are also required to report acts of plagiarism or other student misconduct: if students involved hold scholarships, this may result in a loss of the scholarship. This may also result in the loss of a student visa.

Assignment submissions will be examined, both automatically and manually, for such submissions.

COMP(2041|9044) 20T2: Software Construction is brought to you by
the School of Computer Science and Engineering
at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2041@cse.unsw.edu.au
CRICOS Provider 00098G