

## CSSE2310/CSSE7231 — Semester 2, 2023 Assignment 3 (version 1.1 – 18 September 2023)

Marks: 75 (for CSSE2310), 85 (for CSSE7231)

Weighting: 15%

**Due: 4:00pm Thursday 5th October, 2023**

This specification was created for the use of Adnaan BUKSH (s4743556) only.  
Do not share this document. Sharing this document may result in a misconduct penalty.

### Introduction

The goal of this assignment is to demonstrate your skills and ability in fundamental process management and communication concepts, and to further develop your C programming skills with a moderately complex program.

You are to create a program called `testuqwordladder` that creates and manages communicating collections of processes that test a `uqwordladder` program (from assignment one) according to a job specification file that lists tests to be run. For various test cases, your program will, if required, generate the expected output from a known good program (`good-uqwordladder`) and then run the tests with a sample program (the program under test) to compare their standard outputs, standard errors and exit statuses and report the results. The assignment will also test your ability to code to a programming style guide and to use a revision control system appropriately.

CSSE7231 students will be required to implement additional functionality for full marks.

### Student Conduct

This section is unchanged from assignment one – but you should remind yourself of the referencing requirements.

**This is an individual assignment.** You should feel free to discuss **general** aspects of C programming and the assignment specification with fellow students, including on the discussion forum. In general, questions like “How should the program behave if {this happens}?” would be safe, if they are seeking clarification on the specification.

You must not actively help (or seek help from) other students or other people with the actual design, structure and/or coding of your assignment solution. It is **cheating to look at another student’s assignment code** and it is **cheating to allow your code to be seen or shared in printed or electronic form by others**. All submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct actions will be initiated against you, and those you cheated with. That’s right, if you share your code with a friend, even inadvertently, then **both of you are in trouble**. Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository until at least three months after the result release date for the course (or check with the course coordinator if you wish to post it sooner). Do not allow others to access your computer – you must keep your code secure. Never leave your work unattended.

You must follow the following code referencing rules for **all code committed to your SVN repository** (not just the version that you submit):

Code Origin	Usage/Referencing
Code provided to you in writing <b>this semester</b> by CSSE2310/7231 teaching staff (e.g. code hosted on Blackboard, found in <code>/local/courses/csse2310/resources</code> on <code>moos</code> , posted on the discussion forum by teaching staff, provided in Ed Lessons, or shown in class).	May be used freely without reference. (You <u>must</u> be able to point to the source if queried about it – so you may find it easier to reference the code.)
Code you have personally written this semester for CSSE2310/7231 (e.g. code written for A1 reused in A3) – provided you have not shared or published it.	May be used freely without reference. (This assumes that no reference was required for the original use.)

Code Origin	Usage/Referencing
Code examples found in man pages on <code>mooss</code> .	May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code (in the required format – see the style guide). If such code is used without appropriate referencing then this will be considered misconduct.
Code you have personally written in a previous enrolment in this course or in another ITEE course and where that code has <u>not</u> been shared or published.	
Code (in any programming language) that you have learned from or that you have taken inspiration from but have not copied <sup>1</sup> .	
Code written by, modified by, or obtained from, or code based on code written by, modified by, or obtained from, a code generation tool (including any artificial intelligence tool) that you personally have interacted with, without the assistance of another person.	May be used provided you understand that code AND the source of the code is referenced in a comment adjacent to that code (in the required format) AND an ASCII text file (named <code>toolHistory.txt</code> ) is included in your repository and with your submission that describes in detail how the tool was used. The file must be committed to the repository at the same time as any code derived from such a tool. If such code is used without appropriate referencing and without inclusion of the <code>toolHistory.txt</code> file then this will be considered misconduct.
Other code – includes (but may not be limited to): code provided by teaching staff only in a previous offering of this course (e.g. previous assignment one solution); code from public or private repositories; code from websites or based on code from websites; code from textbooks; any code written or partially written or provided by or written with the assistance of someone else; and any code you have written that is available to other students.	May <b>not</b> be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken.

**You must not share this assignment specification** with any person, organisation, website, etc. Uploading or otherwise providing the assignment specification or part of it to a third party including online tutorial and contract cheating websites is considered misconduct. The university is aware of many of these sites and many cooperate with us in misconduct investigations. You are permitted to post small extracts of this document to the course Ed Discussion forum for the purposes of seeking or providing clarification on this specification.

**The teaching staff will conduct interviews with a subset of students about their submissions**, for the purposes of establishing genuine authorship. If you write your own code, you have nothing to fear from this process. If you legitimately use code from other sources (following the usage/referencing requirements in the table above) then you are expected to understand that code. If you are not able to adequately explain the design of your solution and/or adequately explain your submitted code (and/or earlier versions in your repository) and/or be able to make simple modifications to it as requested at the interview, then your assignment mark will be scaled down based on the level of understanding you are able to demonstrate and/or your submission may be subject to a misconduct investigation where your interview responses form part of the evidence. Failure to attend a scheduled interview will result in zero marks for the assignment. The use of referenced code in your submission (particularly from artificial intelligence tools) is likely to increase the probability of your selection for an interview.

In short – **Don't risk it!** If you're having trouble, seek help early from a member of the teaching staff. Don't be tempted to copy another student's code or to use an online cheating service. Don't help another CSSE2310/7231 student with their code no matter how desperate they may be and no matter how close your relationship. You should read and understand the statements on student misconduct in the course profile and on the school website: <https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct>.

## Specification – `testuqwordladder`

`testuqwordladder` reads a set of test jobs from a file whose name is provided as a command line argument. (The format of this file is specified below.) `testuqwordladder` will, if required, generate the expected output (standard output, standard error and exit status) of a known-good implementation (`good-uqwordladder`, found in the user's `PATH`<sup>2</sup>) and save these to files in a specified directory. Generation of these files will be required if

<sup>1</sup>Code from elsewhere must not be directly copied or just converted from one programming language to another.

<sup>2</sup>Note that the version of `good-uqwordladder` provided for this assignment will be common to all students so its command line options and output messages may vary somewhat from your assignment one version.

any of them are “out of date” with respect to the job specification file, i.e. the job specification file has been modified more recently than one or more of the test output files, or if regeneration of all output files has been requested (via a command line argument).

After expected outputs have been generated (if necessary) then **testuqwordladder** will run each of the test jobs using the program-under-test (named on the **testuqwordladder** command line, e.g. your solution to assignment one) and compare its outputs (**stdout** and **stderr**) with the expected outputs and its exit status with the expected exit status.

The comparison of the outputs must use the **cmp** program (found in the user’s **PATH**) with input piped from the program-under-test.

Figure 1 illustrates the processes and pipes to be created by **testuqwordladder** for each test job (after the expected outputs have been created if necessary). Three processes will be created (labelled A, B and C in Figure 1): an instance of the program being tested (A), and two instances of **cmp** – one for comparing the standard outputs of the two programs (B) and one for comparing the standard errors (C). This will require the creation of two pipes (labelled 1 and 2 in Figure 1). The standard input for the program being tested will come from a file whose name is specified in the job specification file – this contains sample user input for that test.

Full details of the required behaviour are provided below.

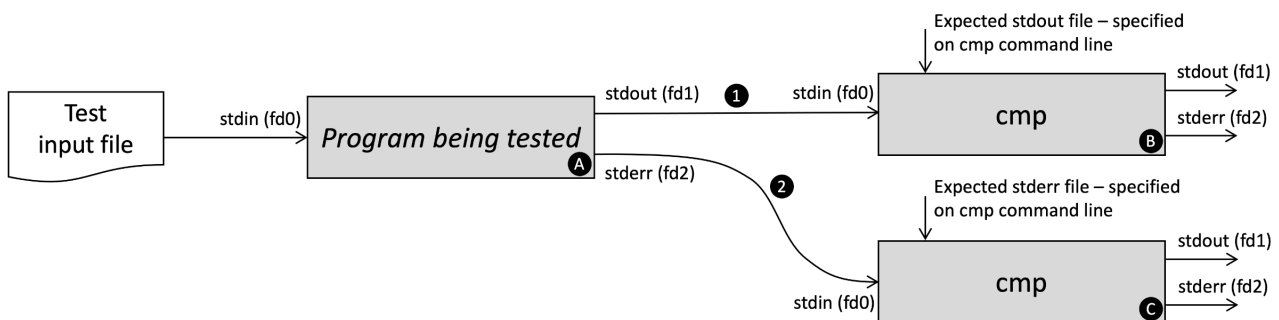


Figure 1: Processes (A to C) and pipes (1 to 2) to be created by **testuqwordladder** for each test job.

## Command Line Arguments

**testuqwordladder** has two mandatory arguments – the name of the job specification file and the name of the program being tested. There are three optional arguments which may appear in any order (but must appear before the mandatory arguments).

Usage of the program and the meaning of the arguments are as follows:

```
./testuqwordladder [--diffshow N] [--jobdir dir] [--regenerate] jobfile program
```

- **--diffshow** – this argument is for CSSE7231 functionality only (see page 9) – CSSE2310 students do not need to check for or accept this argument – it will not be used in any tests for CSSE2310 students though it must still be output in the usage message – see below.
- **--jobdir** – if present, the associated value argument (*dir*) specifies the name of the directory where the expected output files are to be saved. If not present, the directory **./tmp** is to be used.
- **--regenerate** – if present, the expected output files must all be (re)generated, whether they are out of date with respect to the job specification file or not. If not present, the expected output files are only (re)generated if they don’t exist or are out of date with respect to the job specification file.
- **jobfile** – must be present and is the name of the file containing details of the tests to be run (referred to in the remainder of this document as the *job specification file*). More details on the format of this file are given below. The argument may not begin with **--**. (If a file’s name begins with **--** then the user must give the path to the file, e.g. **./--jobfile**, to avoid a usage error.)
- **program** – must be present (after the job specification file name) and is the name of the program being tested against **good-uqwordladder**. The name can be a relative or absolute pathname or otherwise (if it doesn’t contain a slash **/**) is expected to be found on the user’s **PATH**. The argument may not begin with **--**. (If a program’s name begins with **--** then the user must give the path to the program, e.g. **./--prog**, to avoid a usage error.)

Prior to doing anything else, your `testuqwordladder` program must check the validity of the command line arguments. If the program receives an invalid command line then it must print the message:

Usage: `testuqwordladder [--diffshow N] [--jobdir dir] [--regenerate] jobfile program`

to standard error (with a following newline) and exit with an exit status of 3.

Invalid command lines include (but may not be limited to) any of the following:

- arguments that begin with `--` that are not either `--diffshow` or `--jobdir` or `--regenerate`.<sup>3</sup>
- any of the `--diffshow`, `--jobdir` or `--regenerate` option arguments is specified more than once on the command line. (Note that it is not a usage error if the value argument associated with the `--diffshow` or `--jobdir` option arguments is one of these strings.)
- an argument beginning with `--` appears after the *jobfile* or *program* arguments
- the *program* and/or the *jobfile* arguments are not specified on the command line
- an unexpected argument is present

Checking whether the program name and/or the job specification file name are valid is not part of usage checking.

## Job Specification File

If the command line arguments are valid then `testuqwordladder` reads the job specification file listed on the command line (the *jobfile* argument). The whole file must be read and checked prior to any expected output files being generated or any test jobs being run.

If `testuqwordladder` is unable to open the job specification file for reading then it must print the following message to `stderr` (with a following newline) and exit with an exit status of 6:

`testuqwordladder: Unable to open file "filename"`


where *filename* is replaced by the name of the file (as given on the command line). (The double quotes around the filename must be present in the output message.)




## File Format

The job specification file provided to `testuqwordladder` is a text file, with one line per test job. The file may contain any number of lines.

If `testuqwordladder` is able to open the job specification file for reading, then it should read and check all of the test jobs in the file before generating any expected output files or starting any tests.

Lines in the job specification file beginning with the `#` (hash) character are comments<sup>4</sup>, and are to be ignored by `testuqwordladder`. Similarly, empty lines (i.e. with no characters before the newline) are to be ignored.

All other lines are to be interpreted as job specifications, split over 2 or more separate fields delimited by the tab character (`'\t'` in C) as follows. The tab character is illustrated as  below. Square brackets (`[ ]`) indicate optional fields.

`test-idinput-file-name[arg1[arg2 ...]]`

(Note that there is a extension to this format for CSSE7231 students – see page 9.)

The `test-id` field is a non-empty string that uniquely identifies the test and that contains no tab characters or forward slashes (`'/'`).

The `input-file-name` field is the name of a file to be used as the standard input to the the test program and to `good-uqwordladder` for that test. It may be an absolute pathname or relative to the current directory. It must not be an empty field, and the file name can not contain tab characters. (A tab character will be interpreted as the end of the file name and the start of the next field.)

The third and subsequent fields (if present) are command line arguments to be provided to the test program and to `good-uqwordladder` when the test is run. These fields (if present) may be empty. Any number of arguments may be specified. Arguments may not contain tab characters. (Tab characters are interpreted as separators between arguments.)

Note that individual job specifications are independent – a given test job will always be run in the same way independent of any other test jobs described in the file.

<sup>3</sup>To allow for `getopt()` implementations of argument parsing (not required) we will not run any tests using the argument `--`

<sup>4</sup>“beginning with” means the `#` character is in the leftmost position on the line – there are no preceding spaces or other characters

The tab character has special meaning in job specification files and will always be treated as a separator between fields. It is not possible to specify a tab character as part of a test id, file name or an argument. See the `split_string()` function described on page 11 for an easy way to split the tab-delimited job specifications.

## Checking the Jobs

Each job listed in the job specification file must be checked in turn. Checks take place in the order described below and if `testuqwordladder` must exit due to an error on a line then no further lines are checked. Note that for the error messages described in this section:

- Messages are printed to `stderr` and are followed by a newline.
- Line numbers in the job specification file are counted from 1 and include any comment or blank lines.
- Double quotes around file names in the error messages must be present.

If a job line is not syntactically valid then `testuqwordladder` must print the following message and exit with an exit status of 19:

```
testuqwordladder: Syntax error on line N of job spec file "filename"
```

where *N* is replaced by the line number in the job specification file, and *filename* is replaced by the name of the job specification file (as given on the command line).

A syntactically valid line will:

- contain at least one tab character,
- have at least one character before the first tab character (i.e. a non-empty test ID),
- have no forward slash ('/') characters before the first tab character (i.e. test IDs can't contain '/'), and
- have at least one non-tab character after the first tab character (i.e. a non-empty input file name).

If the line is syntactically valid but the test ID is the same as a test ID from a previous line then `testuqwordladder` must print the following message and exit with an exit status of 1:

```
testuqwordladder: Duplicate Job ID on line N of job file "filename"
```

where *N* is replaced by the line number in the job specification file (where the second use of the test ID is found), and *filename* is replaced by the name of the job specification file.

If the line is syntactically valid and the test ID is not duplicated but the input file specified on the job line is unable to be opened for reading, then `testuqwordladder` must print the following message and exit with an exit status of 20:

```
testuqwordladder: Unable to open file "inputfile" specified on line N of file "filename"
```

where *inputfile* is replaced by the name of the file specified on the job line, *N* is replaced by the line number in the job specification file, and *filename* is replaced by the name of the job specification file (as given on the command line).

If these checks are passed (the line is syntactically valid, the test ID is not a duplicate, and the input file can be opened for reading) then `testuqwordladder` can move on to the next line in the job specification file.

If `testuqwordladder` reaches the end of the file and doesn't find any job specifications (i.e. the file is empty or only contains blank lines and/or comments) then `testuqwordladder` must print the following message and exit with an exit status of 4:

```
testuqwordladder: Job spec file "filename" is empty
```

where *filename* is replaced by the name of the job specification file (as given on the command line).

The following are examples of valid jobfiles<sup>5</sup> containing explanatory comments. Tab characters are illustrated with the `→` symbol.

```
1 # Run test with no command line arguments and an immediate EOF on stdin
2 1.1→→/dev/null
3 # Run test with an empty string as the only command line argument and immediate EOF on stdin
4 1.2→→/dev/null→→

1 # Run two tests - the first specifies a starter word, the second specifies just a length
2 1.1→→testfiles/cab-start→→--start→cab
3 test2→input-file→→--len→5
```

<sup>5</sup>This assumes that all of the listed input files are valid readable files.



```

1 # Run four tests - with various combinations of arguments
2 ids can contain spaces → testfiles/1.in → --start → head → --end → tail
3 2.0 → testfiles/2.in → --start → head → --length → 5
4 3.1 → file name with spaces → --dictionary → /usr/share/dict/words → --start → head
5
6 # Blank line on the line above will be ignored
7 test4 → 4.in → --dictionary → testfiles/common-words → --end → tail

```

## Generating Expected Outputs

If all of the test jobs in the job specification file are valid then `testuqwordladder` must (re)generate the expected outputs of each test, if necessary.

First, `testuqwordladder` must ensure that a test directory is created (or already exists) that will hold the expected outputs. The directory name is given by the value associated with the `--jobdir` command line argument, or, if not present, then `./tmp` is to be used. If the given directory path doesn't already exist, and it is not possible to create the given directory, ~~or it doesn't already exist~~, then `testuqwordladder` must print the following message to `stderr` (with a following newline) and exit with an exit status of 16:

```
testuqwordladder: Can't create directory named "dirname"
```

where *dirname* is replaced by the name of the directory – as given on the command line, or `./tmp` if the `--jobdir` argument is not given on the command line. If it doesn't already exist, the directory must be created with at least `rxw` permissions for the owner. Note that it is sufficient to check that the directory path already exists; it is not necessary to check that an existing path with that name is a directory.

If the given test directory can be created (or already exists) then `testuqwordladder` must iterate over each test job from the job specification file. Each test job will need three expected output files in the test directory. The three files are

- `testid.stdout` – containing the expected standard output
- `testid.stderr` – containing the expected standard error
- `testid.exitstatus` – containing the expected exit status (an integer printed as ASCII text, followed by a newline).

where `testid` is replaced by the test ID for that test job (i.e. all the characters before the first tab character on the relevant line in the job specification file).

If any of the expected output files is missing, or if any of the files have a modification time before the modification time of the job specification file, or if `--regenerate` is specified on the `testuqwordladder` command line, then `testuqwordladder` must print the following to `stdout` (with a following newline) and the expected output files for that test must all be (re)generated:

```
Regenerating expected output for test testid
```

where `testid` is replaced by the test ID.

Regeneration of the expected output files involves running `good-uqwordladder` with the command line arguments specified in the test job specification and with standard input coming from the input file specified in the test job specification. The standard output that results from this execution must be saved to the `testid.stdout` file in the test directory; the standard error must be saved to the `testid.stderr` files in the test directory; and the exit status must be saved as a text value with a following newline to the `testid.exitstatus` file. Note that `good-uqwordladder` must be run as a direct child of `testuqwordladder` (i.e. not a grand child).

As an example, if the job specification file contains a line like this (where `→` indicates a tab character):

```
1 1.1 → /dev/null → --start → head → --end → tail
```

then the equivalent of the following shell commands will be run to (re)generate the expected outputs:

```

1 good-uqwordladder --start head --end tail < /dev/null > testdir/1.1.stdout \
2   2> testdir/1.1.stderr
3 echo $? > testdir/1.1.exitstatus

```

where `testdir` is replaced by the name of the test directory.

If it is not possible to create one of the expected output files then `testuqwordladder` must print the following message to `stderr` (with a following newline) and exit with an exit status of 15:

```
testuqwordladder: Unable to open file "filename" for writing
```

where *filename* is replaced by the name of the file that could not be created, in the form *testdir/testid.type* where *testdir* is replaced by the name of the test directory, *testid* is replaced by the test ID, and *type* is replaced by one of `stdout`, `stderr`, or `exitstatus`. The ability to open the `.stdout` file for writing must be checked before the `.stderr` file which must be checked before the `.exitstatus` file.

Additional files in an existing test directory (i.e. files not related to the tests in a given job specification file) should be left untouched – there is no need to delete them.

## Running Test Jobs and Reporting Results

If the expected outputs of all of the test jobs are able to be created (or already existed and didn't need to be recreated), then `testuqwordladder` should move on to running the tests.

For each job in the job specification file (in the order specified in that file), `testuqwordladder` must:

- Start the job (as described below in *Running a Test Job*)
- Allow up to 1.5 seconds for the job to finish and terminate the processes that make up that job (with `SIGKILL`) if they are still running after 1.5 seconds
- Report the result of the job (as described below in *Reporting the Result of a Test Job*). Note that if a process is killed then it will not have exited normally and the test will be reported as failing.

When one job is complete, the next job can be started.

Allowing up to 1.5 seconds for the job to finish can most simply be implemented by sleeping for 1.5 seconds and then sending the `SIGKILL` signal to all processes that make up the job. (It is likely they are already dead, but `testuqwordladder` does not need to check this prior to attempting to send a signal.)

To allow tests to run quicker, `testuqwordladder` can implement more advanced timeout functionality. Rather than sleeping for 1.5 seconds after starting each test, your program can start the next test immediately after the previous one finishes, provided no test runs for more than 1.5 seconds, i.e. `SIGKILL` is still sent to the child processes after 1.5 seconds if the test is still running.

### Running a Test Job

Just before running a test job, `testuqwordladder` must output and flush the following to standard output (followed by a newline):

```
Running test: testid
```

where *testid* is replaced by the test ID.

For each individual test job, `testuqwordladder` must create two pipes and three child processes as shown in Figure 1. The three processes must all be direct children of `testuqwordladder` (i.e. not grandchildren). The processes are:

- an instance of the program being tested (as specified on the `testuqwordladder` command line). Standard input for this process must come from the input file specified in the test job. Standard output and standard error must be sent to standard input of the other two processes (see below).
- an instance of `cmp` which compares standard outputs. This `cmp` process must be run with one command line argument – the name of the file containing the expected standard output (which will be of the form *testdir/testid.stdout*). The standard input of this `cmp` process will be the standard output of the program being tested. `cmp` must be found in the user's `PATH` – do not assume a particular location for `cmp`.
- an instance of `cmp` which compares standard errors. This `cmp` process must be run with one command line argument – the name of the file containing the expected standard error (which will be of the form *testdir/testid.stderr*). The standard input of this `cmp` process will be the standard error of the program being tested. `cmp` must be found in the user's `PATH` – do not assume a particular location for `cmp`.

The standard output and standard error of the `cmp` processes must be redirected to `/dev/null`.

## Reporting the Result of a Test Job

`cmp` will exit normally with an exit status of 0 if the data received on standard input matches the contents of the file whose name is given on the command line, otherwise it will exit with a non-zero exit status. See the `cmp` man page for more details.

If any of the programs were unable to be executed then `testuqwordladder` must print the following to standard output (followed by a newline):

Unable to execute test job *testid*

where *testid* is replaced by the test ID. This counts as a test failure. The message must be printed at most once per test, even if multiple programs were unable to be executed. No other messages are to be output about this test. (If one test job fails then all test jobs are likely to fail. A possible scenario is where one of the programs is not found in the user's PATH. Your `testuqwordladder` program must not assume that future executions will fail – all tests must be attempted, even if the same failure occurs.)

If the three processes run, then for one individual test `testuqwordladder` must report the following – in this given order. (In all of the messages below, *testid* is replaced by the test ID. All messages are sent to `testuqwordladder`'s standard output and are terminated by a single newline, and must be flushed at the time of printing.)

- If the standard outputs of the two programs match (as determined by the exit status of the relevant `cmp` instance), then `testuqwordladder` must print the following:  
Job *testid*: Stdout matches  
If the standard outputs do not match, then `testuqwordladder` must print the following:  
Job *testid*: Stdout differs
- If the standard errors of the two programs match (as determined by the exit status of the relevant `cmp` instance), then `testuqwordladder` must print the following:  
Job *testid*: Stderr matches  
If the standard errors do not match, then `testuqwordladder` must print the following:  
Job *testid*: Stderr differs
- If the program under test exits normally and the exit status of the program under test matches the expected exit status (found in the relevant `.exitstatus` file), then `testuqwordladder` must print the following:  
Job *testid*: Exit status matches  
otherwise it must print:  
Job *testid*: Exit status differs

## Reporting the Overall Result

A test passes if the standard output, standard error and exit statuses of the program being tested match the expected outputs in the relevant files generated earlier.

When the running of test jobs has finished and at least one test has been completed, then `testuqwordladder` must output the following message to `stdout` (followed by a newline):

`testuqwordladder: M out of N tests passed`

where *M* is replaced by the number of tests that passed, and *N* is replaced by the number of tests that have been completed (which may be fewer than the number of the tests in the job specification file if the tests are interrupted – see below).

If at least one test has been completed and all tests that have been run have passed then `testuqwordladder` must exit with exit status 0 (indicating success). Otherwise, `testuqwordladder` must exit with exit status 12.

## Interrupting the Tests

If `testuqwordladder` receives a SIGINT (as usually sent by pressing Ctrl-C) then it should abort any test in progress (by killing and reaping the processes), not commence any more test jobs, and report the overall result (as described above) based on the tests that have been completed prior to the interruption (i.e. excluding any test in progress at the time of interruption). If no tests have been completed prior to the interruption then `testuqwordladder` must output the following message to `stdout` (followed by a newline) and exit with exit status 9:

`testuqwordladder: No tests have been finished`



Your program is permitted to use a single non-`struct` global variable to implement signal handling.

We will not test `SIGINT` being sent to your `testuqwordladder` program during the generation of expected output files – which should be quite quick. We will only test `SIGINT` being sent after test jobs are started.

## Other Functionality

`testuqwordladder` must free all dynamically allocated memory before exiting. (This requirement does not apply to child processes of `testuqwordladder`.)

Child processes created by `testuqwordladder` must not inherit any unnecessary open file descriptors opened by `testuqwordladder`. (File descriptors inherited from `testuqwordladder`'s parent must remain open.)

`testuqwordladder` is not to leave behind any orphan processes (i.e. when `testuqwordladder` exits then none of its children must still be running). `testuqwordladder` is also not to leave behind any zombie processes – all child processes from one test must be reaped before the next test starts.

`testuqwordladder` should not busy wait, i.e. it should not repeatedly check for something (e.g. process termination) in a loop. This means that use of the `WNOHANG` option when waiting is not permitted.

## Assumptions

We won't run any tests that violate the assumptions below and your program can behave in any way that you like (including crashing) if one of these assumptions turns out not to be true. You can assume the following:

- `good-uqwordladder` will always be found in the user's `PATH` and execution will not fail.
- `good-uqwordladder` will always run in a reasonable time (i.e. there is no need for a timeout on `good-uqwordladder` when (re)generating expected output files).
- `good-uqwordladder` will always exit normally, i.e. will not exit due to receiving a signal.
- `fork()` always succeeds (assuming your program calls `fork()` a reasonable number of times).
- `malloc()` always succeeds (assuming your program is using a reasonable amount of memory).
- `pipe()` always succeeds (assuming your program is using a reasonable number of file descriptors).
- If a job input file or job specification file exists and is readable at some point in `testuqwordladder`'s execution then it will continue to exist and be readable during that execution (assuming your program does not change that).

## CSSSE7231 Functionality

For the purpose of this competition, CSSSE7231 students are expected to implement additional functionality for `testuqwordladder`.

### Setting Environment Variable for Tests (CSSSE7231 only)

CSSSE7231 students must support specification of environment variables that allow environment to be specified for each test. Each environment variable must take the form:

`ENV=VALUE`

and will appear in the job specification line after the input file name and before the command line argument.

For example, `testuqwordladder --input-file=job1[→ENV1=VALUE1[→ENV2=VALUE2 ...]] --command-line=arg1` are valid. Brackets `[]` indicate optional parts. As both input fields are separated by tabs, as illustrated with `testuqwordladder --input-file=job1 --ENV1=VALUE1 --ENV2=VALUE2 --command-line=arg1`, here, any fields immediately after the job input file name that contain an equals sign `=` are to be treated as environment variable values. The first (or any) found in the job input file name that does not contain an equals sign is to be treated as the first argument field (and any remaining fields are argument fields).

Any number of such test specification fields can be:

```
1 1.1 testfile --input-file=job1 --ENV1=VALUE1 --ENV2=VALUE2 --command-line=arg1
2 1.2 testfile --input-file=job1 --ENV1=VALUE1 --ENV2=VALUE2 --command-line=arg1
3 1.3 testfile --input-file=job1 --ENV1=VALUE1 --ENV2=VALUE2 --command-line=arg1
```

60  
61  
62  
63  
64  
65  
66  
67

63  
64  
65  
66  
67

## 68

469  
470  
471

872  
873  
874

875  
876  
877  
878  
879  
880  
881  
882

84

485  
486  
487

## 188

89  
90

91  
92

## 93

94

```

1 # One job, initial and target words provided, immediate EOF on stdin
2 test 1 --from head --to tail

```

In the following runs of `testuqwordladder`, assume that `./uqwordladder` is a fully functional program. Assume that before the first run the directory `./tmp` does not exist.

```

1 $ ./testuqwordladder example1 ./uqwordladder
2 Regenerating expected output for test test 1
3 Running test: test 1
4 Job test 1: Stdout matches
5 Job test 1: Stderr matches
6 Job test 1: Exit status matches
7 testuqwordladder: 1 out of 1 tests passed
8 $ ./testuqwordladder example1 ./non-existent-program
9 Running test: test 1
10 Unable to execute test job test 1
11 testuqwordladder: 0 out of 1 tests passed

```

Note that in the first run of `testuqwordladder`, because the `/tmp` directory does not exist, the expected output for the test is generated – but this does not happen in the second run of `testuqwordladder` because these files already exist and are not out of date. Note also that there is up to a 1.5 second delay between lines 3 and 4, and between lines 9 and 10.

## Example Two

In the following example, assume that `./buggy-uqwordladder` works correctly except that it exits with a segmentation fault when given an empty dictionary. Assume that `testfiles/2.1.in` exists. The job specification file `example2` has the following contents:

```

1 1.1 --from head --dictionary /dev/null --to tail
2 2.1 testfiles/2.1.in --length 4 --from alphabet

```

```

1 $ ./testuqwordladder example2 ./buggy-uqwordladder
2 Regenerating expected output for test 1.1
3 Regenerating expected output for test 2.1
4 Running test: 1.1
5 Job 1.1: Stdout differs
6 Job 1.1: Stderr matches
7 Job 1.1: Exit status differs
8 Running test: 2.1
9 Job 2.1: Stdout matches
10 Job 2.1: Stderr matches
11 Job 2.1: Exit status matches
12 testuqwordladder: 1 out of 2 tests passed

```

## Provided Library: libcsse2310a3

A library has been provided to you with the following functions which your program may use. See the man pages on moss for more details on these library functions.

```
char* read_line(FILE *stream);
```

The function attempts to read a line of text from the specified stream, allocating memory for it, and returning the buffer.

```
char **split_string(char* line, char delimiter);
```

This function will split a string into substrings based on a given delimiter character.

```
int compare_timespecs(struct timespec t1, struct timespec t2);
```

This function will compare two times represented in `struct timespec` types and determine which is earlier than the other.

To use the library, you will need to add `#include <csse2310a3.h>` to your code and use the compiler flag `-I/local/courses/csse2310/include` when compiling your code so that the compiler can find the include file. You will also need to link with the library containing this function. To do this, use the compiler arguments `-L/local/courses/csse2310/lib -lcsse2310a3`.

## Style

Your program must follow version 2.4.1 of the CSSE2310/CSSE7231 C programming style guide available on the course Blackboard site.

## Hints

1. Make sure you review the lecture/contact content and supplementary videos and examples and the Ed Lessons exercises from weeks 6 and 7. These provide necessary background for this assignment and there may be code snippets that you can reuse or adapt.
2. While not mandatory, the provided library functions will make your life a lot easier – use them!
3. You can examine the file descriptors associated with a process by running `ls -l /proc/PID/fd` where PID is the process ID of the process.
4. The system call `mkdir()` can be used to create a directory path – see the man page for details. (Run “`man 2 mkdir`” to see the man page.) `mkdir()` will fail with `errno` being `EEXIST` if the path already exists.
5. The system call `stat()` can be used to find out information about a file, e.g. its modification time can be found in the `st_mtim` member of the `stat` structure – see the man page for details. This `st_mtim` member is of type `struct timespec` which is defined in the `nanosleep()` man page.
6. `usleep()` or `nanosleep()` can be used to sleep for a non-integer number of seconds.
7. Execution (`exec`) failure can be detected by having the child process return an unexpected exit status to the parent (e.g. 99).
8. Consider the use of `sigtimedwait()` for `SIGCHLD` if you’re implementing the more advanced timeout functionality.
9. (CSSE7231 students) You may find the `putenv()` function useful.
10. A demo program will be provided: `demo-testuqwordladder` – you can check the expected behaviour of your `testuqwordladder` by running this program. Test scripts (`testa3.sh` and `reptesta3.sh`) will also be provided. You are strongly encouraged to use them.

## Suggested Approach

It is suggested that you write your program using the following steps. Test your program at each stage and commit to your SVN repository frequently. Note that the specification text above is the definitive description of the expected program behaviour. The list below does not cover all required functionality but will get you started.

1. Write a program to parse the expected command line arguments and handle usage errors.
2. Write code to read and parse the job specification file and save it to a data structure (and output error messages if problems are found).
3. Write code that attempts to create the test directory
4. Write code to generate the expected output files (all the time).
5. Add code to check whether the expected output files need to be regenerated
6. Write code that creates the necessary pipes and correctly starts the three processes associated with one test job (with the necessary command line arguments).

## Forbidden Functions

You must not use any of the following C statements/directives/etc. If you do so, you will get zero (0) marks for the assignment.

- `goto`
- `#pragma`
- gcc attributes

You must not use any of the following C functions. If you do so, you will get zero (0) marks for any test case that calls the function.

- `longjmp()` and equivalent functions
- `system()`
- `popen()`
- `mkfifo()` or `mkfifoat()`
- `signal()`, `sigset()` – you should use `sigaction()` instead
- `waitpid()` or `waitid()` using the `WNOHANG` option – these functions are permitted without that option
- `execlp()` or `execvp()` or related family members to execute any program other than `good-uqwordladder`, the program under test, `cmp`, `diff`, or `head`. (The last two are for CSSE7231 students.)
- Functions described in the man page as non standard, e.g. `execvpe()`, `strcasestr()`

## Submission

Your submission must include all source and any other required files (in particular you must submit a **Makefile**). Do not submit compiled files (e.g. `.o` files and compiled programs).

Your program `testuqwordladder` must build on `moss.labs.eait.uq.edu.au` and in the Gradescope test environment with the command:

```
make
```

Your program must be compiled with `gcc` with at least the following options:  
`-pedantic -Wall -std=gnu99`

You are not permitted to disable warnings or use pragmas to hide them. You may not use source files other than `.c` and `.h` files as part of the build process – such files will be removed before building your program.

If any errors result from the `make` command (i.e. no executable is created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Your program must not invoke other programs or use non-standard headers/libraries other than those explicitly described in this specification.

Your assignment submission must be committed to your subversion repository under

```
https://source.eait.uq.edu.au/svn/csse2310-sem2-sXXXXXXX/trunk/a3
```

where `sXXXXXXX` is your moss/UQ login ID. Only files at this top level will be marked so **do not put source files in subdirectories**. You may create subdirectories for other purposes (e.g. your own test files) but these will not be considered in marking – they will not be checked out of your repository.

You must ensure that all files needed to compile and use your assignment (including a `Makefile`) are committed and within the `trunk/a3` directory in your repository (and not within a subdirectory or some other part of your repository) and not just sitting in your working directory. Do not commit compiled files or binaries. You are strongly encouraged to check out a clean copy for testing purposes.

To submit your assignment, you must run the command

```
2310createzip a3
```

on `moss` and then submit the resulting zip file on Blackboard (a GradeScope submission link will be made available in the Assessment area on the CSSE2310/7231 Blackboard site)<sup>6</sup>. The zip file will be named

<sup>6</sup>You may need to use `scp` or a graphical equivalent such as WinSCP, Filezilla or Cyberduck in order to download the zip file to your local computer and then upload it to the submission site.



sXXXXXXX\_csse2310\_a3\_timestamp.zip

where sXXXXXXX is replaced by your moss/UQ login ID and *timestamp* is replaced by a timestamp indicating the time that the zip file was created.

The 2310createzip tool will check out the latest version of your assignment from the Subversion repository, ensure it builds with the command ‘make’, and if so, will create a zip file that contains those files and your Subversion commit history and a checksum of the zip file contents. You may be asked for your password as part of this process in order to check out your submission from your repository. You will be asked to confirm references in your code.

You must not create the zip file using some other mechanism and you must not modify the zip file prior to submission. If you do so, you will receive zero marks. Your submission time will be the time that the file is submitted via GradeScope on Blackboard, and **not** the time of your last repository commit nor the time of creation of your submission zip file.

We will mark your last submission, even if that is after the deadline and you made submissions before the deadline. Any submissions after the deadline<sup>7</sup> will incur a late penalty – see the CSSE2310/7231 course profile for details.

Note that Gradescope will run the test suite immediately after you submit. When complete<sup>8</sup> you will be able to see the results of the “public” tests.

## Marks

Marks will be awarded for functionality, style and svn commit history messages. Marks may be reduced if you are asked to attend an interview about your assignment and you do not attend or are unable to adequately respond to questions – see the **Student conduct** section above.

## Functionality (60 marks CSSE2310/ 70 marks CSSE7231)

Provided your code compiles (see above) and does not use any prohibited statements/functions (see above), and your zip file has been generated correctly and has not been modified prior to submission, then you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never create a child process then we can not test pipe communication between the processes of that job, or your ability to send it a termination signal. Memory-freeing tests require correct functionality also – a program that frees allocated memory but doesn’t implement the required functionality can’t earn marks for this criteria. This is not a complete list of all dependencies, other dependencies may also exist. Your tests must run in a reasonable time frame, which could be as short as a few seconds for usage checking to many tens of seconds when **valgrind** is used to test for memory leaks. If your program takes too long to respond, then it will be terminated and you will earn no marks for the functionality associated with that test. The markers will make no alterations to your code (other than to remove code without academic merit).

Functionality marks (out of 60) will be assigned for **testuqwordladder** in the following categories (CSSE2310 and CSSE7231):

1. **testuqwordladder** correctly handles invalid command lines and the inability to open job specification files (5 marks)
2. **testuqwordladder** correctly handles errors in job specification files and job specification files containing no jobs (5 marks)
3. **testuqwordladder** correctly creates test directory (if it doesn’t exist) and handles the inability to create a test directory (4 marks)
4. **testuqwordladder** correctly runs **good-uqwordladder** and creates expected output files when none initially exist (includes handling the inability to create such files) (6 marks)
5. **testuqwordladder** correctly (re)generates (or not) expected output files when at least one already exists (includes handling the inability to create such files) (6 marks)

<sup>7</sup>or your extended deadline if you are granted an extension.

<sup>8</sup>Gradescope marking may take only a few minutes or more than 30 minutes depending on the functionality/efficiency of your code.

6. **testuqwordladder** is able to run the required three processes for one test job (described in a single line job specification file). File descriptors neednot be setup correctly. Multiple valid single-job files and command line arguments will be tested. (4 marks)
7. **testuqwordladder** constructs the necessary pipes and **stdin/stdout/stderr** redirections for the three processes in one test job (described in a single line job specification file). Multiple valid single-job files and command line arguments will be tested. (4 marks)
8. **testuqwordladder** correctly implements testing of one job (including reporting results and handling job specification files with comments and blank lines) (6 marks)
9. **testuqwordladder** correctly implements testing of multiple jobs (including reporting results and handling job specification files with comments and blank lines) (5 marks)
10. **testuqwordladder** correctly implements test interruption (signal handling) (5 marks)
11. **testuqwordladder** correctly closes all unnecessary file descriptors in child processes (4 marks)
12. **testuqwordladder** frees all allocated memory prior to exit (original process, not children) (4 marks)
13. **testuqwordladder** runs tests as quickly as possible (1.5 second timeouts are only necessary for tests that run that long) (2 marks)

Some functionality may be assessed in multiple categories, e.g. the ability to run multiple test jobs must be working to fully test more advanced functionality such as checking for unnecessary file descriptors being closed and for memory to be freed.

Functionality marks (out of 10) will be assigned in the following categories (CSSE7231 only):

14. **testuqwordladder** correctly handles setting environment variables when (re)generating expected output files (2 marks)
15. **testuqwordladder** correctly handles setting environment variables when running tests (3 marks)
16. **testuqwordladder** correctly handles **--diffshow** argument (5 marks)

## Style Marking

Style marking is based on the number of style guide violations, i.e. the number of violations of version 2.4.1 of the CSSE2310/CSSE7231 C Programming Style Guide (found on Blackboard). Style marks will be made up of two components – automated style marks and human style marks. These are detailed below.

You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the **indent(1)** tool. Your style marks can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

You are encouraged to use the **style.sh** tool installed on **mooss** to style check your code before submission. This does not check all style requirements, but it will determine your automated style mark (see below). Other elements of the style guide are checked by humans.

All **.c** and **.h** files in your submission will be subject to style marking. This applies whether they are compiled/linked into your executable or not<sup>9</sup>.

## Automated Style Marking (5 marks)

Automated style marks will be calculated over all of your **.c** and **.h** files as follows. If any of your submitted **.c** and/or **.h** files are unable to be compiled by themselves then your automated style mark will be zero (0). (Automated style marking can only be undertaken on code that compiles. The provided **style.sh** script checks this for you.)

If your code does compile then your automated style mark will be determined as follows: Let

- $W$  be the total number of distinct compilation warnings recorded when your **.c** files are individually built (using the correct compiler arguments)

---

<sup>9</sup>Make sure you remove any unneeded files from your repository, or they will be subject to style marking.

- $A$  be the total number of style violations detected by `style.sh` when it is run over each of your `.c` and `.h` files individually<sup>10</sup>.

Your automated style mark  $S$  will be

$$S = 5 - (W + A)$$

If  $W + A \geq 5$  then  $S$  will be zero (0) – no negative marks will be awarded. Note that in some cases `style.sh` may erroneously report style violations when correct style has been followed. If you believe that you have been penalised incorrectly then please bring this to the attention of the course coordinator and your mark can be updated if this is the case. Note also that when `style.sh` is run for marking purposes it may detect style errors not picked up when you run `style.sh` on `moss`. This will not be considered a marking error – it is your responsibility to ensure that all of your code follows the style guide, even if styling errors are not detected in some runs of `style.sh`. You can check the result of Gradescope style marking soon after your Gradescope submission – when its test suite completes running.

## Human Style Marking (5 marks)

The human style mark (out of 5 marks) will be based on the criteria/standards below for “comments”, “naming” and “other”. The meanings of words like *appropriate* and *required* are determined by the requirements in the style guide. Note that functions longer than 50 lines will be penalised in the automated style marking. Functions that are also longer than 100 lines will be further penalised here.

### Comments (2.5 marks)

Mark	Description
0	The majority (50%+) of comments present are inappropriate OR there are many required comments missing
0.5	The majority of comments present are appropriate AND the majority of required comments are present
1.0	The vast majority (80%+) of comments present are appropriate AND there are at most a few missing comments
1.5	All or almost all comments present are appropriate AND there are at most a few missing comments
2.0	Almost all comments present are appropriate AND there are no missing comments
2.5	All comments present are appropriate AND there are no missing comments

### Naming (1 mark)

Mark	Description
0	At least a few names used are inappropriate
0.5	Almost all names used are appropriate
1.0	All names used are appropriate

### Other (1.5 marks)

Mark	Description
0	One or more functions is longer than 100 lines of code OR there is more than one global/static variable present inappropriately OR there is a global struct variable present inappropriately OR there are more than a few instances of poor modularity (e.g. repeated code)
0.5	All functions are 100 lines or shorter AND there is at most one inappropriate non-struct global/static variable AND there are at most a few instances of poor modularity
1.0	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no or very limited use of magic numbers AND there is at most one instance or poor modularity
1.5	All functions are 100 lines or shorter AND there are no instances of inappropriate global/static variables AND there is no use of magic numbers AND there are no instances of poor modularity

## SVN Commit History Marking (5 marks)

Markers will review your SVN commit history for your assignment up to your submission time. This element will be graded according to the following principles:

<sup>10</sup>Every `.h` file in your submission must make sense without reference to any other files, e.g., it must `#include` any `.h` files that contain declarations or definitions used in that `.h` file.

- Appropriate use and frequency of commits (e.g. a single monolithic commit of your entire assignment will yield a score of zero for this section)
- Appropriate use of log messages to capture the changes represented by each commit. (Meaningful messages explain briefly what has changed in the commit (e.g. in terms of functionality) and/or why the change has been made and will be usually be more detailed for significant changes.)

The standards expected are outlined in the following rubric:

Mark (out of 5)	Description
0	Minimal commit history – only one or two commits OR all commit messages are meaningless.
1	Some progressive development evident (three or more commits) AND at least one commit message is meaningful.
2	Progressive development is evident (multiple commits) AND at least half the commit messages are meaningful.
3	Multiple commits that show progressive development of ALL functionality (e.g. no large commits with multiple features in them) AND at least half the commit messages are meaningful.
4	Multiple commits that show progressive development of ALL functionality AND meaningful messages for all but one or two of the commits.
5	Multiple commits that show progressive development of ALL functionality AND meaningful messages for ALL commits.

## Total Mark

Let

- $F$  be the functionality mark for your assignment (out of 60 for CSSE2310 students or out of 70 for CSSE7231 students).
- $S$  be the automated style mark for your assignment (out of 5).
- $H$  be the human style mark for your assignment (out of 5).
- $C$  be the SVN commit history mark (out of 5).
- $V$  is the scaling factor (0 to 1) determined after interview(s) (if applicable – see the Student Conduct section above) – or 0 if you fail to attend a scheduled interview without having evidence of exceptional circumstances impacting your ability to attend.

Your total mark for the assignment will be:

$$M = (F + \min\{F, S + H\} + \min\{F, C\}) \times V$$

out of 75 (for CSSE2310 students) or out of 85 (for CSSE7231 students).

In other words, you can't get more marks for style or SVN commit history than you do for functionality. Pretty code that doesn't work will not be rewarded!

## Late Penalties

Late penalties will apply as outlined in the course profile.

## Specification Updates

Any errors or omissions discovered in the assignment specification will be added here, and new versions released with adequate time for students to respond prior to due date. Potential specification errors or omissions can be discussed on the discussion forum or emailed to [csse2310@uq.edu.au](mailto:csse2310@uq.edu.au).

## Version 1.1

- Fixed error in wording of when message about inability to create test directory is printed.
- Made it clear that `exec*()` family members can not be used to execute programs other than those specified.
- Added additional assumption that can be made (`pipe()` calls can be assumed to succeed).
- Added suggestion on the command to run to see the right `mkdir` man page.
- Added clarification about how `SIGINT` handling will be tested.
- Fixed minor typo.

650  
651  
652  
653  
654  
655  
656