Version 27 Jan 2020

CS4218 – Software Testing

Project Description

1. Introduction

CS4218 covers the concepts and practices of software testing and debugging. An important portion of CS4218 is the project work. Through this project students will learn and apply testing and debugging techniques followed by quality assessment activities. Teams of students will implement a shell and test its functionality using different testing techniques that will be taught during the course.

Students will learn how to professionally apply testing approaches and techniques with the state of art testing automation framework JUnit. Students will be shown good and poor styles of manual unit-test generation. Students will evaluate the quality and thoroughness of the test cases and project code using different coverage metrics. They will apply testing, debugging and other quality assurance activities in a simulated industrial setting.

2. Project Teams

Students should form teams of maximum 4 at the beginning of the semester. Once formed, teams would be final for the duration of the course.

3. Coding

All programming assignments must be completed using JAVA. External libraries/plugins should not be used. The implementation should not rely on network communication, databases. You should **not** rely on any platform-specific functionality. Use Java properties “file.separator” and “path.separator” when working with file system. Use “line.separator” property when working with newlines, since different operating system may use different separator.

Your code must conform to CS4218 code conventions. These conventions are checked automatically using PMD tool (see installation instructions in LumiNUS). Your methods should not be too long (more than 50 lines of code). You must have proper Javadoc comments for every method. Follow the Java naming convention. Use naming convention for test classes. For example, test class for the class name “Foo” should be called “FooTest”. For test method, explain the scenario that you are testing (e.g., for a test that check for negative value for method foo, use name like testFooNegativeValue).

Use the following versions of software:

- JDK 8 or higher
- IntelliJ IDEA
- JUnit 5

4. Assessment

Students can get a maximum of 40 marks for the project, divided as follows:

- 5 marks: Lab Attendance and Participation
- 2 marks: Shell and Application Implementation
- 6 marks: Unit Tests
- 3 marks: Test-driven Development
- 4 marks: Integration Tests
- 2 marks: Code Quality
- 8 marks: Hackathon Bugs
- 2 marks: Hackathon Fixes
- 8 marks: Quality Assurance (QA) Report

Please see Submission Instructions section for details.

5. Project Timeline

Week	Lab	Deadline
1&2	No Lab. Form teams, register your team, and familiarize yourself with the Project Description (current document).	
3	Project introduction, Version control systems, PMD tool	
4	Test driven development	
5	Unit testing	
6	Debugging	
Recess		
7	Integration testing Code coverage	Mon, 2 Mar, 2pm: **Milestone 1**
8	Automated testing tools	
9	Flexible – Complete development	
10	Hackathon. Write tests and help to find bugs in each other's code	Mon, 23 Mar, 2pm: **Milestone 2** Fri, 27 Mar, 2pm: **Hackathon**
11	Explanation of Hackathon results – clarification of bug/feature	Mon, 30 Mar, 2pm: **Rebuttal**
12	Quality Assurance report and its structure Fri – PH, Good Friday	
13	GUI Testing, Testing Distributed Systems	Sat, 18 Apr, 10am: **Milestone 3**

6. Background

A shell is a command interpreter. Its responsibility is to interpret commands that the user types and to run programs that the user specifies in her command lines. Figure 1 shows the relationship between the shell, the kernel, and various applications/utilities in a UNIX-like operating system:

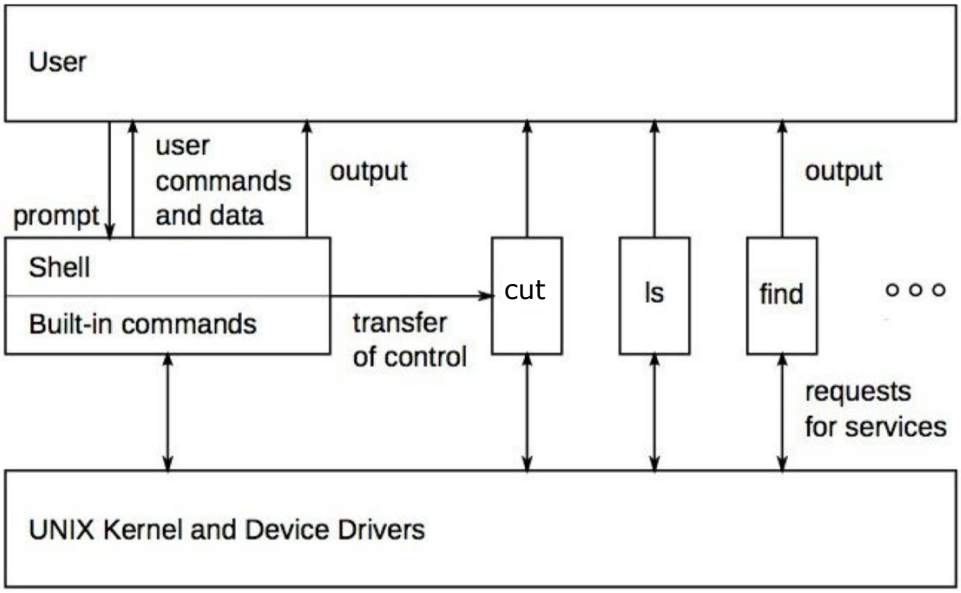

Figure 1: relationship between the shell, the kernel, and various applications

Shell can be thought of as a programming language for running applications. From the user's perspective, it performs the following loop:

1. Print prompt message.
2. Wait for user's command.
3. Parse and interpret user's command, run specified applications if any.
4. Print output.
5. Go to 2.

An application in a UNIX-like system can be *roughly considered* as a block with two inputs and three outputs, as shown in Figure 2. When an application is run, it reads an array of command line arguments and text data from its Standard Input stream (stdin). During execution, it writes output data to its Standard Output stream (stdout) and error information to its Standard Error stream (stderr). After execution, the application returns Exit Status, a number that indicates an execution error if it is non-zero.

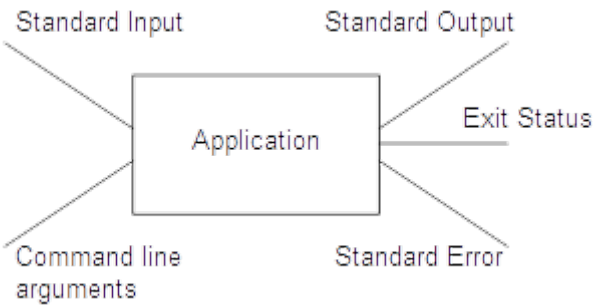

Figure 2

The important feature of shells in UNIX-like systems is the ability to compose complex commands from simple ones. For example, the following command combines the applications “paste” and “grep”:

```
paste articles/* | grep "Interesting String"
```

As shown in Figure 3, Shell expands “articles/*” into the list of all the files in the “articles” directory and passes them to “paste” as command line arguments. Then, “paste” merges the contents of all these files and passes the results to “grep” using the pipe operator “|”. “grep” finds and displays all the lines that include “Interesting String” as a substring. The connection between these two commands is made using the pipe operator “|” that connects the stdout of “paste” with the stdin of “grep”:

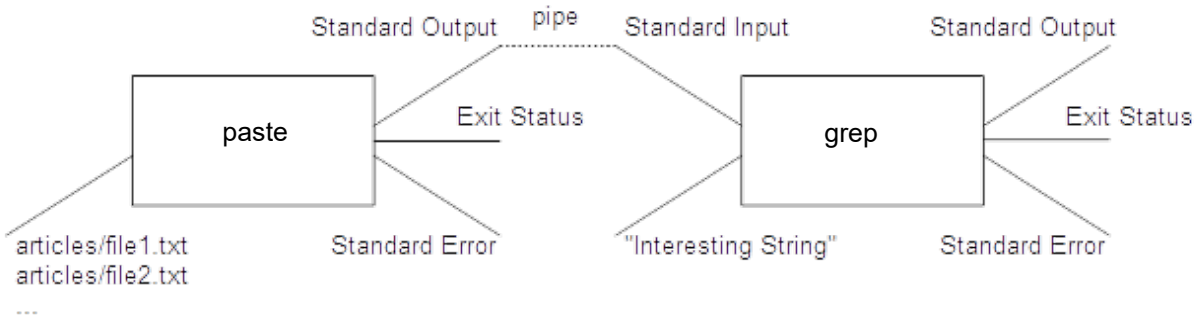

Figure 3

Further information about shells in UNIX-like systems can be found here:

- Explaining Bash syntax: http://explainshell.com/
- The Unix Shell: http://v4.software-carpentry.org/shell/index.html
- Advanced Bash-Scripting Guide: http://www.tldp.org/LDP/abs/html/
- Bash Hackers Wiki: http://wiki.bash-hackers.org

7. Requirements

The goal of the project is to implement and test a shell and a set of applications. The shell and the applications must be implemented in JAVA programming language. The required functionality is a subset (or simplification) of the functionality provided by UNIX-like systems. Particularly, the specification was designed in such a way that it maximally resembles the behaviour of Bash shell in GNU/Linux and Mac OS. However, there are several important distinctions:

1. JVM is used instead of OS Kernel/drivers to provide required services.
2. Shell and all applications are run inside the same process.
3. Applications raise exceptions instead of writing to stderr and returning non-zero exit code in case of errors, as shown in Figure 4.

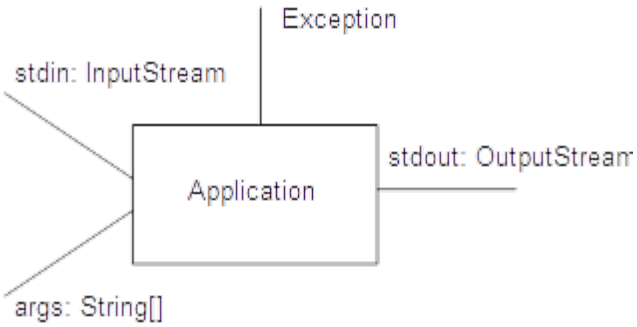

Figure 4

An application in the project implementation is a JAVA class that implements the Application interface and uses InputStream and OutputStream as stdin and stdout respectively. Main JAVA interfaces for applications are provided in a skeleton. Meanwhile, an almost complete implementation of shell is provided, so that students just need minimal work on implementation and pay more attention on the testing. On the other hand, some **bugs are injected** to the implementation of shell, and students should use testing techniques to find and fix those bugs. Specifically, students need to finish the following tasks:

- fix the existing bugs in the provided implementation of shell
- complete the implementation of shell
- implement missing applications
- test the implementation of shell and applications

Note that students need to use the **interfaces provided**, and are **not allowed** to modify their definition. The interfaces will be later used in unit testing, so it is important to maintain compatibility among teams.

However, students are allowed to modify the implementation we provide for the shell (but not the interface definitions). Note that students are expected to test and fix bugs in our skeleton code assuming they are using the implementation provided.

Since this is a software testing class, students are expected to document all testing activities and submit a report about them at the end of the project (Milestone 3). **Students are advised to check before starting their project the QAReport format published on LumiNUS as a reference.**

The required functionality is split into three groups: Basic Functionality, Extended Functionality 1, and Extended Functionality 2.

The Basic Functionality (BF) includes:

- Shell: calling applications, pipe operator, semicolon operator
- Applications: rm, echo, paste, sed, exit.

The Extended Functionality 1 (EF1) includes:

- Shell: globbing, quoting.
- Applications: diff, grep, wc, cd, cp.

The Extended Functionality 2 (EF2) includes:

- Shell: IO-redirection, command substitution.
- Applications: cut, ls, sort, find, mv.

8. Shell specification

Shell can be considered as a programming language where applications play the same role as functions in languages like C and JAVA. Shell parses user's command line to determine the applications to run and the data to pass to these applications. Our shell supports two ways to specify input data for applications: by supplying command line arguments and by redirecting output streams.

8.1. Command line parsing

User's command line can contain several subcommands. When Shell receives a command line, it does the following:

1. Parses the command line on the command level. Shell recognizes three kind of commands: call command, sequence command (using semicolon operator), pipe command (using pipe operator).

Command line uses the following grammar:

```
<command> ::= <call> |  <seq> | <pipe>
<seq>  ::= <command> ";" <command>
<pipe> ::= <call> "|" <call> | <pipe> "|" <call>
```

Note: ";" is sequence operator, "|" is pipe operator, | is logic or, [text] is used to denote optional text.

The non-terminals `<call>` and `<quoted>` are described below.

2. The recognized commands are evaluated in the proper order.

8.2. Call command

Example

```
grep "Interesting String" < text1.txt > result.txt
```

Find all the lines of the file text1.txt that contain the string "Interesting String" as a substring and save them to the file result.txt.

Syntax

Shell splits call command into arguments and redirection operators.

```
<call> ::= <non-keyword>
  [ <whitespace> ] [ <redirection> <whitespace> ]* [<argument>]
  [ <whitespace> <atom> ]* [ <whitespace> ]
<atom> ::= <redirection> | <argument>
<argument> ::= ( <quoted> | <unquoted> )+
<redirection> ::= "<" [ <whitespace> ] <argument> |
                  ">" [ <whitespace> ] <argument>
```

A non-keyword character is any character except for newlines, single quotes, double quotes, backquotes, semicolons ";" and vertical bars "|".

Whitespace is one or several tabs or spaces. An unquoted part of an argument can include any characters except for whitespace characters, quotes, newlines, semicolons ";", vertical bar "|", less than "<" and greater than ">".

Semantics

A call command is evaluated in the following order:

1. Command substitution is performed (See section 8.8 on Command Substitution). Note that command substitution can be a call command.
2. The command is split into arguments and redirection operators. The command string is split into substring corresponding to the `<argument>` non-terminal. Note that one backquoted argument can produce several arguments after command substitution. All the quotes symbols that form `<quoted>` non-terminal are removed (see section on Quoting).
3. Filenames are expanded (see section on Globbing).
4. Application name is resolved.
5. Specified application is executed.

When Shell executes an application, it performs the following steps:

1. IO-redirection, if needed (see Section on IO Redirection).
2. Running. Run the specified application (the first `<argument>` without a redirection operator), supplying given command line arguments and redirection streams.

8.3. IO Redirection

Example

```
paste < file.txt
```

Display (merge) the content of file.txt.

Syntax

```
<redirection>::= "<" [ <whitespace> ] <argument> |
                 ">" [ <whitespace> ] <argument>
```

Semantics

- If several files are specified for input redirection or output redirection, throw an exception.
- If no files are given, throw an exception.
- Open InputStream from the file for input redirection (the one following “<“ symbol).
- Open the OutputStream to the file for output redirection (the one following “>“ symbol).
- If the file specified for input redirection does not exist, throw an exception.
- If the file specified for output redirection does not exist, create it. No exception should be thrown.

8.4. Quoting

Example

To pass several arguments to an application, we can separate them by spaces:

```
echo hello world
```

In this example, “echo” gets two command line arguments: “hello” and “world”. In order to pass “hello world” as a single argument, we can surround it by quotes, so that the interpretation of the space character as a separator symbol is disabled:

```
echo “hello world”
```

In this case, “echo” receives “hello world” as a single argument (without quotes).

Our shell supports three kinds of quotes:

- single quotes (‘)
- double quotes (“)
- backquotes (`)

The first (‘) and the second ones (“) are used to disable interpretation of all or some special characters, the last one (`) is used to make command substitution. Special characters are: \t (tab), * (globbing), ‘ (single quote), “ (double quote), ` (backquote), | (pipe), < (input redirection), > (output redirection), ; (semicolon), space.

Syntax

```
<quoted>         ::= <single-quoted> | <double-quoted> |
                     <backquoted>
<single-quoted> ::= “‘” <non-newline and non-single-quote> “’”
<backquoted>     ::= “`” <non-newline and non-backquote> “`”
<double-quoted> ::= ““”(<backquoted> | <double-quote-content>)*””””
```

where <double-quote-content> can contain any character except for newlines, double quotes and backquotes.

Single quote disables the interpretation of all special symbols. For example:

```
$echo ‘Travel time Singapore -> Paris is 13h and 15`’
```

would output: `Travel time Singapore -> Paris is 13h and 15`

Double quote disables the interpretation of all special symbols, except for ` (backquote). For example, in the following command:

```
$echo “This is space:`echo “ “`.”
```

the outer “echo” receives one argument rather than two and outputs:

```
This is space: .
```

The same example, using single quote:

```
$echo ‘This is space:`echo “ “`.’
```

would output: `This is space:`echo “ “`.`

```
$echo "This is space:' '."
```

would output: `This is space:' '.`

```
$echo "'This is space `echo “ ”`'"
```

The single quote is disabled by double quote, so that the backquote will not be disabled. This command would output: `'This is space  '`

```
$echo '"This is space `echo " "`"'
```

The single quote disables double quote and backquote. It would output: `"This is space `echo " "`"`

Note that we do not use character escaping (\) in our shell.

8.5. Pipe operator

Example

```
$ paste articles/text1.txt | grep "Interesting String"
```

Find all the line of the file articles/text1.txt that contain “Interesting String” as a substring.

Syntax

```
<pipe>::= <call> "|" <call> | <pipe> "|" <call>
```

Semantics

Pipe is a left-associative operator that can be used to bind a set of call commands into a chain. Each pipe operator binds the output of the left part to the input of the right part, then evaluates these parts concurrently. If an exception occurred in any of these parts, **the execution of the other part must be terminated with a non-zero exit code.**

8.6. Globbing

Example

```
ls articles/*
```

Display the names of all the files in the articles directory.

Syntax

The symbol * (asterisk) in an unquoted part of an argument is interpreted as globbing.

Semantics

For each argument ARG in a shell command that contains unquoted * (asterisk) do the following:

1. Collect all the paths to existing files and directories such that these paths can be obtained by replacing all the unquoted asterisk symbols in ARG by some (possibly empty) sequences of non-slash characters.
2. If there are no such paths, leave ARG without changes.
3. If there are such paths, replace ARG with a list of these path separated by spaces.

Note that globbing (filenames expansion) is performed after argument splitting. However, globbing produces several command line arguments if several paths are found.

8.7. Semicolon operator

Example

```
cd articles; paste text1.txt
```

Change the current directory to articles. Display the content of the file text1.txt.

Syntax

```
<seq> ::= <command> ";" <command>
```

Semantics

Run the first command; when the first command terminates, run the second command. If an exception is thrown during the execution of the first command, the execution of the second command can continue and may return a non-zero exit code.

8.8. Command substitution

Example

```
paste `ls x*` > all.txt
```

List all file that start with x in alphabetical order and merge their content (see paste). Output of the command is redirected in “all.txt”.

Syntax

A part of a call command surrounded by backquotes (`) is interpreted as command substitution iff the backquotes are not inside single quotes (corresponds the non-terminal `<backquoted>`).

Semantics

For each part SUBCMD of a call command CALL surrounded by backquotes:

1. SUBCMD is evaluated as a separate shell command yielding the output OUT.
2. SUBCMD together with the backquotes is substituted in CALL with OUT. After substitution, symbols in OUT are interpreted the following way:
    - Whitespace characters are used during the argument splitting step. Since our shell does not support multi-line commands, newlines in OUT should be replaced with spaces.
    - Other characters (including quotes) are not interpreted during the next parsing step as special characters. For example:

    ```
    echo `echo "'quote is not interpreted as special character'"`
    ```

    would output: `'quote is not interpreted as special character'`
3. The modified CALL is evaluated.

Note that command substitution is performed after command-level parsing but before argument splitting.

9. Applications specification

Applications that require stdin stream do not read it directly (from console), but only use input streams provided by shell through redirections. If expected stdin is not provided, the application must raise an exception. If required command line arguments are not provided or arguments are wrong or inconsistent, the application should raise an exception as well.

If the applications specification is not comprehensive enough, you are supposed to further specify the requirements in your **Assumptions.pdf** file AND code comments. When in doubt, follow the UNIX shell specification of the applications.

9.1. rm

Description

The rm command attempts to remove the non-directory type files specified on the command line.

Command format

`rm [Options] FILES...`

`Options` – One or more of the following options may appear:
`-r` – Traverse recursively to delete the folder contents
`-d` – attempt to delete directory as well as all other types of file

Examples

Remove the file “1.txt” from current directory
```
$ rm 1.txt
```
Remove the folder named “test”
```
$ rm -d test
```

9.2. echo

Description

The echo command writes its arguments separated by spaces and terminates by a newline on the standard output.

Command format

`echo [ARG]`

`ARG` – list of arguments

Examples

Display A B C (separated by space characters)
```
$ echo A B C
```
Display “A*B*C” (separated by *)
```
$ echo "A*B*C"
```

9.3. paste

Description

Merge lines of files, write to standard output lines consisting of sequentially corresponding lines of each given file, separated by a TAB character.

Command format

`paste [FILE]...`
`FILE` – the name of the file or files. If not specified, use stdin.

Examples

Merge two files A.txt and B.txt (lines from the two files will be merged sequentially and separated by TAB)

```
$ paste A.txt B.txt >AB.txt
```

A.txt	B.txt	AB.txt
A	1	A 1
B	2	B 2
C	3	C 3
D	4	D 4

9.4. sed

Description

The sed command copies input file (or input stream) to stdout and performs string replacement. For each line containing a match to a specified pattern (in JAVA format), replaces the matched substring with the specified string.

Command format

`sed REPLACEMENT [FILE]`

`REPLACEMENT` – specifies replacement rule, as follows:

- `"s/regexp/replacement/"` – replace the first (in each line) substring matched by regexp with the string replacement.
- `"s/regexp/replacement/X"` – X is a number. Only replace the Xth match of the regexp.

Note that the symbols "/" used to separate regexp and replacement string can be substituted by any other symbols. For example, "s/a/b/" and "s|a|b|" are the same replacement rules. However, this separation symbol should not be used inside the regexp and the replacement string.

`FILE` – the name of the file. If not specified, use stdin.

Examples

add a leading angle bracket and space to each line (quote a message) in file.txt
`$ sed "s/^/> /" file.txt`
delete leading angle bracket & space from each line (unquote a message)
`$ sed "s/^> //1" file.txt`

9.5. exit

Description

Exit command terminates the execution.

Command format

```
exit
```

9.6. diff

Description

Display the differences between two files, or each corresponding file (with differences) in two directories. For two directories, diff compares corresponding files in both directories, in alphabetical order; this comparison is not recursive. For files that are identical, diff normally produces no output.
Command output convention in displaying the differences:
< - denotes lines only in the first file
> - denotes lines only in the second file

Command format

```
diff [Options] FILES...
```

`Options` – One or more of the following options may appear:

- `-s` - Report when two files are the same by printing a message “Files [file_names] are identical”
- `-B` - Ignore changes with all blank lines
- `-q` - Output only whether files differ by printing a message “Files [file_names] differ”

`FILES` – ‘FILE1 FILE2’ or ‘DIR1 DIR2’. If a FILE is ‘-’, read standard input.

Options can be used in any order or combination: all following options should be allowed for `diff -sBq …, diff -s -B …, diff -qs -B …`

Examples

Differences for two files A.txt and B.txt

```
$ diff A.txt B.txt
```

A.txt	B.txt	Output
line1 line2 line3 line5	line1 line3 line5 line6	<line2 >line6

Differences for two directories A and B

```
$ diff A B
```

An example output follows:

```
Binary files A/image2.bmp and B/image2.bmp differ
Common subdirectories: A/old and B/old
Only in A: out_image2.bmp
```

```
diff A/createAcct.sh B/createAcct.sh
< for u in $ul
> for u in $ul1
```

9.7. grep

Description

The grep command searches for lines containing a match to a specified pattern. The output of the command is the list of the lines matching the pattern. Each line is printed followed by a newline.

Command format

`grep [-i][-c] PATTERN [FILES]`

`PATTERN` – specifies a regular expression in JAVA format (https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html).
`-i` – perform case insensitive matching. By default, grep is case sensitive.
`-c` – Only a count of selected lines is written to standard output.
`FILES` – the name of the file or files. If not specified, use stdin.

Examples

Search the file example.txt for the string “hunting the shark”:
`$ grep “hunting the shark” example.txt`

9.8. wc

Description

The wc utility displays the number of lines, words, and bytes contained in each input file, or standard input (if no file is specified) to the standard output. A line is defined as a string of characters delimited by a <newline> character. Characters beyond the final <newline> character will not be included in the line count. A word is defined as a string of characters delimited by white space characters. If more than one input file is specified, a line of cumulative counts for all the files is displayed on a separate line after the output for the last file.

Command format

`wc [-clw] [FILES]`

By default, wc will display lines, words, and bytes in the format

`lines words bytes filename (filename is empty for standard input)`

One or more of the following arguments can appear. The arguments can appear together `(-cl)` or separated `(-c -l)`

`-c` The number of bytes in each input file is written to the standard output.

`-l` The number of lines in each input file is written to the standard output.

`-w` The number of words in each input file is written to the standard output.

Note that if more than one arguments appear, it will display in the order lines words bytes

`FILES` – the name of the file or files. If not specified, use stdin.

Examples

Display `3 3 21 test.txt`

```
$ wc test.txt
```

Display `3 21`

```
$ wc -c -l < test.txt #input redirection is regarded as standard
                       input

# Display
       3       3      21 test.txt
       3       3      21 test2.txt
       6       6      42 total
$ wc test1.txt text2.txt
```

9.9. cd

Description

The cd command changes the current working directory.

Command format

`cd PATH`

`PATH` – relative or absolute directory path. If the PATH does not exist, raise exception. State your assumptions about your absolute path (highly dependent on the operating system you are using).

Examples

Change current working directory to A/New folder (relative path). The following shell comments will be run in the new current working directory.

$ `cd A/New`

9.10. cp

Description

Copy the content of source file to destination file.

Command format

`cp SOURCE_FILE DESTINATION_FILE`

`cp [FILES] DIRECTORY`

In the first form, copy the contents of source file to destination file.

In the second form, copy contents of each of the named source file to the destination directory. Names of the files themselves are not changed.

Examples

copy content of “foo.txt” to “bar.txt”

```
$ cp foo.txt bar.txt
```

copy all .txt files to the directory “foo”

```
$ cp *.txt foo
```

9.11. cut

Description

Cuts out selected portions of each line (as specified by list) from each file and writes them to the standard output. If no file arguments are specified, cut from the standard input. Column numbering starts from 1.

Command format

```
cut [Option] [LIST] FILES...
```

`Option` – One of the following options may appear:
`-c` – Cut by character position
`-b` – Cut by byte position

`LIST` - can be a list of comma separated numbers, a range of numbers or a single number.

`FILES` – 'FILE1 FILE2…', If a FILE is '-', read standard input.

Examples

Display 'a'
```
$ echo "baz" | cut -b 2
```

Display 'Ts'. Suppose the file contains one line: "Today is Tuesday."
```
$ cut -c 1,8 test.txt
```

Display 'Today is'. Suppose the file contains one line: "Today is Tuesday."
```
$ cut -c 1-8 test.txt
```

9.12. ls

Description

List information about files.

Command format

```
ls [-d][FOLDERS][-R]
```
`FOLDERS` – the name the folder or folders. If no folders are specified, list files for current directory. Hidden files should not be listed.
`-d` – list the folders only
`-R` – List files and subfolders recursively

Examples

List the directories in the current directory
```
$ ls -d */
```

run ls on the expansion of all entries in the current directory
```
$ ls *
```

9.13. sort

Description

The sort command orders the lines of the specified files or input and prints the same lines but in sorted order. Compares each line character by character. A special character (e.g., +) comes before numbers. A number comes before capital letters. A capital letter comes before small letters, etc. Within each character class, the characters are sorted according to their ASCII value.

Command format

`sort [-nrf] [FILES]`

One or more of the following arguments can appear. The arguments can appear together `(-nr)` or separated `(-n -r)`.

`-n` If specified, treat the first word of a line as a number. For instance, sorting file.txt:

```
10
1
2
```

when `-n` is specified -- the order will be `1, 2, 10`.

Otherwise, consider numbers as normal characters -- the order will be `1, 10, 2`

`-r` – Sort in reverse order.

`-f` – Convert all lowercase characters to their uppercase equivalent before comparison, that is, perform case-independent sorting.

`FILES` – the name of the file or files. If not specified, use stdin.

9.14. find

Description

The find application recursively searches a specific file in the given directories and prints corresponding locations to standard output.

Command format

`find FOLDERS -name [filename]`
`FOLDERS` – the name the folder or folders. The folders are recursively searched.
`filename` – the name of file to find specified using a regular expression in JAVA format. The argument after '-name' must be quoted with double quotes.

Example

Display `A/test.txt` if there is a text.txt file in directory A
`$ find A -name "test*"`

No result is displayed if the file named "test" does not exist in folder A
`$ find A -name "test"`

9.15. mv

Description

Move files or directories from one place to another. By default, it will overwrite an existing file.

Command format

```
mv [-n] SOURCE TARGET
mv [-n] [SOURCE] … DIRECTORY
```

In the first form, renames the file named by the source operand to the destination path named by the target operand. This form is assumed when the last operand does not name an already existing directory.
In the second form, moves each file named by a source operand to a destination file in the existing directory named by the directory operand.

`Option` – One of the following option may appear:
`-n` – do not overwrite any existing file.

Examples

move all files with a .txt extension to the folder “test”
```
$ mv *.txt test
```

10. FAQ

Before asking your questions, check the following page with FAQ: https://bit.ly/36t2gpJ

Any questions you might have about the project you can post them on the LumiNUS forum. Alternatively, you may contact our TA Wang Zi (e0053542@u.nus.edu).

11. Submission Instructions

All submission must be made through LumiNUS. Apart from that, for each submission, there must be a corresponding tag in your git repository.

11.1. Milestone 1

Due date is Mon, 2 Mar, 2pm
Important note: The team's name, number, and members' names **should not appear** in any of the files submitted on LumiNUS (only the ZIP file should have the team number)! Your tests will be sent to another team for usage in their test-driven development.

1. Upload a ZIP file in the format described below to LumiNUS Files, under the folder 'Project-Milestone1'
    - Zip file format: **<lab_slot>_<teamNumber>_MS1.zip**
    - Example: fri13_TEAM5_MS1.zip (fri13 stands for the lab slot from Friday, 13:00).
2. In the repository of your team, tag the submitted commit with a tag name "ms1" (meaning milestone 1). Note that the *same files* as those in the submitted zip files should be in the repository.
    - Create a tag, e.g., **git tag -a ms1 -m "Milestone 1"**
    - Push the created tag into the repository, e.g., **git push origin ms1**
    - More details are in: http://git-scm.com/book/en/v2/Git-Basics-Tagging

Your ZIP file submitted on LumiNUS should contain the following (basically, all your IntelliJ project files):

- A document (**Assumptions.pdf**) containing the assumptions you have made in your implementation. Include here any clarifications you might have made to the applications specification.
- A **report file (named Milestone1.pdf)** in bullet-point format containing (max of 4 pages, 10pt font):
    - Your plans for implementation and testing covering the process you followed for implementation and testing, how test cases were generated to ensure full coverage for individual components and the integration of components, tools and techniques used for testing etc.
    - Summary of test cases provided (what have you covered during testing, did you have any plan for generating tests?, etc.)
- The **fixed template code** without any of our faults. To be on the safe side, you can also submit the test cases for each of our faults in the template code.
- The clean, documented **source code for basic functionality** (BF)
- The clean, documented **source code for one of the extended functionalities (EF1 or EF2)**
    - Please check on LumiNUS Project->Project Topics if you have to implement EF1 or EF2. Half of the teams have been assigned to implement EF1 and the other half EF2. Implement the functionalities according to this assignment.
- **Unit tests** for basic and extended functionalities: **BF and EF1 and EF2.**
- **Test cases for all functionalities.**
- You can separate the test cases for unimplemented functionalities in different folders. You should also include documentation for all the test cases.
- Indicate how to run all your test cases. The tests for the unimplemented functionality may fail.
- Any resource files used in the project.

After Milestone 1, the teaching team will publish a set of test cases to be used by teams in TDD in Milestone 2. These test cases might not be used as they are by all teams. Adjustments, changes, and re-implementation might be needed to match the assumptions your team has made.

Assessment

- **Unit Tests [6 marks]**
  - Unit tests for BF, EF1, and EF2 are graded for all teams.
- **Integration tests [1 mark]**
  - Whether various chains of interactions and integration upon shell state are tested properly
  - Whether both positive and negative tests (exceptional case/corner case/error handling) are written.
- **Implementation [2 marks]**
  - The implementation for the requested functionalities and applications is successfully completed

11.2. Milestone 2

Due date is Mon, 23 Mar, 2pm.

Important note: The team's name, number, and members' names **should not appear** in any of the files submitted on LumiNUS (only the ZIP file should have the team number)! Your code and tests will be sent to another team during the Hackathon.

1. Upload a ZIP file in the format described below to LumiNUS Files, under the folder 'Project-Milestone2'
   - Zip file format: **<lab_slot>_<teamNumber>_MS2.zip**
   - Example: fri13_TEAM3_MS2.zip (fri 13 stands for the lab slot on Fri, 13:00).
2. In the repository of your team, tag the submitted commit with a tag name "ms2" (meaning milestone 2). Note that the *same files* as those in the submitted zip files should be in the repository.
   - Create a tag, e.g., **git tag -a ms2 -m "Milestone 2"**
   - Push the created tag into the repository, e.g., **git push origin ms2**
   - More details are in: http://git-scm.com/book/en/v2/Git-Basics-Tagging

Your ZIP file submitted on LumiNUS should contain the following (basically, all your IntelliJ project files):

a. A document (**Assumptions.pdf**) containing the assumptions you have made in your implementation (include here any additions you might have made to the applications specification).

b. A report file (**Milestone2.pdf**) in bullet-point format containing (max of 4 pages, 10pt font):
   - Details about TDD process. Please document briefly in your report the experience you had using the testcases from other teams.
   - Integration testing – plan and execution

c. The clean, documented **implementation** of Basic Functionality (**BF**) and both Extended Functionalities (**EF1 & EF2**)

d. **All your test cases! You should include unit, integrations and system test cases**
e. Document any assumptions that you have made to ensure that features do not get classified as bugs (in the Hackathon).
f. Any resource files used in the project.

Document your code! Write positive and negative test cases! After the Hackathon you will need to defend your code and reduce the number of bugs issued against your code.

Assessment

- **Integration tests [3 marks]**
  - Whether various chains of interactions and integration upon shell state are tested properly
  - Whether both positive and negative tests (exceptional case/corner case/error handling) are written.
- **TDD [3 marks]**
  - Whether all test cases provided for TDD exercise are passing.
  - Whether more test cases are written.
- **Implementation and Code quality [2 marks]**
  - Whether source code and test cases are successfully tested with PMD.
  - Whether source code and test cases are properly formatted using the IntelliJ formatter.
  - Whether JUnit tests are properly written in a good style
- **Effective use of automatic testing tools [*up to* 2 bonus mark]**
  - If the tools discovers a bug, a bug report must be provided that includes (1) generated test, (2) buggy code fragment, (3) applied fix, (4) command line options and other resources
  - However, if the tools did not produce any error revealing tests, then you can explain how you used the regression tests generated by these tools in your project.
  - In the submission you should only include the tests that were relevant with a brief description of tests that were not considered and why they weren't.

11.3. Hackathon

Due on Fri, 27 Mar, 2pm.
Your team will receive the source code implemented by 3 other teams. You have to test these projects and find as many bugs as possible in a short-time. This activity should take place during the lab session and points will be given for your activity. You should reuse the test cases you have written for your own implementation, or prepare new testcase.

Submission:

1. Upload a ZIP file in the format described below to LumiNUS Files, under the folder 'Project-Hackathon'
    - Zip file format: **<lab_slot>_<teamNumber>_Hackathon.zip**
    - Example: fri13_TEAM3_Hackathon.zip (fri13 stands for the lab slot on Fri, 13:00).

Your ZIP file submitted on LumiNUS should contain the following (basically, all your IntelliJ project files):

a. 3 report files (BugReports_for<teamLetter>.pdf) in bullet-point format containing:
    - General method used in testing
    - A table containing the bug reports:

Bug Report Number	**Description**	**Testcase**	**Comments (this column should be left empty; for marking usage)**
1	…	…	

b. The **implementation** you have received for testing (pertaining to the other teams)
c. **Testcases that generate errors** (please do not submit all test cases, but only those that show the presence of bugs for each team, in a separate folder)
d. **Any other resource files used.**

Assessment

- **Hackathon Bugs [6 marks]** - Teams can get up to 6 marks for finding and documenting bugs in the implementation provided. Marks will be awarded after rebuttal.

11.4. Rebuttal

Due on Mon, 30 Mar, 2pm.

Your team will receive the bug reports produced by other teams for your implementation. You will be able to classify the bug reports provided by the hacking team as valid or invalid. The bug rebuttal file will be discussed with the tutor during the lab session on Wednesday or Friday.

Submission:

1. Upload a ZIP file in the format described below to LumiNUS Files, under the folder 'Project-Rebuttal'
    - Zip file format: **<lab_slot>_<teamNumber>_Rebuttal.zip**
    - Example: fri13_TEAM3_Rebuttal.zip (fri13 stands for the lab slot on Fri,13:00).

Your ZIP file submitted on LumiNUS should contain the following:

a. The report files "**BugReports_from<teamNumber>.pdf**" received by the team describing the bugs found in their code by another team.
b. Report files named "**Rebuttal_for<teamNumber>.pdf**" containing the following table (one rebuttal file for each bug report received). The numbers of the bug reports should correspond with those found in "BugReports_from<teamNumber>.pdf":

Bug Report Number	**Status**	**Reason if invalid**	**Comments (this column should be left empty; for marking usage)**
1	Valid/ Invalid	(e.g. duplicate of bug report #X, bug report is not in accordance with project specification on page Y line Z "quote line", …)	

Assessment

- **Hackathon Bugs [2 marks]** - Teams can get up to 2 marks for their rebuttal – preparing the rebuttal files and discussing with other the teams.

11.5. Milestone 3

Due date is Sat, 18 Apr, 10am.

1. Upload a ZIP file in the format described below to LumiNUS Files, under the folder 'Project-Milestone3'
    - Zip file format: **<lab_slot>_<teamNumber>_MS3.zip**
    - Example: fri13_TEAM3_MS3.zip (fri13 stands for the lab slot on Fri,13:00).
2. In the repository of your team, tag the submitted commit with a tag name "ms3" (meaning milestone 3). Note that the *same files* as those in the submitted zip files should be in the repository.
    - Create a tag, e.g., **git tag -a ms3 -m "Milestone 3"**
    - Push the created tag into the repository, e.g., **git push origin ms3**
    - More details are in: http://git-scm.com/book/en/v2/Git-Basics-Tagging

Your ZIP file submitted on LumiNUS should contain the following (basically, all your IntelliJ project files):

a. The quality assurance report file (**Milestone3-QAreport.pdf**)
    - The template for the QA report will be posted on LumiNUS. The QA report will be described in detail during the lab session in week 12.

b. A report file named "**Fixes.pdf**" containing the following table. The numbers of the bug reports should correspond the valid bugs found in "BugReports.pdf" and "Rebuttal.pdf":

Bug Report Number	**Status**	**Testcase (filename)**	**Comments (this column should be left empty; for marking usage)**
from<teamNumber>_1	Fixed/ Not fixed	…	

c. The report file "**BugReports_from<teamNumber>.pdf**" received by the team describing the bugs found by another team, received before the Rebuttal phase (Note: you should not submit the BugReport file you have generated for another team's implementation).

d. The report files previously submitted in the Rebuttal phase "**Rebuttal_for<teamNumber>.pdf**"

e. The clean, documented **implementation** of Basic Functionality (**BF**) and both Extended Functionalities (**EF1 & EF2**). This code should pass all test-cases written by your team as well as at least 30% of the Hackathon test cases (bug reports).

f. **All your test cases!** A separate file (folder) named "Hackathon" should be included. This file should contain all test cases for bugs found during Hackathon. Your implementation should pass at least 30% of these test cases found during the Hackathon. Note that this file will be evaluated.

g. Any resource files used in the project.

Assessment

- **Quality Assurance (QA) Report [8 marks]**
- **Hackathon Fixes [2 marks]** - Fixes for bugs found during the Hackathon.
- The final code must adhere to code quality standards. If the code is found not adhering to code quality standards or the code is found to be failing for many test cases, marks may be deducted.