CS 35L Week 2 Worksheet

Bash Scripting, Regular Expressions, & Emacs Lisp Worksheet

Bash Scripting:

The following questions are meant to give you some practice to understand Bash Scripting. Feel free to use this cheat sheet.

1. **Syntax**. Bash is a programming language. However, there are a lot of things different about Bash from your typical scripting language. Let's take a look at some of them.
    a. Let's take a look at the syntax of Bash by referencing this cheat sheet. What are some similarities and differences between other programming languages you have used in the past? (most likely C++).

Block Scope:

- Bash lacks block-level scoping (like {} blocks in C++). A variable declared within an if or for block in Bash remains accessible outside the block.
- In C++, {} blocks define a new scope, so variables declared within a block are limited to that block

Basic Arithmetic (+, -, *, /):

- Both Bash and C++ support these operators, but their usage differs:
    - In C++, arithmetic operators work directly on integers, floats, and doubles.
    - In Bash, arithmetic needs to be enclosed within $((...)) or let for integer calculations:

```
x=$((5 + 3))  # Bash arithmetic
let x=x+1    # Increment in Bash
```

Equality and Inequality (==, !=):

- In C++, you use == and != for comparisons, regardless of data type.
- In Bash, you need to use -eq, -ne, -lt, -le, -gt, -ge for numerical comparisons:

Logical Operators (&&, ||):

- C++ supports && for logical AND and || for logical OR.
- Bash also supports && and || in [[...]] and for command chaining, but within test or [...] conditions, -a and -o are often used for AND/OR, though they're less common due to readability concerns.

    a. Bash is the first example of a **scripting** language (we will talk about many of these in CS 35L). Particularly, it is **interpreted** as opposed to **compiled** (e.g. C++). What are the implications of this in terms of handling invalid syntax? Take a look at the same programming written in Bash and C++ and talk about the differences (try running both if you are unsure).

Bash	C++

`#!/bin/bash`
 	
`# Here is a comment!`
`echo "Hello world!"`
`notacommand;`
`exit 0`
	`#include <iostream>`
	`using namespace std;`
	
	`int main() {`
	`  // Here is a comment!`
	`  cout << "Hello world!" << endl;`
	`  notacommand();`
	`  return 0;`
	`}`

In an interpreted language like Bash, syntax is checked and executed line-by-line at runtime. This means that invalid syntax in Bash may not be detected until the script reaches that specific line during execution. If an error is encountered, Bash will stop running and output an error message. This can lead to situations where part of the script runs successfully before an error halts further execution, making debugging iterative since you may need to run the script multiple times to uncover each syntax error.

On the other hand, C++ is a compiled language, meaning the entire code is analyzed for syntax and (some) semantic correctness before it's converted into machine code by the compiler. As a result, syntax errors are identified during the compilation phase, and the program will not run until all errors are fixed. This up-front error-checking provides more detailed feedback on what went wrong, often with specific line numbers and descriptions, making it easier to catch errors before runtime.

b. An interesting (and completely valid syntactically) example of a valid Bash script is `:(){ :|:& };:`. This is a fork bomb which can cause denial of service (DoS) attacks on Linux machines. What does each symbol mean?

This is a really good example of how to write a **function** in Bash. The above link provides a more detailed explanation on how this works. Here is the breakdown of each symbol.

- **:** is the **name** of the function. Rewriting this use the function name **foo** you can read the program as `foo(){ foo|foo& };foo` . The above program defines a function called "**:**" and calls the function recursively inside of it. Outside of the function it then calls its defined function again. Note: unlike C++, you don't need to have () when **calling** the function.

- **(),{}** are the symbols which define the function. () is needed to indicate that you are defining a function and {} help enclose the function body.
- **;** indicates the end of the function declaration.
- | is the pipe operator. & is the operator indicating to fork the current command. This is the key part of the program since the fork bomb constantly forks the current function which creates the Denial of Service (DoS) attack.

A quick summary on what the above fork bomb example does. It defines a function called ":" where it calls itself recursively inside of the function and pipes the output of one call of the function to another call of the function which runs all of this in the background by forking it (using &). After this function declaration, it calls the function it defined. Even though this shell script is very simple and only forks the current process a bunch of times, this is enough to use all of a computer's resources and cause a crash of your computer which is why it has dangerous security implications.

(Spring 2024 Midterm) In the standard shell, the built-in command 'eval X' treats X as if it were a command and then executes that command. Also, Bash and some other shells have a 'select' command with a special syntax, as an extension to the standard shell (and where the details of the syntax and semantics of this command do not matter to this question); whereas other shells do not support 'select' and will print a brief syntax error message and exit if the script tries to use 'select' in this way.

Suppose the shell script 'foosh' contains the following code:

```
if
    (eval 'select x in; do break; done') </dev/null 2>&0
then
    eval 'select x in foo bar baz; do echo "$x"; done'
else
    echo "$0: 'select' is not implemented." >&2 &&
    echo "$0: Please use a better shell, like Bash." >&2
    exit 1
fi
```

Explain what this code is doing and why. For example, why does it use 'eval X' (twice) and not just plain 'X'?

Solution (IO Redirection, order of shell parsing, subshells):

The script tests if the shell supports the `select` command and behaves differently based on the result:

- If `select` is supported by the Shell, the script runs the `select` command.
- If `select` is not supported by the Shell, the script prints an error message and exits.

1. First `eval` in the `if` Statement:

- The first `eval` checks if `select` is supported by attempting to execute the command silently (`</dev/null 2>&0`). If it fails, the script proceeds to the `else` block without crashing.

2. Second `eval` in the `then` Block:

- Even though the first `eval` ensures that the `select` construct exists, it still can't be used directly, as `do`/`done` are special syntax constructs that are only valid as part of a loop

body. Even if the body of the `then` statement is being skipped due to the condition failing, the shell still needs to parse the following lines to find the `else`, and thus will syntax error anyways without the `eval`.

If you remove `eval` and directly run `select x in; do break; done`:

- **In a Shell Supporting `select`**:
  - It works without issue.
- **In a Shell Without `select`**:
  - The shell attempts to parse the script and fails immediately, causing the script to terminate with a syntax error before even reaching the `if` condition.

Why Use `eval`?

`eval` takes a string as input and interprets it as a shell command. Without `eval`, the shell immediately encounters any syntax errors in unsupported constructs, i.e. the select

  - By wrapping `select x in; do break; done` inside `eval`, the script delays the parsing and execution of the command string until runtime, i.e. **`select will be parsed at runtime.`**

In unsupported shells, the syntax error occurs **at runtime** when `eval` attempts to execute the command. This allows the script to test for support dynamically and handle errors gracefully.

Subshells, i.e. commands in parentheses()

1. **Subshell Execution**:
   - The parentheses create a subshell, meaning the command inside the parentheses (e.g. select) is executed in a new, child shell process, separate from the current shell.
   - This isolation ensures that any changes to the shell environment (like variables, file descriptors, etc.) made inside the subshell do not affect the parent shell.
2. **Error Isolation**:
   - If the `eval` command inside the parentheses causes an error (e.g., a syntax error in a shell that does not support `select`), the error is contained within the subshell. The parent shell remains unaffected.
   - The `<>/dev/null 2>&0` redirects standard input and error output to `/dev/null`, suppressing any visible error messages from the `eval` command.
3. **Logical Test for `if`**:
   - The `if` statement evaluates the exit status of the subshell:
     - If the command inside the parentheses (`eval 'select x in; do break; done'`) executes successfully, the subshell exits with status `0`, and the `if` block is executed.

- If the command fails (e.g., due to a syntax error in an unsupported shell), the subshell exits with a non-zero status, and the `else` block is executed.

General Order of Shell parsing:

https://mywiki.wooledge.org/BashParser

2. Syntax Soup. Bash has a lot of useful and interesting syntax for many of its commands. Let's try and explore some of them by comparing some example programs. Try and note the differences between each of the following pairs of commands.

c. `echo /etc/*` vs `echo /etc/???`

This shows the difference between globbing and wildcards in Bash. The first command will show the paths of all of the files in the /etc directory (* is a wildcard which matches anything) while the second command will only show the contents of files that have names that are three characters long (? is a wildcard that matches only one letter).

d. `a | b` vs `a || b`

The left example shows the pipe operator. The right example shows the logical OR operator in Bash. One quick distinction with Bash is that unlike the OR operator in C++ which uses Booleans, the logical OR operator uses status codes which may be unintuitive. For example, **echo "hello" || false** has status codes 0 || 1 which results in the overall operation evaluating to 0 indicating a SUCCESS (in C++ it's the opposite since C++ considers 1 to be true). The logical OR also has some weird behavior with short circuiting similar to booleans but every point of the code is still run. An example of short circuiting is **true || echo "hello"** which will print nothing and **cat doesnotexist || echo "hello"** which will print both the error message from cat and "hello".

e. `a & b` vs `a && b`

The left example shows the fork operator which runs the programs **a** and **b** in parallel (if you apply this command and the end of a command, this runs the program in the background. See the fork bomb example above for an example of this). The right example shows the logical AND operator. Similar to the logical OR operator, the logical AND operator may be a bit unintuitive since it works on status code and has some interesting behavior while short circuiting. **echo "hello" && false** will have status codes 0 && 1 which will result in an overall status code of 1. However, it will also print **"hello"** because the first command is still run. **false && echo "hello"** will return with a status code of 1 but it will print **nothing**. **true && echo "hello"** is an example of a command with returns with a status code of 0.

f. `echo $VARIATION` vs `echo ${VAR}IATION`

The first calls variables normally, looking for a variable named VARIATION and echoing its value. The second calls variables as a grouping expression, looking for a variable named VAR and echoing the concatenation of the value of VAR with the string "IATION". In Bash, curly braces are "optional" for variable names in the sense that using a dollar sign followed by the variable name by itself does not require braces, but if you want to concatenate the value of the variable with some string afterward, you must use curly braces to indicate what is the variable name and what is just part of the string.

g. `echo ${10}` vs `echo $10`

The first echoes the 10th positional argument. The second echoes the 1st position argument followed by the string "0". In Bash, **positional arguments** are referenced using $ followed by a number. For example, if I named my script foo.sh, then **./foo.sh first** would print nothing for the first (since there is no 10th positional argument), and "first" for the second (since the 1st positional argument is "first").

h. `echo $pwd` vs `echo $(pwd)`

The first echoes the value of a variable named pwd (see part f, first example). The second echoes the result of the command pwd. **$()** syntax lets you call commands, directly embedding the string into the command. **$(())** does something similar in that it involves arithmetic evaluation (see later examples).

i. `echo "Hello" > a` vs `echo "Hello" >> a`

> clobbers (overwrites) a file. It does not matter whether the content length is greater than the contents of where it's being redirected to, the entire contents of **a** will be replaced with "Hello". **>>** is appending to the end of a file. The contents of the file will not be overwritten and instead "Hello" will be appended to the end of file **a**.

j. `if [ $# != "1" ]; then echo "Error" >&2; else echo "$1"; fi` vs
`if [[ $# -ne 1 ]]; then echo "Error" >&2; else echo "$1"; fi`

This example shows two ways of writing conditional statements within Bash. [ ] is syntactic sugar for the command **test** which is responsible for evaluating these sorts of conditional statements. There are some useful features of using the double brackets which you can check out using this link but the single brackets are more backwards compatible while the double brackets are more convenient. Note != is for comparing strings and -ne is numerical comparison.

k. `for i in {1..10}; do echo "$1"; done` vs
`for ((i = 0; i < 10; i++)); do echo "$1"; done`

Two different ways of writing for loops in Bash. The first one expands a list and does a for-each loop over the entries of the list. The second way evaluates inside Bash's arithmetic evaluation mode.

l. Focus not only on the while loop but the usage of brackets (i.e. [ ]).

```
arr=(1 2 3 4 5 6)
x=1
while [ $x -le 5 ]; do
     echo "Welcome ${arr[x]} times"
     x=$(( $x + 1 ))
```

```
arr=(1 2 3 4 5 6)
x=1
while (($x <= 5)); do
     echo "Welcome ${arr[x]} times"
     x=$(( $x + 1 ))
```

`done`	`done`

Two different ways to iterate over an array using two different modes. For more information, check out this link.

- **Unit Testing**. A big idea of CS 35L is working the mentality of test driven development (TDD). The idea is that you often write the test cases for your program FIRST before implementing the program. Suppose you are attempting to follow TDD and want to write unit tests for the Bash script echo2. Write a separate Bash script called **test_echo2.sh** which includes a few unit tests for echo2. If a test fails, it should print an error message indicating that it failed. If a test succeeds, it should print a success message indicating that "Test #" has passed. If you are feeling ambitious, you could also try and implement calculating **coverage** (we are using a slightly lax definition of coverage in that we are just looking for you to print what percentage of test cases passed / failed).

We will be revisiting the ideas in this example when we tackle Assignment 6 which talks more about test driven development (TDD). Getting some practice ahead of time! Helpful: grep -q returns a status code of 0 if the search succeeds and 1 if the search fails.

Note: There are often a lot of ways to do the same thing in Bash. For example, in the script below, the ! command is used with if to check if a command failed. You could also run the command, then check the value of $? with if (as explained later in the worksheet).

```
#!/bin/bash

PASSED=0

echo "Test 1 - Runs without error."
if ./echo2 "a" >/dev/null; then
   echo "PASSED"
   PASSED=$((PASSED + 1))
else
   echo "FAILED"
fi

echo "Test 2 - Correct output."
if [[ $(./echo2 "a" | grep -ow "a" | wc -l) -eq 2 ]]; then
```

```
    echo "PASSED"
    PASSED=$((PASSED + 1))
else
    echo "FAILED"
fi

echo "Test 3 - Runs with error."
if ! ./echo2 2>/dev/null; then
    echo "PASSED"
    PASSED=$((PASSED + 1))
else
    echo "FAILED"
fi

echo "Test 4 - Prints error message."
if ./echo2 2>&1 | grep -q "Error"; then
    echo "PASSED"
    PASSED=$((PASSED + 1))
else
    echo "FAILED"
fi

# Final coverage
COVERAGE=$((PASSED * 100 / 4))
echo "Coverage: $COVERAGE%"
```

- **Echo (Echo)**. Write a basic shell script that implements echo2, where it acts like the default echo command if you run the echo command twice. Remember to quote your input.

Quoting your input is important to prevent a common bug in Bash known as argument splitting (when there is a space in your input causing Bash to treat your input as two commands rather than one).

a. Add error handling so that if echo2 does not have one argument, it will print an error message like "Oops not enough args" to standard error.

echo2

```
#!/bin/bash

if [[ $# -ne 1 ]]
then
    echo "Error" >&2
    exit 1
fi

echo "$1"
echo "$1"
```

b. Extend your implementation of echo2 to echon, which will act as if you used the echo command n times, where n is a non-negative integer. Assume the following: the first argument is n, a valid non-negative integer; and the second argument is the content that will be echoed and will be quoted.

echon

```
#!/bin/bash

if [[ $# -ne 2 ]]; then
    echo "Error" >&2
    exit 1
fi

for ((i = 0; i < $1; i++)); do
    echo "$2"
done
```

c. Let's replace echo with a fake program called **fakeprogram**. Suppose we added this to our path and called it inside our bash script in the place of echo. **fakeprogram** was written by students so it fails about 50% of the time so you need to implement error handling for if the status code of the first **fakeprogram** is 1, then you should print an error message like "Oops, the first fakeprogram failed!" to standard error and exit with a status code of 1. Do the same for the second call of **fakeprogram**.

echo2 with fakeprogram

```
#!/bin/bash
```

```
if [[ $# -ne 1 ]]; then
    echo "Oops not enough args" >&2
    exit 1
fi


if ! fakeprogram "$1"; then
    echo "Oops, the first fakeprogram failed!" >&2
    exit 1
fi

if ! fakeprogram "$1"; then
     echo "Oops, the second fakeprogram failed!" >&2
     exit 1
fi
```

- **CS 35L Fall 2023 Midterm.** The follow is an interesting midterm question which highlights the difference use cases of I/O redirection, common Bash scripting ideas, and the **|** and **&** operations. **Note**: potentially some of the operations might not have an answer.

(8 minutes). Suppose you want the effect of these four shell commands:

```
(A && B) >C

(D || E) <F

(G | H) >>I

(J & K) 2>L
```

However, your keyboard's '`&`' and '`|`' keys are broken; you can't use those two characters. What shell commands would you use instead? Try to be as simple and as elegant as possible.

If it's not possible to do one or more of these four commands without using '`&`' and '`|`', briefly explain why not.

- **(A && B) > C**. Can be rewritten using an if-statement. The main idea is that we want to check the status code of each program run. Here it is as a one-liner. Note that **$?** is a special variable in Bash that contains the status code of the previous command run.
    - **A>C; if [[$? -ne 0]]; then exit 1; fi; B>>C**
- **(A || E) <F**. Can be rewriting similarly to the first question using an if-statement while also checking the status code for short circuiting.
    - **A < F; if [[$? -eq 0]]; then exit 0; fi; E < F**
    - **Aside.** This solution is slightly incomplete because it does not handle partial reads from a file and continues later on by running the command. However, the emphasis of the LA solution here focuses on combining information learned in class to tackle the question which is the approach on exams.
- **(G | H) >> I**. Can be rewritten using I/O redirection and process substitution. We can use **<()** syntax to let us refer to the standard output of G as a file. Then, we redirect the stdout of G into the stdin of H.
    - **H < <(G) >> I**
    - **Note:** This syntax ***is not*** **<<** followed by **()**. This syntax should be read as **<** followed by **<()**.
    - **Aside:** We can use **>()** syntax to refer to the standard input of the command as a file. See the man pages for more info.
- **(J & K) 2>L.** Many students were asking about this. Since "**&**" represents the fork operator and in-class, we did not learn about a reasonable alternative to this operation, then you can argue that using basic primitives of Bash scripting that recreating this operation is **not possible**.
    - **Aside**. It may be possible to rewrite this using the **exec** command but that's not in the scope of this class.

Regular Expressions (RegEx):

The following questions are to give extra practice with using regular expressions (regex). These cheat sheets: link 1, link 2.

1. Regular expressions are useful for capturing patterns which is a common use case in software construction. What might be some use cases where we want to capture patterns quickly when writing/working with software? Feel free to also check out some of the resources below which give a nice introduction to regex.

Regular expressions are often useful since we frequently need to perform search of certain patterns while reading source code or working on scripting applications such as web scraping to quickly parse large amounts of data.

a. A useful website when debugging your regular expressions is https://regex101.com/. Try checking out the "Explanation" window and writing some test cases to match your regex. The "Regex Debugger" is also for helping you understand why your regex does not match certain test cases.
    i. Very useful for stepping through your examples and seeing how your regex matches certain patterns.
b. In Assignment 1, we used "linux.words" as a dictionary for our spell checker. As a refresher, each line in "linux.words" is a separate entry in the dictionary. Write a regular expression that accepts **UCLA student IDs** (9 digits, only numbers) in linux.words. That is, think of a regular expression that **matches all lines that contain a valid UCLA ID**. (**Hint**: Start with your own UCLA ID. Think of what invariants (features) characterize the string.)
    i. ^[0-9]{9}$
c. Regex has a lot of supporting computer science theory with finite automata (something you will learn more about in CS 181). This knowledge is a bit outside the scope of this course but if you are interested, a former LA made a "Regex in One Page" document that uses some of the ideas from finite automata.
    i. An interesting application of using finite automata and regex is within scanners/lexers for programming languages. If you are interested in learning about this, take CS 132: Compiler Construction!

2. **Activity**. What is a regular expression for all words in **/usr/share/dict/linux.words** that matches all solutions of Wordle? Can you write a shell script to solve today's Wordle using **grep**, **/usr/share/dict/linux.words**, and **shuf**?

All valid Wordle words in linux.words:
```
grep -E "^[a-z]{5}$" /usr/share/dict/linux.words
```

Approach to solve the daily Wordle.
- `grep -E "^[a-z]{5}" /usr/share/dict/linux.words | shuf -n 1`
- `grep -E "^[a-z]{5}" /usr/share/dict/linux.words | grep -E "^[^def]{3}[^defg][^def]$" | grep -E "g" | shuf -n 1`
- ...

A challenge for yourself is after you have found today's Wordle answer, try combining all of your grep's into one regex. **Hint**: use the alteration operator (i.e. the OR operator **|**) to combine together. If you want some extra practice with Regex, try solving the Wordle for a different day!

3. **Activity**. What is a regular expression that can match all (more than 99%) of email addresses? Let's try **eggert@cs.ucla.edu** first. What about an email address like **universal.hipster202@hotmail.com**? Let's use ERE to do this :)

   ```
   [[:space:]]*[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}[[:space:]]*
   ```

   The [[:space:]] matches for blank space; the {2,} says that the length of the group must be between 2 and an unlimited number (i.e. the top-level domain must have at least two letters!)

   What if we want to use only single line input?

   ```
   ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
   ```

   What if we want to do this in BRE instead?

   ```
   [[:space:]]*[a-zA-Z0-9._%+-]\+@[a-zA-Z0-9.-]\+.[a-zA-Z]\{2,\}[[:space:]]*
   ```

   Only accept .com or .gov endings:

   ```
   [[:space:]]*[a-zA-Z0-9._%+-]\+@[a-zA-Z0-9.-]\+.\(com\|gov\)[[:space:]]*
   ```

   The longer answer: https://www.regular-expressions.info/email.html

4. **POSIX Regular Expressions**. There are actually many different flavors of regular expressions (e.g. Perl Compatible Regular Expressions - PCRE). However, the main focus of this class involves using POSIX regular expressions which come in two flavors: Basic Regular Expressions (**BRE**) and Extended Regular Expressions (**ERE**). **grep** by default uses **BRE** and **grep -E** uses **ERE**. What are the similarities and differences between **BRE** and **ERE**?

   The main difference between BRE and ERE involves the usage of **meta-characters**, specifically these meta-characters:

   - ?
   - +

- {
- |
- (
-)

In BRE, the meta-characters **?, +, {, |, (,** and **)** lose their special meaning and should be escaped if we want their special meaning. For example, **+** would literally match a plus sign. If we wanted to use the special meaning (oen or more), we need to escape them using ****. For example, **a\+** matches one or more a's.

Note: The closing curly bracket **}** does not need to be escaped. Whether or not a closing curly bracket is special is determined by the presence of **\{** before.

In EREs, the escaping is flipped; the default is that **?, +, {, |, (,** and **)** have their special meaning and escaping them removes the special meaning. If we want to match one on or more, we would use **+**; if we want to match a literal plus sign, we would use **\+**.

5. **Match Me**. Write a regular expression that matches each of the following data. You can write this using **ERE**. If you want extra practice, try writing all of these using **BRE**. If you get stuck, try using https://regex101.com/. **Note:** The default regex engine used by regex101 (PCRE2 with JavaScript syntax) requires you to **escape /**. That is, if you want a literal / in your regex, you must escape it using **\/**. If you need more explanations on any of the examples below, copy and paste the examples and regex into regex101.com and use the "Explanation" tab to understand how the regex works.

Data	Match	Regex
At his court, Orsino, sick with love for the Lady Olivia, learns from his messenger that she is grieving for her dead brother and refuses to be seen for seven years.	At his court Orsino sick ... **Note:** Goal is to extract all valid words from the Shakespearean play.	[A-Za-z]+ Words contain only letters and must contain at least one character.
/[a-z]/ /[0-9]*/	/[a-z]/ /[0-9]*/	\/.*\/ This introduces the idea of

//	// **Note**: This is the syntax that JavaScript uses when writing regular expressions. The goal here is to write a regular expression that captures regular expressions. :)	meta-characters (special characters within regex) by using a nice example of writing a regular expression to capture a regular expression. **".*"** means 0 or more of any characters. For the purposes of this assignment, remember to treat / as a special character within the regex, so we have to escape the character (see above note).

bool a = true;
bool b = false;
bool ab = true;
bool ba = false;
int a = 1;
	a = true
	b = false
	ab = true
	ba = false
		[a-z]+ = (true|false)
		
		Shows the idea of using groups and the alteration (i.e. OR) operator (|).

capture me 12345	capture me 12345	(?<=).*(?=<\/ol>) This example shows the power of using lookahead and lookbehinds (i.e. we want to find patterns that are before and after but we only want to capture what's in between). Note: lookaheads and lookbehinds are only a feature of PCRE (used in grep -P on GNU Grep).
abccccccccdxabcd abdxabc	abccccccccdxabcd abdxabc	(abc*d)x(abc*d) Shows capturing groups.
abcdxabcd abdxabd	abcdxabcd abdxabd	(abc*d)x\1 Example from lecture. Shows backreference to match the

abccccccccdxabcd		pattern captured by the first group.

6. Shipping Company

Background:
Your company is developing an address validation system for a shipping logistics application. Customers often input addresses incorrectly when placing orders, leading to failed deliveries. To improve this system, you've been assigned to write a regular expression that identifies valid addresses from a dataset of customer-provided entries.

You are tasked with extracting valid addresses from a large dataset of customer information. A valid address follows this format:

<Street>, <City>, <State>, <Zip-Code>

Where:

1. **Street**: Starts with a number, followed by one or more alphabetic words, which can include spaces (e.g., 123 Main Street).
2. **City**: Contains only alphabetic characters (e.g., Springfield).
3. **State**: Contains only alphabetic characters (e.g., California or CA).
4. **Zip-Code**: Is either 5 digits (e.g., 12345) or 9 digits (e.g., 12345-6789).
5. **Separators**: Fields are separated by a comma followed by a space (,).

Your task is to write a **GNU Extended Regular Expression (ERE)** to **extract only valid addresses from a text file.** Each line in the file contains a potential address, but only some lines meet the criteria. Invalid entries should not match.

Input/Output Examples:

Valid Inputs:

- 123 Main Street, Springfield, California, 12345: ✅ Matches
- 456 Elm St, Metropolis, IL, 54321-9876: ✅ Matches
- 789 Broadway Ave, Gotham, NY, 10001: ✅ Matches

Invalid Inputs:

- 123MainStreet Springfield California 12345: ❌ Missing commas
- Elm Street, Gotham, NY, 10001: ❌ Missing street number
- 456 Elm St, Springfield, IL, 1234: ❌ Invalid zip code (only 4 digits)
- 123 Main St, Gotham, N-Y, 10001: ❌ Invalid state format (contains -)

Solution:

```
^[0-9]+ [A-Za-z ]+, [A-Za-z]+, [A-Za-z]+, [0-9]{5}(-[0-9]{4})?$
```

```
# In practice
grep -E '^[0-9]+ [A-Za-z ]+, [A-Za-z]+, [A-Za-z]+, [0-9]{5}(-[0-9]{4})?$' addresses.txt > valid_addresses.txt
```

For student reference:

- **^**: Anchors the match to the start of the line to ensure the entire line is validated.
- **[0-9]+**: Matches the street number, which consists of one or more digits.
- **(space)**: Ensures a space after the street number.
- **[A-Za-z]+**: Matches the street name, allowing only alphabetic characters and spaces. This ensures the street name can have multiple words (e.g., "Main Street").
- **,** : Matches a comma followed by a space, separating the street from the city.
- **[A-Za-z]+**: Matches the city name, allowing only alphabetic characters (e.g., "Springfield"). Does not allow spaces or numbers.
- **,** : Matches another comma followed by a space, separating the city from the state.
- **[A-Za-z]+**: Matches the state name, allowing only alphabetic characters (e.g., "California" or "CA"). Does not allow spaces or special characters.
- **,** : Matches a comma followed by a space, separating the state from the zip code.
- **[0-9]{5}**: Matches exactly 5 digits for the standard zip code format.
- **(-[0-9]{4})?**: Matches an optional hyphen followed by 4 digits, allowing for the extended 9-digit zip code format.
- (-[0-9]{4}): The hyphen and 4 digits.
- ?: Makes the extended zip code optional.
- **$**: Anchors the match to the end of the line to ensure no extra characters appear after the zip code.

Helpful cheatsheat: https://learnbyexample.github.io/gnu-bre-ere-cheatsheet/

Pattern	Description
`^`	restricts the match to the start of the string
`$`	restricts the match to the end of the string

`.`	match any character, including the newline character
`?`	match `0` or `1` times
	use `\?` in BRE mode
`*`	match `0` or more times
`+`	match `1` or more times

	use `\+` in BRE mode
`{m,n}`	match `m` to `n` times
`{m,}`	match at least `m` times
`{,n}`	match up to `n` times (including `0` times)
`{n}`	match exactly `n` times

7. **Spring 2023 Midterm**. The following question is a great practice question with using regex for a common use case in web scraping (searching through HTML).

(8 minutes). Your Web design studio has a style rule that the only void element allowed is "`<br/>`" and it must be spelled exactly that way. However, some of the web pages being generated don't follow the rule: they might contain text like this:

` `	Closing slash omitted.
`<br title="hello">`	Attribute present.
`< br / >`	Extra spaces.
`<hr/>`	Not "br".

Write an extended regular expression (ERE) that you can use to grep for lines containing void elements that don't follow your design studio's rule. Your ERE should not match "
", and it should match all the above examples of undesirable void elements.

There are a lot of possible answers to this question (it is actually impossible to solve generally). One big idea is that the goal is to match all of the possible examples above and then respond in the second half of the question below, indicating where your regex fails to match. The goal is to not perfectly answer your question but to do your best with catching a lot of cases and then arguing for the tradeoffs in the second half of the question. A potential solution is the following.

```
<([^b\/] | b[^r] | br[^\/] | br\/[^>])[^>]*>
```

This misses some edge cases (discussed in next part).
Note: This solution once again escapes / so you can copy and paste into regex101!

(4 minutes). Does your ERE have bugs? That is, are false positives or false negatives possible? A false positive occurs if your ERE matches text that is not prohibited, and a false negative occurs if your ERE fails to match text that is prohibited.

If so, give an example; if not, briefly explain why not.

There is a guarantee that your ERE has bugs. This is because HTML is not a **regular language** (i.e. you can't write a general regular expression to capture all of HTML. The

reasoning is related to finite automata which you will learn more about when you take CS 181). Don't worry about understanding how finite automata works for this class but basically, it's okay if there are edge cases that your regex misses. Your goal here is to practice debugging your regex and find interesting edge cases that your regex fails to capture. A potential answer to the above based on the above answer is. Make sure to include false positives and false negatives for full points.

- **False Positives**: non-empty tags like **<p>**.
- **False Negative**: Incomplete <br/ tag.