UCLA Computer Science 97 Midterm Solutions

Fall 2020

(Instructor: Dr. Paul Eggert)

These solutions are intended to be as comprehensive and instructive as we[1] can make them. They are not exemplars of the level of detail expected of you in the actual midterm.

1. **This problem asks you to write some shell scripts. Your scripts should be simple and clear. They should minimize the use of temporary files; if they do need to create temporary files they should remove the temporaries before exiting. If a script runs into any trouble, it should exit with status 0; otherwise, it should exit with nonzero status [*sic*].[2] Your script can use any shell control structures such as functions and loops, but should limit themselves to a small set of utilities, such as:**

   ```
   [ . break cat cd chmod comm continue cp diff echo env eval
   exit expr false find fmt fold grep head kill ln ls mkdir mv
   printf ps pwd rm rmdir seq shuf sort tail test tr true uniq
   ```

   That is, you should not solve the problem in C++ or Python or some other programming language; just use the shell. If you need any utility not in the above list, briefly justify why it's needed.

   You can write several scripts if you like, and have your scripts invoke each other.

   Along with each question answer, write a brief after-action report explaining how you came up with and debugged the scripts, including any blind alleys you explored.

   The basic goal of these shell scripts will be to test the 'myspell' program that you wrote in Assignment 1. That is, you want to give 'myspell' some input data and compare the resulting output to what it should be, and you want to have lots of test cases to test 'myspell' thoroughly.

   a. **(12 min). Write a shell script 'genspelldata' that generates test cases for 'myspell'. Your script can assume the existence of the file '/usr/share/dict/linux.words' as in Assignment 1. It should take three operands *G B F*, where *G* is the number of good words in the output, *B* is the number of bad words in the output, and *F* is the name of a text file containing bad words, one**

[1] Created by CS 97 Learning Assistants. Contributors: Connie Chen, Timothy Gu, Jamie Liu, Frank Zheng.

[2] This requirement is peculiar, since it is usually the other way around in the Unix world, where exit code 0 signifies success and a non-zero code signifies failure. In fact, Dr. Eggert has admitted this to be a typographical mistake on his part. In the following solutions, we try to stay faithful to this original requirement. When grading this exam, most points should be given if the student solution provides *any* way of distinguishing between successes and errors.

per line. (All numbers are unsigned decimal integers.) Words should be chosen randomly from the respective sources, with repetition.

For example, if the file 'bad.words' contains the four lines

```
notaword
anothernonword
ain't
secretary
```

the command 'genspelldata 5 3 bad.words' might output:

```
tailorless ain't Avicennism notaword mariengroschen
catadrome underbailiff ain't
```

because this output has 5 good words and 3 bad words; two of the bad words are identical, but that sort of thing is expected.

Answer: Let's try to break the problem into digestible chunks. The first part is getting a random word from a file. Looking through the set of utilities available to us, it appears the only utility that provides a source of randomness is '**shuf**', which shuffles its input file. However, shuf itself is not going to be sufficient, since we know that 'shuf' by default does not allow repeated selections. Looking through its documentation, we see some useful options:

- **'-n *count*'**: Output at most *count* lines. By default, all input lines are output.
- **'-r'**: Repeat output values, that is, select with replacement.

Having the capability to generate any number of selections – with repetition – the only remaining challenge for us to somehow weave the good words with the bad words. One may realize we can use the shuf command *again* on the combined list of good and bad words.

Finally, the sample output has spaces rather than newlines. We can use the tr command to replace all newlines with spaces.

Combining all the above remarks, we end up with the following:

```
#!/bin/sh

G="$1"
B="$2"
F="$3"

{
    shuf -r -n "$G" /usr/share/dict/linux.words
    shuf -r -n "$B" "$F"
} | fmt
exit 1
```

Remarks:

- We created variables for each input operand, so that we may refer to them as $G, $B, and $F. You don't have to, and may instead just say $1, $2, and $3 respectively.
- We took care to avoid creating temporary files (as recommended by the instructions) using { } braces, which creates a command grouping. A command grouping executes like a small shell script of its own, with its standard output being piped to another process. However, it may be easier to see what's going on with other equivalents.

  Here's an alternate version that uses temporary files:

  ```
  shuf -r -n "$G" /usr/share/dict/linux.words >tmpgoodwords
  shuf -r -n "$B" "$F" >tmpbadwords
  cat tmpgoodwords tmpbadwords | shuf | tr '\n' ' '
  rm tmpgoodwords tmpbadwords
  ```

  Here's a similar version that uses variables instead of temporary files.

  ```
  good=$(shuf -r -n "$G" /usr/share/dict/linux.words)
  bad=$(shuf -r -n "$B" "$F")
  printf '%s\n%s\n' "$good" "$bad" | shuf | fmt
  ```

  Note that using variables is not the "best" way to write this script. With variables, the entire output of 'shuf' is stored in memory, so the space complexity would be *linear*. Conversely, when redirecting the output of a command group, as shown in the original solution, the output can be streamed. This means that the output is processed as it is generated, resulting in *constant* space complexity. (Knowledge of space complexity is not a prerequisite for this course, so don't worry if this confuses you.)

- The 'fmt' command is used to replace newlines with spaces in the output. It is similar to '`tr '\n' ' '`'.

b. **(10 min). Assuming the shell scripts 'genspelldata' and 'myspell' exist in the same directory, write a shell script 'testmyspell' that takes three[*sic*] unsigned decimal number operands *N G B F* and tests 'myspell' with *N* different randomly chosen test cases, each containing *G* good words and *B* bad words (the latter taken from the file *F*). For example, 'testmyspell 1000 5 3 bad.words' should test 'myspell' 1000 times, each with a randomly-chosen test case generated by 'genspelldata 5 3 test.words'.**

Answer: It's not immediately clear what scope of the "testing" is required for this question. For this sample solution, we will interpret it as just making sure every bad word output by 'myspell' comes from the provided file *F*.

```
#!/bin/sh

N="$1"
G="$2"
B="$3"
F="$4"

for i in $(seq "$N")
do
   ./genspelldata "$G" "$B" "$F" |
       ./myspell |
       while IFS= read -r badword
       do
           grep -Fx "$badword" "$F" >/dev/null
           case "$?" in
           0)  # $badword found in $F
               ;;
           1)  # $badword not found in $F
               echo "myspell printed '$badword' as a bad" \
                    "word, but it is not found in $F"
               exit 0
               ;;
           *)  # something else went wrong
               exit 0
               ;;
           esac
       done
done
exit 1
```

Remarks:

- The 'grep' chunk of the solution was derived from an answer to this Stack Overflow question: "How to test if string exists in file with Bash?". No, we do not know how to do that off the top of our heads.
- The 'case' statement is similar to a 'switch' statement in C++ or Java. We use it here to distinguish between the cases where $badword was not found in $F, and where 'grep' failed (e.g., $F does not exist). In an exam setting, it's completely okay to replace it with an 'if' statement.
- We used a `for-in` loop with 'seq' to run the test *N* times. Other options, either less concise or less portable (only works with some shell implementations), are possible.

- We used ‘`IFS= read -r`’ in order to read lines of input. This is preferable over a `for-in` loop, just in case a word ‘myspell’ deems misspelled contains a space. See some additional discussions in this area here at Greg Wooledge’s Wiki.
- Here’s a more concise solution that does not use a ‘case’ statement or the ‘read’ command, but nevertheless would probably get full marks for the question. Note that the use of variables makes it slightly less ideal than the provided solution (see the remark under (1a) for why this is the case).

```
for i in $(seq "$N")
do
   output=$(./genspelldata "$G" "$B" "$F" | ./myspell)
   for badword in $output
   do
       if grep -Fxq "$badword" "$F"
       then
           echo "myspell printed ‘$badword’ as a bad" \
                "word, but it is not found in $F"
           exit 0
       fi
   done
done
exit 1
```

c. **(10 min). Suppose you are maintaining ‘genspelldata’ and are worried that the changes that you make might break it. You want to have some regression tests for ‘genspelldata’, so that you can test it after changing it, and make sure that it outputs exactly the same thing after the change as before the change. Describe and implement extensions to the behavior of ‘genspelldata’ to make it regression-testable in this sense, while keeping the functionality that the script already has. Your extensions should add an option or options to ‘genspelldata’ to make it regression-testable. (Hint: see full documentation for ‘shuf’.)**

Answer: We may notice that our current ‘genspelldata’ script often creates different output each time we run it. This is since when we run ‘shuf’, it’s using a random number generator built into the operating system. After reading the documentation of ‘shuf’, we notice the ‘`--random-source=`*`file`*’ option, which makes ‘shuf’ “use *file* as a source of random data used to determine which permutation to generate.” This means that as long as we specify the same file with this option, ‘shuf’ will always give the same deterministic output.

There are a few different ways we can extend ‘genspelldata’ to set this option in certain cases. The sample solution we provide here allows regression-testable output when called as ‘`genspelldata` *`G B F`* `--random-source=`*`file`*’.

```
#!/bin/sh

G="$1"
B="$2"
F="$3"
opts="$4"

{
   shuf $opts -r -n "$G" /usr/share/dict/linux.words
   shuf $opts -r -n "$B" "$F"
} | shuf $opts | fmt
```

A few changes were made to (1a). First, we assign $4 to the variable $opts. Then, when invoking 'shuf', we make sure to add $opts to the command. Note: by not actually parsing the $opts ourselves and leaving it to 'shuf' to interpret, we were able to save some work.

Remark: The careful reader may notice that this solution does not work in two cases:

- if *file* contains a space, or
- if one tries to use the '`--random-source` *`file`*' syntax, with a space rather than a '=' separating the option and the value.

An even better solution here is to use the special $@ variable, which fixes both of the above issues and also allows additional 'shuf' options to be specified – though it is beyond the range of expected shell knowledge.

```
#!/bin/sh

G="$1"
B="$2"
F="$3"
shift 3

{
   shuf "$@" -r -n "$G" /usr/share/dict/linux.words
   shuf "$@" -r -n "$B" "$F"
} | shuf "$@" | fmt
```

2. **(14 min). Reimplement the 'genspelldata' command in Python, using only the argparse, random, string, and sys modules. (See (1a) for the specification of 'genspelldata'.) Or, if it's not practical to implement 'genspelldata' that way, explain why not, implement it as best as you can in Python some other way, and explain any shortcomings in your approach. Either way, make your solution as simple and clear as you can.**

   Answer: Let us again break this into steps. From assignments, we know that Python allows reading files using the 'open()' function. However, we also need to look for a way to shuffle a list, in an analogous way to how we used the 'shuf' command. A look at the documentation of the random module turned up 'random.shuffle()', which shuffles the input list (analogous to '`shuf`'); as well as 'random.choices()', which selects *k* items from the input with replacement (analogous to '`shuf -r -n` *k*').

Here is a Python solution that implements 'genspelldata' in Python with only a subset of the specified modules. All underlined words and symbols are links to the Python documentation. The overall algorithm is identical to our solution in (1a).

```
#!/usr/bin/env python3
import argparse
import random

parser = argparse.ArgumentParser(
    description='Generate random test cases for myspell.')
parser.add_argument('good', metavar='G', type=int,
                    help='number of good words to output')
parser.add_argument('bad', metavar='B', type=int,
                    help='number of bad words to output')
parser.add_argument('badfile', metavar='F', type=str,
                    help='file containing a list of bad words')

args = parser.parse_args()

all_good_words = []
with open('/usr/share/dict/linux.words', 'r') as f:
   for good_word in f:
       all_good_words.append(good_word.rstrip('\r\n'))

all_bad_words = []
with open(args.badfile, 'r') as f:
   for bad_word in f:
       all_bad_words.append(bad_word.rstrip('\r\n'))

selected_good_words = random.choices(all_good_words, k=args.good)
selected_bad_words = random.choices(all_bad_words, k=args.bad)

all_selected_words = [*selected_bad_words, *selected_good_words]
random.shuffle(all_selected_words)
```

```
print(' '.join(all_selected_words))
```

3. **Consider the following Elisp code:**

```
(defun delete-horizontal-space (&optional backward-only)
  "Delete all spaces and tabs around point.
If BACKWARD-ONLY is non-nil, delete them only before point."
  (interactive "*P")
  (let ((orig-pos (point)))
    (delete-region
     (if backward-only
         orig-pos
       (progn
         (skip-chars-forward " \t")
         (constrain-to-field nil orig-pos t)))
     (progn
       (skip-chars-backward " \t")
       (constrain-to-field nil orig-pos)))))
```

   a. **(6 min). Give two ways to call this function from an Emacs session in which you want to delete all spaces and tabs around the cursor. Prefer convenience.**

      Answer: One way is
      `M-x delete-horizontal-space`.

      When you run it this way, Emacs helpfully informs you that "you can run the command 'delete-horizontal-space' with `M-\`" in the echo area. Indeed, `M-\` is a second way of doing the same thing.

      A third possible answer is to bind another key combination to the command using
      `M-x global-set-key RET` *`key`* `delete-horizontal-space RET`
      where *key* is the desired key binding. Afterwards, you can just use *key* to run the command.

   b. **(6 min). Extend the function so that the caller can optionally delete spaces and tabs only **after** the current position.**

      Answer: The trick here is to observe how the code was written for the 'backward-only' option, and do something similar for a new 'forward-only' argument. In particular, we modify the ‘end’ parameter of the delete-region function, setting it to ‘orig-pos’ if ‘forward-only’ is non-nil.

```
(defun delete-horizontal-space (&optional
                                backward-only
                                forward-only)
  "Delete all spaces and tabs around point.
If BACKWARD-ONLY is non-nil, delete them only before point.
If FORWARD-ONLY is non-nil, delete them only after point."
  (interactive "*P")
  (let ((orig-pos (point)))
    (delete-region
     (if backward-only
         orig-pos
       (progn
         (skip-chars-forward " \t")
         (constrain-to-field nil orig-pos t)))
     (if forward-only
         orig-pos
       (progn
         (skip-chars-backward " \t")
         (constrain-to-field nil orig-pos))))))
```

Remark: This solution works, but is rather inconvenient to use in practice, since `M-x` and `M-\` only allow one argument to be specified – the prefix argument. A more idiomatic approach would be to use the single argument to determine the direction of deletion:

```
(defun delete-horizontal-space (&optional dir)
  "Delete all spaces and tabs around point.
If DIR is positive, delete them only before point.
If DIR is - or negative, delete them only after point."
  (interactive "*P")
  (let ((orig-pos (point)))
    (delete-region
     (if (and dir (> (prefix-numeric-value dir) 0))
         orig-pos
       (progn
         (skip-chars-forward " \t")
```

```
            (constrain-to-field nil orig-pos t)))
        (if (< (prefix-numeric-value dir) 0)
            orig-pos
          (progn
            (skip-chars-backward " \t")
            (constrain-to-field nil orig-pos))))))
```

With this version, one can now easily invoke all three possibilities:

`M-\`	delete both directions
`C-u M-\`	delete backward only
`C-u - M-\`	delete forward only

c. (6 min). Modify the original (non-extended) function so that it does not worry about fields, i.e., it does not call 'constrain-to-field' and always behaves as if the entire buffer were one field.

Answer: According to its documentation, (constrain-to-field *new-pos old-pos*) "returns the position closest to *new-pos* that is in the same field as *old-pos*." Assuming *new-pos* and *old-pos* are always in the same field, then 'constrain-to-field' will always return *new-pos* – if it's non-nil.

However, in this script *new-pos* is always set to nil. The documentation further explains that "if *new-pos* is nil, then constrain-to-field uses the value of point instead." So we actually need to replace the 'constrain-to-field' call with a call to 'point'.

```
(defun delete-horizontal-space (&optional backward-only)
  "Delete all spaces and tabs around point.
If BACKWARD-ONLY is non-nil, delete them only before point."
  (interactive "*P")
  (let ((orig-pos (point)))
    (delete-region
     (if backward-only
         orig-pos
       (progn
         (skip-chars-forward " \t")
         (point)))
     (progn
       (skip-chars-backward " \t")
       (point)))))
```

4. **(9 min). Explain how the 'wget' shell command fits into the client-server computing model as exemplified by React. For example, what are the pros and cons of using 'wget' to retrieve data from a system built by React?**

   Answer: Recall that in a client–server computing model, there are two parts:
   1. The **servers** are the central part of an application. It is generally the source of information that will be displayed on users' computers.
   2. The **clients** are pieces peripheral to the application. Clients typically request resources from the servers, then display these resources to the user. If the client is interactive, it generally sends user interactions as updates to the servers.

   In this model, the 'wget' shell command serves as a client. It retrieves information by sending HTTP(S) requests to some server. For instance, the shell command

   ```
   wget https://web.cs.ucla.edu/classes/fall20/cs97-1/index.html
   ```

   will send an HTTPS request to the CS 97 website, then save the response body to a file. In this example, the CS 97 website acts as the server, containing information such as the class syllabus and assignments, whereas 'wget' acts as the client.

   In a system built using React, the HTML file tends to be quite barebones, with most of the web page's content being loaded dynamically through JavaScript. In fact, the only content in the index.html file from 'create-react-app' is (emphasis ours):

   ```
   <body>
     <noscript>
       You need to enable JavaScript to run this app.
     </noscript>
     <div id="root"></div>
     <!--
       This HTML file is a template.
       If you open it directly in the browser, you will see an
       empty page. ...
     -->
   </body>
   ```

   Additionally, the JavaScript code frequently uses techniques such as Ajax and JSON to dynamically communicate with the server and update the web page based on user interaction.

With this background, some pros of ‘wget’ might be:

- It allows you to inspect the source of an HTML file that would normally be rendered by a browser.
- It allows debugging Ajax server endpoints in a simple way, without having to click through a series of buttons on the React-rendered UI.
- It is available in a wide range of platforms, including many that major web browsers do not support.
- It frees you from having to implement the HTTP and TLS protocols yourself.

Some cons of ‘wget’ are:

- Since ‘wget’ does not include a JavaScript interpreter, it is unable to display the bulk of the content of a typical React-based web page.[3]

  (In contrast, using ‘wget’ to request the CS 97 website, which does not use React, will allow you to look through a mostly complete HTML tree to understand what will be displayed.)
- ‘Wget’ does not include an HTML renderer, so one cannot visualize the HTML file with it.
- ‘Wget’ cannot readily be integrated into an existing C application as a library, unlike ‘curl’ with its libcurl component. ‘Wget’ also has a more restrictive copyright license than ‘curl’ and several major web browsers.

5. **Consider the following JSX function taken from the React tutorial used as the basis of Homework 3:**

```
function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

   a. **(8 min). Explain what this function is for and how it works. Give the type of all the identifiers and expressions used in this JSX code.**

      Answer: This function serves as a “function component” in React. When a React component doesn’t need to maintain its own state, we can write it as a function

[3] This disadvantage could be mitigated if React server-side rendering is used. In that case, the generated HTML file is close to what React would render upon application startup in a web browser. There exist frameworks such as Gatsby that use this feature extensively.

instead of a class (aside: though with the introduction of hooks, function components can now also maintain state). At a high level, the above 'Square' function renders a square in a tic-tac-toe board.

The 'Square' function has a single parameter, `props`, which is a JavaScript object that contains `onClick` and `value` properties. The value of `props.onClick` should be a function, and the value of `props.value` should be a string.

The function then returns a JSX `button` element. The `button` element has a `className` attribute, set to the string "`square`," and an `onClick` attribute set to the function `props.onClick`. It also has a single child, the string `props.value`.

When React renders this component, this JSX `button` element will ultimately be translated into an HTML `button` element. This button will have the HTML attribute `class` with value "`square`," which with proper CSS will style the button to look like a square. The child of this button element will be a text node with the value of `props.value`, which will likely either be an "X," "O," or space. When the button is clicked, it will call the function `props.onClick`, which will update the game state.

b. **(4 min). Write a JSX class that behaves the same way as this function, when used as a React component.**

Answer: We move the body of the function into the 'render' method of a new class that extends `React.Component`. Additionally, the 'props' argument now becomes the 'this.props' property.

Note that the solution to this question is also available through the React Tutorial (section Lifting State Up, search for "the Square component looks like this").

```
class Square extends React.Component {
  render() {
    return (
      <button className="square" onClick={this.props.onClick}>
        {this.props.value}
      </button>
    );
  }
}
```

6. **(15 min). Briefly compare and contrast the role of dependencies and parallelism during development, during builds, and during installation. What do the dependencies have in common, and how do they differ? Give an example of a development dependency in your ongoing class project, and give an example of a build and of an installation dependency; take the latter two examples from the assignments.**

   Answer: **Development dependencies** refer to the scenarios in project management, where a component in an application should be written before another.

   Build dependencies refer to a piece of software the compiler or build tool needs to build another piece of software. In C and C++, dependencies are generally files (like .o object files).

   Installation dependencies refer to pieces of software that need to be installed for another piece of software to work. This type of dependency generally refers to packages or libraries.

   Here are some similarities:

   - A key takeaway is that in all three situations, *dependencies inhibit parallelism.*
     - If we have fifteen files to be written, but with none depending on another, we can write all fifteen files in parallel – an embarrassingly parallel situation. However, as soon as we introduce some dependency, say file *B* contains the number of lines of file *A,* to generate file *B* we have to *wait* until file *A* has been fully written.
   - All three situations involve a piece of software that cannot work without another piece of software.
     - In the words of development dependencies, if one were to write some code that saves pictures to a database, without first writing the code to connect to the database, the picture-saving code would not function.
   - In all three situations, we have long dependency chains.
     - In the words of npm installation dependencies, React depends on 'loose-envify', which then depends on 'js-tokens'. One can imagine arbitrarily long dependency chains, leading to memes about the topic.

   Differences:

   - Generally, development dependencies and build dependencies are stricter than npm installation dependencies.
     - npm is able to install most packages in parallel, even for packages that technically depend on others. This is since for most packages installation consists solely of extracting a tarball archive. If package *A* depends on package *B* but is installed before *B,* it would not function; but npm can still install them out of order.

    - In contrast, there is relatively little flexibility in development dependencies. There is little sense in creating a comment platform for a video-sharing website, if the video-sharing component has not been built.
    - This is just one possible hierarchy of strictness that we thought of. Others could be acceptable if sufficiently justified.
- While build and installation dependencies mostly arise out of technical considerations, development dependencies can have more “human” concerns. If our team is trying to bring our product to the market quickly, we may not want to depend on another team finishing their project – even if it makes technical sense to do so.

Examples:

Development dependency: (examples from other parts of our answer)

- In an online video-sharing website, for developing a comment platform to make sense, the team should already have developed features to store and play videos.
- Before writing code to save pictures to a database, one would need to first write the code to talk to a database.

Build dependency:

- The React app we created in Assignment 3 using ‘create-react-app’ functions in the following way. It bundles all JavaScript files in the project into a single JavaScript file through a tool called ‘webpack’. However, before the bundling could happen, JSX source files in your React app must first be transpiled to JavaScript using Babel. Here, JSX transpilation is a *dependency* of JavaScript bundling.
- In previous Computer Science classes like Computer Science 31, sometimes to create the executable, one has to compile several C++ files to `.o` files first.

Installation dependency:

- React version 17.0.1, for example, depends on ‘loose-envify’ and ‘object-assign’ packages. Using the ‘react’ package requires those other packages to be installed.
- Your application in Assignment 3 depends on React to work.
- To create a React application using ‘create-react-app’, you must first have Node.js installed.