Monash University
Faculty of Information Technology
2^{nd} Semester 2018

FIT2014
Assignment 1
Linux tools, regular expressions, induction
DUE: 11:55pm, Friday 17 August 2018

How to manage this assignment

- Do as much as possible of it *before* your week 4 prac class. There will not be time during the class itself to do the assignment from scratch; there will only be time to get some help and clarification.

Instructions

Please read these instructions carefully **before** you attempt the assessment:

- To begin working on the assignment, download the `asgn1.tar.gz` workbench from Moodle and unpack it. Be sure to configure **and** test the submission-builder before you begin working. Refer to Lab 0 for a reminder on how to do all of these tasks.
- The workbench provides locations and names for all solution files. These will be empty, needing replacement. Do **not** add or remove files from the workbench. All solutions must be submitted to Moodle by the deadline above, and must be packaged using the workbench's makefile system.
- Solutions to written questions must be submitted as PDF documents. You can create a PDF file by scanning your **legible** (use a pen, write carefully, etc.) hand-written solutions, or by directly typing-up your solutions on a computer.
- Start work on this assignment early, and try to make substantial progress **before** your scheduled lab class: your tutor won't have time to help you start from scratch. Before you attempt any problem—or seek help on how to do it—be sure to read and understand the question, as well as any accompanying code.
- To aid the marking process, you must adhere to **all** naming conventions that appear in the assignment materials, including files, directories, code, and mathematics. Not doing so will cause your submission to incur a one-day late-penalty (in addition to any other late-penalties you might have). Be sure to check your work carefully.

Your submission must include:

- an `awk` script, `prob1`, for Problem 1;
- a `sed` script, `build-awk`, for Problem 2(a);
- the generated `awk` script `analyseGrLa`, for Problem 2(b);
- a file `inputFileOfWords` being the word list you used for Problem 2(c);
- a file `prob2c.pdf` with your solution and required information for Problem 2(c);
- a file `prob3.pdf` with your solution to Problem 3.

To submit your work, simply enter the command '`make`' from within the `asgn1` directory, and then submit the resulting `.tar.gz` file to Moodle. As last time, make sure that you have tested the submission mechanism and that you understand the effect of `make` on your directory tree.

Introduction to the Assignment

In Lab 0, you met the stream editor `sed`, which detects and replaces certain types of *patterns* in text, processing one line at a time. These patterns are actually specified by *regular expressions*. You will use `sed` again in Problem 2 of this Assignment, to transform a list of strings into a list of *instructions*, with one instruction line for each string.

You will also learn about `awk`, which is a simple programming language that is widely used in Unix/Linux systems and which also uses regular expressions. After an introductory exercise (Problem 1), you will construct an `awk` program to study the frequency of Greek and Latin roots (i.e., word components) in English text.

Finally, Problem 3 is about applying induction to a problem of counting strings.

Introduction to `awk`

In an `awk` program, each line has the form

/*pattern*/   { *action* }

where the *pattern* is a regular expression (or certain other special patterns) and the *action* is an instruction that specifies what to do with any line that contains a match for the *pattern*. The *action* (and the {. . . } around it) can be omitted, in which case any line that matches the *pattern* is printed.

Once you have written your program, it does not need to be compiled. It can be executed directly, by using the `awk` command in Linux:

```
$ awk -f programName   inputFileName
```

Your program is then executed on an input file in the following way.

```
//   Initially, we're at the start of the input file, and haven't read any of it yet.
If the program has a line with the special pattern BEGIN, then
    do the action specified for this pattern.
Main loop, going through the input file:
{
    inputLine  :=  next line of input file
    Go to the start of the program.
    Inner loop, going through the program:
    {
        programLine  :=  next line of program (but ignore any BEGIN and END lines)
        if  inputLine  contains a string that matches the pattern in programLine, then
            if there is an action specified in the programLine, then
            {
                do this action
            }
            else
                just print inputLine        //   it goes to standard output
    }
}
If the program has a line with the special pattern END, then
    do the action specified for this pattern.
```

Any output is sent to standard output.

You should read about the basics of `awk`, including the way it represents regular expressions and the main instruction types used in its actions. Any of the following sources should be a reasonable place to start:

- A. V. Aho, B. W. Kernighan and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, New York, 1988.
  (The first few sections of Chapter 1 should have most of what you need, but be aware also of the regular expression specification on p28.)
- `http://www.grymoire.com/Unix/Awk.html`
- `http://www.hcs.harvard.edu/~dholland/computers/awk.html` (a good, short introduction)
- the Wikipedia article looks ok
- the `awk` manpage.

Problem 1. [4 marks]

Suppose you have a file of English words, with one word (and no other characters) on each line of the file.

(a) Write a regular expression (in the format used by `awk`) that matches only strings that both start and end with `an`, and occupy an entire line.[1]

(b) Write an `awk` script that takes an input file of words and determines the number of words in the file that both start and end with `an`.

Introduction to the next task

In Problem 2, you will not need to write much code yourself: maybe about half-a-dozen lines, or less, of `sed`! But your `sed` code will generate thousands of lines of `awk` code, which in turn will be used to analyse the frequency of Greek and Latin roots in English words.

For the purposes of this Problem, a *root* of an English word is a subword that is derived from another language, has meaning in its own right, and brings some of that meaning into the English word containing it. This definition is looser than that used in linguistics, and embraces some related concepts such as stems, prefixes, and suffixes.

You'll find a list of Greek and Latin roots for English words in the file

```
roots-in-English-from-Wikipedia-simple
```

which you'll find in Moodle under week 4. This file was obtained from `https://en.wikipedia.org/wiki/List_of_Greek_and_Latin_roots_in_English` by removal of information we don't need, some reformatting, and some cleaning up. Every line of this file contains, in order: the root, a space, a tab, a single word "`Greek`" or "`Latin`" stating which language the root comes from, and another space. The root is given as a string of lower-case characters with a hyphen at one or both ends. For example, here are four lines from the file.[2]

```
abac-   Greek
-quir-  Latin
-oid    Greek
vesper-         Latin
```

A hyphen indicates that, when this root appears in an English word, there must be more letters on the side of the word indicated by the hyphen. A side without a hyphen must be the start or end of the English word. So, the root `abac-` must appear at the start of the word and must be followed

[1] Part (a) does not need to be submitted as a separate file, since its answer is included in (b).

[2] The tabs may not survive translation into this PDF document, but they'll be there in the file! The imperfect alignment is an effect of varying word length together with the tab in each line; this is not a concern.

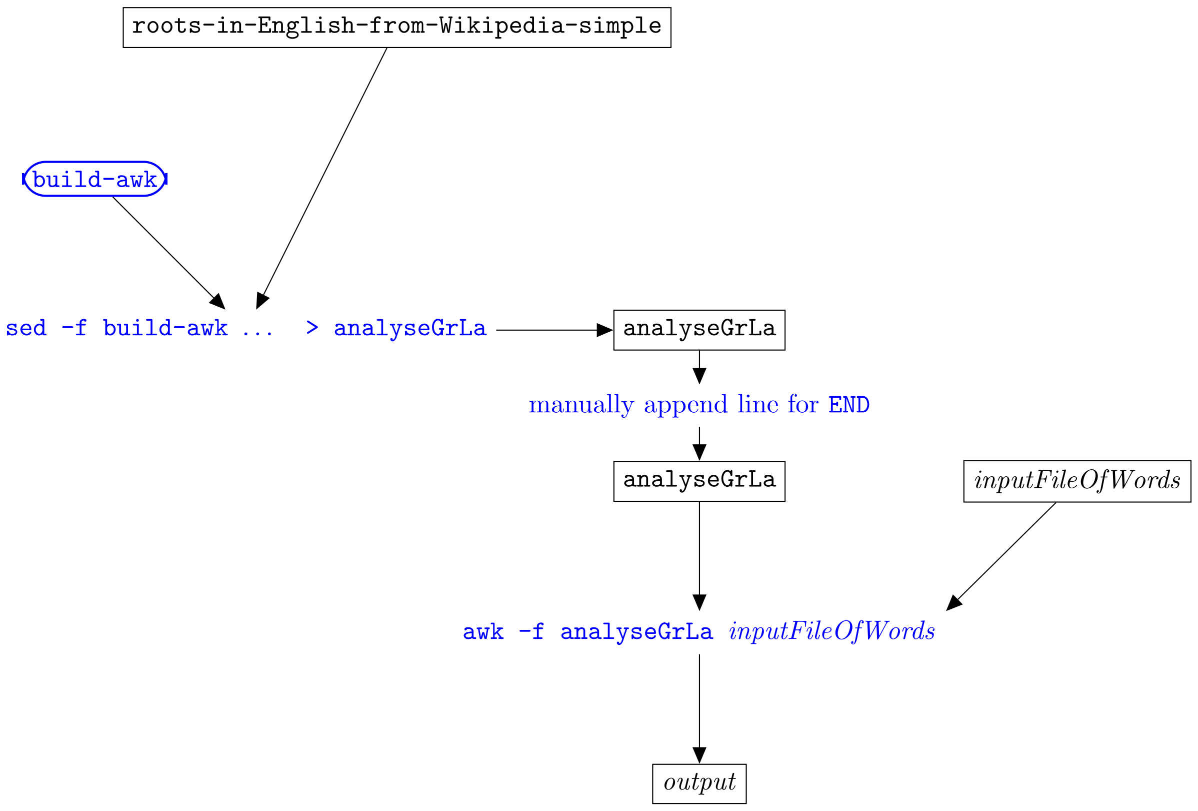
by something else, as in abac*us*; the root `-oid` must appear at the end of the word and must be preceded by something else, as in *aster*oid; and the root `-quir-` must be in the middle of the word, with something else on each side, as in *en*quir*y*.[3]

Our aim is to take a large file of English words and estimate the number of words with Greek roots and the number of words with Latin roots. Our method will be as follows.

1. Use a `sed` script, which we'll call `build-awk`, to convert the file `roots-in-English-from-Wikipedia-simple` to an `awk` script.

2. Run the generated `awk` script on the input file of English words to produce a count of the number of words with Greek roots, and the number of words with Latin roots, in the file.

Our plan, with the appropriate roles and relationships for the files, is shown in the diagram below.



roots-in-English-from-Wikipedia-simple
build-awk
sed -f build-awk ... > analyseGrLa
analyseGrLa
manually append line for END
analyseGrLa
inputFileOfWords
awk -f analyseGrLa inputFileOfWords
output

Problem 2. [10 marks]

(a) Write the `sed` script, `build-awk`:

Write a `sed` script, `build-awk`, to convert each line of the list of roots, `roots-in-English-from-Wikipedia-simple`, into a line of an `awk` script, `analyseGrLa`, that can be used to detect words with Greek or Latin roots and find how many there are of each type.

You'll probably need to write three lines of `sed`: one for roots that are prefixes (e.g., `abac-`); one for roots that are suffixes (e.g., `-oid`), and one for roots that appear in the middle of a word (e.g., `-quir-`).

Each root is to be converted to an appropriate regular expression (between forward-slashes, /.../, in usual `awk` format) with an appropriate action (between braces, {...}). The regular expression

[3]All this is a simplification of how these roots can be extended to words, but we'll make these assumptions for this Assignment.

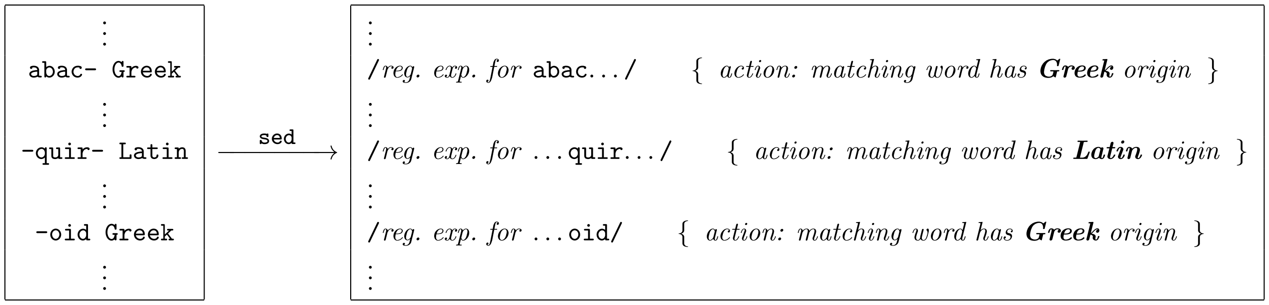
must match precisely those words that contain the root in the required way. (The root uses hyphens to indicate the allowed containments, but you'll have to use regular expression machinery.) If a match occurs, then the appropriate total of word types must be updated. If the root is Greek, we update the Greek total; if the root is Latin, we update the Latin total.

The method of counting here is somewhat imperfect, since some words may have both Greek and Latin elements and be counted as both, while others may contain several roots from one of the languages and so count several times towards the total for that language. We will not worry about these particular imperfections; you just have to count the number of times a word in the input file matches a root for that language, and report that total for each of the two languages. But there are bonus marks for a more sophisticated approach; see below.

The effect of using `sed -f build-awk` is illustrated in the following table. (Note, though, that we just call `sed` *once*; that's enough to do *all* the line transformations.)

File with all the roots: `roots-in-English`...

Generated `awk` script: `analyseGrLa`



⋮
abac- Greek
⋮
-quir- Latin
⋮
-oid Greek
⋮
sed
⋮
/*reg. exp. for* abac.../ { *action: matching word has* ***Greek*** *origin* }
⋮
/*reg. exp. for* ...quir.../ { *action: matching word has* ***Latin*** *origin* }
⋮
/*reg. exp. for* ...oid/ { *action: matching word has* ***Greek*** *origin* }
⋮

Once the `awk` script is generated, it will need an extra line on the end, using the special pattern `END`, to report the totals. You can add this manually. An efficient way of doing this is to write a one-line file (called `endline`, say), containing `END` followed by its corresponding action in braces, and manually append this to your awk script `analyseGrLa` whenever you generate that script from `sed -f build-awk`.

(b) `analyseGrLa`**:**
Provide the generated `awk` script `analyseGrLa`, *including* the extra line you added at the end, as your answer to this part.

(c) Using your `awk` script:
Choose a long English text file (at least 10,000 words), with one word per line, and no non-alphabetic characters. This could be either

- a standard word list, with as many words as possible; or
- the words from a long English text file, such as the one you used for Lab 0, Part 6, Exercise 1.

These two options are different in nature; the former takes no account of how frequent words are in typical English text, and has no repetitions; the latter has many words that reappear many times through the text, so words of high frequency will make a greater contribution to the results. Either option is fine. It will be interesting to compare the results from the two kinds of data. But, for your submission, you only need to do one of the two options.

You will most likely need to clean up your chosen file somewhat, to remove nonalphabetic characters, put one word on each line, etc. This can be done using the tools and techniques you developed in Lab 0. Indeed, you may find that you already have a suitable file from your work for that Lab.

Run your `awk` script, `analyseGrLa`, on your *inputFileOfWords*. Your submission for this part must include:

- specification of the source of the file (e.g., URL with date downloaded);

- the number of words in the file;
- a short description of what changes you made to the file and what tool was used (e.g., removed punctuation using `tr`; line breaks between words using `sed`; ... etc.);
- the output, which should just be one line, or only a few lines.

All this information, including the output, is to be placed in a single file.

Bonus Marks if you can get your `awk` script to find, for each word in *inputFileOfWords*, its Greek and Latin proportions, defined as follows:

If the word has at least one Greek or Latin roots, then

$$\begin{aligned}\text{Greek proportion} &= \frac{\text{number of Greek roots it has}}{\text{total number of roots it has}},\\ \text{Latin proportion} &= \frac{\text{number of Latin roots it has}}{\text{total number of roots it has}},\\ \text{total number of roots it has} &= \text{number of Greek roots it has} + \text{number of Latin roots it has.}\end{aligned}$$

In this case, the two proportions must add up to 1.

Otherwise — i.e., if the word has no Greek or Latin roots at all — then

$$\begin{aligned}\text{Greek proportion} &= 0,\\ \text{Latin proportion} &= 0.\end{aligned}$$

Your `awk` script must add up all the Greek proportions, and all the Latin proportions, and output them at the end. It must also output the number of words with no Greek or Latin root.

Problem 3. [6 marks]

This problem is about simple expressions describing sums and differences of non-negative integers, without parentheses.

A *transaction string* is a string over the alphabet {`0`,`1`,`2`,`3`,`4`,`5`,`6`,`7`,`8`,`9`,`+`,`-`} in which

- the operations `+`, `-` never occur consecutively, and
- the last symbol is not an operation.

In this exercise, we allow positive integers to start with `0`, for simplicity.

Examples of transaction strings include: `000`, `-5`, `25+144-169`, `-1000-0+007`.

Examples of strings that are *not* transaction strings include: `153+46+-44`, `64---46`, `+`, `99+`. The first two of these have two consecutive operations, while the last two end in an operation.

We will prove an upper bound on the number of transaction strings of length n. Since our alphabet has size 12, we have the naive upper bound 12^n. But we will do better than that.

Let a_n be the number of transaction strings of length n.

(a) Find a_1, a_2 and a_3.

(b) Give (with justification) an expression for a_n in terms of a_{n-1} and a_{n-2}, that works for all $n \geq 3$.

(c) Prove, by induction on n, that $a_n \leq 11.8^n$ for all n.

FOR BONUS MARKS: see if you can use a lower number instead of 11.8 here, and still get the proof to work. How low can you go?