CS 214 Spring 2024
Project II: Spelling checker

David Menendez

Due: March 8, at 11:59 PM (ET)

For this assignment, you and your partner will write a program that detects and reports spelling errors in one or more text files. This project will give you practice using the Posix functions for reading and writing files and for traversing directories.

1 Program

Write a program `spchk` that reads a dictionary file and checks the words in one or more text files to see if any are not listed in the dictionary. The first argument to `spchk` will be a path to a dictionary file. Subsequent arguments will be paths to text files or directories.

1.1 Dictionary file

A dictionary file contains one or more "correct" words in lexicographic order, one per line. Words are sorted with case sensitivity, according to ASCII values, so "Zygote" appears before "a". On iLab machines, the file `/usr/share/dict/words` is a large dictionary containing more than a hundred thousand words.

1.2 Text files

For our purposes, a text file contains bytes encoding characters using the ASCII encoding. A text file can be broken into *lines*, which are a sequence of characters ending with a newline character. These lines may contain *words*, which we will consider a sequence of non-whitespace characters.

We number all the lines in a file, starting from one. Each character in a line has a *column number*, which also starts from one. We will want to track the line and column number for each word, defined as the column number of the word's first character.

Trailing punctuation To avoid problems with common sentence punctuation, we ignore punctuation marks occurring at the end of a word. Similarly, we ignore quotation marks (`'` and `"`) and brackets (`(` and `[` and `{`) at the start of a word.

Hyphens A hyphenated word contains two or more smaller words separated by hyphens (`-`). These are correctly spelled if all the component words are correctly spelled.

Capitalization We allow up to three variations of a word based on capitalization: regular, initial capital, and all capitals. That is, if our dictionary contains “hello”, we would consider “hello”, “Hello”, and “HELLO” to be correctly spelled, but not “HeLlO”.

Note that capital letters in the dictionary *must* be capital: if the dictionary contains “MacDonald”, we would accept “MACDONALD” but not “Macdonald” or “macdonald”.

1.3 Directories

When `spchk` is given a directory name as an argument, it will perform a recursive directory traversal and check spelling in all files whose names end with “.txt”, but ignoring any files or directories whose names begin with “.”.

We do not require `spchk` to check the files in any particular order, but all specified files must be checked.

(Note that the requirements to ignore files beginning with a period and not ending with “.txt” only apply to directory traversal, not to files in the argument list.)

1.4 Reporting errors

Each time `spchk` finds an incorrect word, it will report the word along with the file containing the word and the line and column number.

For example:

```
$ spchk my_dict file1 my_files
file1 (35,8): foom
my_files/bad.txt (1,1): Badd
my_files/baz/bar.txt (8,19): almost-correkt
```

Please follow this format exactly.

Print an error message if a file cannot be opened.

`spchk` should exit with status `EXIT_SUCCESS` if all files could be opened and contained no incorrect words, and `EXIT_FAILURE` otherwise.

2 Coding

We can break `spchk` into three major components:

1. Finding and opening all the specified files, including directory traversal
2. Reading the file and generating a sequence of position-annotated words
3. Checking whether a word is contained in the dictionary

You are encouraged to design your code so that these components can be written and tested separately.

You should devise a data structure to represent the dictionary. You will need to do a lookup for each word in the text documents being checked, so choose a data structure that permits quick lookups. As the dictionary is not modified once created, an array with binary search can be effective, but you are free to use trees, hash-tables, or even more powerful structures such as tries. Ideally, your code will have acceptable performance even when using the large system dictionary.

You *must* use `read()` to read the dictionary and text files. No other file reading functions are permitted.

Use of `write()` for printing error messages is suggested, but not required.

3 Testing

Your testing plan should be designed to catch bugs, but also to give confidence that your program does not contain bugs. Consider the requirements for `spchk` and use these to design scenarios where `spchk` should exhibit some known behavior.

Describe your test scenarios in your README, including which problems they are intended to detect.

4 Submission

Submit your source code, Makefile, and README in a Tar archive, along with any test files and dictionaries you mention in your README. The root of the archive should be a directory called P2 (case-sensitive).

Ideally, we should be able to extract your archive, move into P2, call make, and being executing `spchk`. That is,

```
$ tar -xf your-p2.tar
$ cd P2
$ make
$ ./spchk ../dict ../testfile
```

README In your README, note the names and NetIDs for both partners. Describe any notes on your design, or limitations if you were not able to implement everything. Also describe your requirements, test scenarios, and test cases.

5 Grading

Grading will be based on

- Correctness: whether your library operates as intended
- Design: the clarity and robustness of your code, including modularity, error checking, and documentation
- The thoroughness and quality of your test plan