1 About the project

This project is a direct assessment of the skills you acquired by attending the 21 hours C/UNIX initiation course. This project is meant to be done by groups of 2 people, no more. If the class contains an odd number of student then one group of three persons will be allowed. Cheating will be penalized by multiplying your grade by 0.5. Please read the subject very carefully as it contains a lot of details on the expected outputs and behaviors.

You will realize a simple but functional spell checker in C. The program is able to point out most of the spelling mistakes or typographies made in a regular English text.

Hopefully this project will leverage sufficient interest for you to study more on this topic and open some bigger career opportunities.

— Nahim

2 Overview and behavior

This subject will guide you through the realization of a C program named *spell*, referring to its spell checking behavior.

2.1 Spell checker

A spell checker is a computer program which checks the spelling of words in files of text, typically by comparison with a stored list of words. The following extract contains spelling mistake and grammatical mistakes. Only the spelling mistakes would be detected by *spell*.

> In **deeling** with students on the **hih**-school level - that is, the second, third, and forth year of high school - we must bare in mind that to some degree they are at a **dificult sychological** stage, **generaly** called **adolesence**.

“A Language Teacher’s Guide” E.A Meras

2.2 Usage and command line

Without arguments *spell* gives the correct format for the input as follows.

```
$ ./spell
Usage: ./spell dictionary file [file [...]]
```

Listing 1: Spell help message

This usage line means that *spell* takes two required arguments – the dictionary and a file to check – then an optional list of additional files to check. As an example here is the output you will have to give for the above text.

```
$ ./spell rsc/words_en.txt example.txt
Spelling error: deeling
Spelling error: hih
Spelling error: dificult
Spelling error: sychological
Spelling error: generaly
Spelling error: adolesence
```

Listing 2: Command line example

2.2.1 Project organization

The project once done has to be packaged in a zip file named after both of your last name in this format: *lastname1-lastname2-class*. The class here is M1 or M2. Example: *El_ Atmani-Dupont-M2.zip*. Upon extraction, a single directory using the previous naming rule should come out.

Inside that directory, another directory named *src* will contain all your sources files (both C code and SHELL scripts). Near the *src* directory, the *rsc* directory contains the resources given with this project (dictionary and example files) and have to be deleted for the final zip file. Finally, a *bin* directory will contain – after the execution of the *make* command – the compiled *spell* program and the SHELL scripts. A README file can be put in the root directory of your project in order to write any comments you would find useful for me to know, like the part you haven't finished if any. An AUTHOR file has to be created next to the README and will contain the full name of the students that worked on this project, one per line as follow:

```
El Atmani Nahim
Dupont Pierre
```

The zip file should not contain anything else but your code, no binaries, no object files (.o) etc. Make sure to run *make clean* at the end and double check for binary files. One last thing, the project **have to compile**, even if you have not finished the project completely: make it compile without errors. This is important.

```
.
|-- AUTHOR
|-- README
|-- bin
|   `-- spell
|   `-- spell.sh
|-- rsc
|   |-- ex1_en.txt
|   |-- ex2_en.txt
|   |-- ex3_en.txt
|   `-- words_en.txt
`-- src
    |-- Makefile
    |-- ...
    `-- spell.c
```

Listing 3: Project organization

As for the name of the source files, function prototypes, variables etc you're free to use whatever you like. I'd recommend to chose wisely for them to be descriptive and meaningful. This is part of the project and will be assessed. Some prototypes are given through the project, you're free to use them or not as they just illustrate a behavior. They may not be the best available option depending on how you articulated your project. The only obligation you have for the makefile to work properly is to have your main function in a file named *spell.c*. If you don't follow those rules the zip file will likely be too heavy to send by email. For each project that I have received I will reply with a confirmation email, if you don't receive anything within the next 24h send it again. Pleaes include the second student in copy of the email while sending the project to me.

3 C part

Before diving into the code itself let's have a tiny Makefile that will compile our project. As we have not spent a lot of time talking about Makefile, here is one that respect the project organization rules.

```
CC = gcc
CPPFLAGS = -MMD
BUILD := debug
CFLAGS.common = -Wall -Wextra -Werror -pedantic -std=c99
CFLAGS.debug = -g3 -DDEBUG
CFLAGS.release = -O2
CFLAGS = ${CFLAGS.${BUILD}} ${CFLAGS.common}
SRC=
OBJ = ${SRC:.c=.o}
DEP = ${SRC:.c=.d}
BIN=spell
BINDIR = ../bin

all: $(BIN)
	mv $(BIN) $(BINDIR)

$(BIN): ${OBJ}

-include ${DEP}

clean:
	rm -f ${OBJ} ${DEP} $(BINDIR)/$(BIN)
```

Listing 4: Makefile example

The only thing that is left to do is to complete the SRC variable with the name of your source files. Invoking *make* for the compilation – once you're in your source directory – is as trivial as running:

```
$ make
```

Listing 5: make invocation

3.1 Handling arguments

Make sure *spell* handle any amount of arguments. It should prints the usage dialog (Listing 1) if the number of argument is not correct (that is less than three). Think about the *main* function to do so. To say it in other words, the spell checking have to happen sequentially on every files given as *spell*'s arguments on the command line.

3.2 Text manipulation

As part as the spell checking effort, *spell* have to read and parse the text of the different input files. The first operation will be to find the words in the given text and do something for each one of them. We call this step of separating and categorizing a big chunk of text into smaller units the tokenization. Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

3.2.1 Text parsing

Before tokenizing the input, we need to have a stream of text available to process. One could start by writing a function that take a pointer on a FILE and print it's content chunk by chunk. A chunk of text could be a sentence or just a line, or couple of words. The size of the chunk doesn't matter much, the only important thing is that we have to process the whole file eventually and that we do not want to cut words in half.

```c
void text_read(FILE *f);
```

Later the function can stop printing and instead give those chunk of text to the tokenizing function.

3.2.2 Text cleaning

For *spell* to work properly the text extracted from the files given as arguments have to be cleared of everything but lowercase letters. This way the latter step of word matching (3.3) will be straightforward.

As an example the following text contains capitalized letters and punctuation. Note that it could also have contained special characters and numbers.

> Trains and trucks were being shunted and unloaded. Moutains of stores, horse lines and mule liens were everywhere and ther was a babel of shouted commands.

After the first processing step it should come out as:

> **t**rains and trucks were being shunted and unloaded **m**outains of stores horse lines and mule liens were everywhere and ther was a babel of shouted commands

This stream of words – without punctuation, capitalized letters, numbers, special characters, etc – is the input for or next step: the word matching (3.3).

3.2.3 Text tokenization

Big words for a little thing though. Here, our text tokenization can be summarized as separating words from each other in a sentence. Words are usually separated by spaces, which make things even easier because space is now the only word separator we have to care about. The tokenizing function can start by printing every words of a string on a new line. Here is an example of a prototype for such a function. The function can be modified later to call another function on each of those words instead of printing them, e.g: the text cleaning function (3.2.2).

```
void text_tokenize(const char *text);
```

3.3 Word matching

3.3.1 Dictionary initialization

At this point, we have a function that output clean tokens (words) ready to be compared. The only thing we have to do now is to find if the word is in the dictionary. To do this, we will load the dictionary in memory so that we avoid reading in the file for each word we need to check and consume a lot of resources and time for nothing.

If you have looked at the dictionary *rsc/words_en.txt* you may have noticed that it is sorted, and only contain one word – all lowercase – by line. To load the dictionary in memory I'd recommend you to count how many lines are in the dictionary (with C code of course, don't hardcode the number of line, I may change my dictionary while correcting) and create an array of pointer of character of this size. Basically having one pointer by cell pointing at one string, which would be a copy of the dictionary's word at this line. This function would then return the allocated and initialized array, and a prototype for this function could be:

```
const char **dic_init(const char *filepath);
```

Which means that from a path like *rsc/words_en.txt* the function would return an array of strings containing every words in the dictionary, one per cell.

3.3.2 Binary search

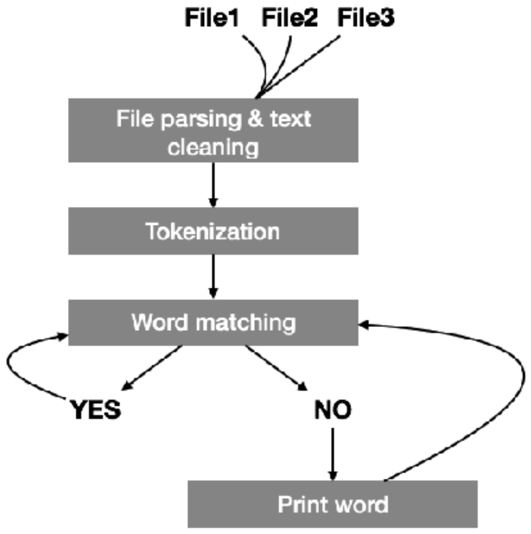
Now that we have a sorted array of strings, it will be trivial to search and find a word in it. We're going to use a search mechanism called binary search. An excellent article describing the mechanism of the binary search is available on the Khan Academy. A possible prototype for such a function could be:

```
int dic_contains(const char **dic, const char *word);
```

The function, using the binary search mechanism would return true if the *word* is contained in the *dic*, false otherwise.

3.4 Wrapping up

Now that you can give every word (one by one) as an input to the *dic_contains()* function and tell that they're in it or not we're pretty much done. The only thing that is left to do is to wire everything and print out – like in the Listing 2 – the misspelled words.



To summarize here is the steps that have to be followed to spell check all the files:

1. Open a file
2. Get a chunk of text from it
3. Clean the chunk from bad characters
4. Get a word from the chunk (tokenization)
5. Give the word to the matching function
6. Print if the word is not in the dictionary
7. Repeat 4-6 until there is no more words in the chunk
8. Repeat 2-6 until there is no more chunks in the file
9. Repeat 1-7 until there is no more files

3.5 Output

Here is the expected output for the file *ex1_en.txt*.

```
$ ./spell ../rsc/words_en.txt ../rsc/ex1_en.txt
Spelling error: streat
Spelling error: fase
Spelling error: werk
Spelling error: meel
Spelling error: hadn
Spelling error: dore
```

Listing 6: Expected output for one file

If no spelling mistakes are encountered, *spell* has no output. If more than one file are given to *spell* to process then the spelling mistakes (if any) will be sequenced in the order the files have been parsed.

```
$ ./spell ../rsc/words_en.txt ../rsc/ex1_en.txt ../rsc/ex2_en.txt
Spelling error: streat
Spelling error: fase
Spelling error: werk
Spelling error: meel
Spelling error: hadn
Spelling error: dore
Spelling error: primarry
Spelling error: secondarry
Spelling error: recieved
Spelling error: phisics
Spelling error: comunication
Spelling error: religeon
Spelling error: ambigous
Spelling error: cource
Spelling error: atempt
Spelling error: commprehend
Spelling error: luckilly
Spelling error: aquainted
Spelling error: simillar
Spelling error: paticular
Spelling error: steinbeck
Spelling error: malkovich
Spelling error: expresion
Spelling error: expresive
```

Listing 7: Expected output for two or more files

4 Shell

The shell part of this project shouldn't take you much time. The shell script have to display this help usage if:

- no arguments are given
- the number of arguments is not correct with the supplied option
- *-h* is used

```
$ ./spell.sh -h
Usage: ./spell.sh TARGET [TARGET [...]]
 TARGET can be either a file or a directory. If a directory is
 given as a TARGET, spell will run on every (non-code) ASCII
 file in that directory

 -h: display this help
 -r: if TARGET is a directory and the -r option is used,
     spell will recurse through directories too
 -v: be more verbose
```

Listing 8: Help message from spell.sh

The shell script should handle files and folder before making a **unique** call to the *spell* binary. Which means that *spell* eventually get a list of files to operate on within a single run. The reason for that is pretty simple. As you wrote it, the dictionary initialization will be done once every time *spell* is launched. Doing this triaging work in the shell script avoid us to pay for loading a huge dictionary each time we want to parse an infinitely smaller file.

4.1 Handling files

spell will refuse to execute on any file that is not pure ASCII text, that is even ASCII code files shouldn't be parsed. If the verbose option is activated, some warning have to be displayed. Also *spell* should be intelligent enough to not run on it's own dictionary.

```
$ cat foo.txt
Hello World
$ ./spell.sh foo.txt /bin/bash ../rsc/words_en.txt
$ ./spell.sh -v foo.txt /bin/bash ../rsc/words_en.txt
foo.txt:  Added for parsing...
/bin/bash:  not a regular text file, skipping
../rsc/words_en.txt:  Skipping dictionnary
```

Listing 9: Handling files with special cases

The format for printing should match *filename:\tWarning*.

4.2 Handling directories

If one of the TARGET is a folder, the script have to expand the content of this folder to files that *spell* can process. Which means that those files have to follow the previous rules too (Handing files 4.1).

```
$ ./spell.sh ../rsc | grep -v Spelling
$ ./spell.sh -v ../rsc | grep -v Spelling
../rsc/huge.txt:  Added for parsing...
../rsc/spell: not a regular text file , skipping
../rsc/ex3_en.txt:  Added for parsing...
../rsc/ex1_en.txt:  Added for parsing...
../rsc/ex2_en.txt:  Added for parsing...
../rsc/words_en.txt:  Skipping dictionnary
```

Listing 10: Handling directories without recursion

Here, the *grep* call is only present to eliminate from the output the spelling mistakes that *spell* have found as this is not the main point of this section.

4.3 The recursive option

The recursive option should be pretty straightforward to add to your current functionalities as it only tells the directory option to not stop on the first level. Hence the previous example becomes:

```
$ mkdir ../rsc/test
$ cp ../rsc/ex3_en.txt ../rsc/test
$ ./spell.sh ../rsc | grep -v Spelling
$ ./spell.sh -vr ../rsc | grep -v "Spelling"
../rsc/test/ex3_en.txt: Added for parsing...
../rsc/huge.txt:  Added for parsing...
../rsc/spell: not a regular text file , skipping
../rsc/ex3_en.txt:  Added for parsing...
../rsc/ex1_en.txt:  Added for parsing...
../rsc/ex2_en.txt:  Added for parsing...
../rsc/words_en.txt:  Skipping dictionnary
```

Listing 11: Handling directories with recursion

As you can see here, *spell* is going through every folder that may be present in the target folder.

4.4 The verbose option

The verbose option print out some valuable informations on *spell*'s behavior. In the following list, replace filename by the actual concerned filename. The verbose option should print out:

- *filename:\tSkipping dictionnary*: when the given file is the *spell* dictionary

- *filename:\tAdded for parsing...*: when the given file is a correct ASCII file for *spell* to process
- *filename:\tnot a regular text file, skipping*: when the given file is not a correct ASCII file for *spell* to process

Please refer to previous listings for examples (e.g: Listing 9, 11).

5 Help

As a little help here is some functions from the standard C library and some shell commands that you may find useful in the realization of this project:

5.1 Man

5.1.1 Stdlib

1. fopen
2. isalpha
3. fscanf
4. sscanf
5. fclose
6. getline
7. malloc
8. calloc
9. free
10. strcmp
11. ...

5.1.2 Shell command

1. getopt
2. file

3. grep
4. find
5. ...

You're free to use any resources you may find useful in the standard library or SHELL features that we haven't seen in the class if you think they make sense and would help you building *spell*.

5.2 Ref

As an ulitme help, you will find in the bin folder a *spell_ref* binary. This is the compiled spell checker that you can use as a reference to compare your behavior with the expected output. Please delete it when you're building the final zip file for your project. In fact, no binaries should be present at this time in the zip file (cf 2.2.1).

Listings