CS 211: Computer Architecture, Fall 2012
Programming Assignment 1: prefixStat

1 Introduction

This assignment is designed to get you some experience with programming in C, as well as compiling, linking, running, and debugging a C program in our evironment. Your task is to write a C program called `prefixStat` that reads a test file and outputs following results, which consists of two parts:

- Part I: Read from file—outputing the words and word count, outputing the lines and line count.
- Part II: Prefix matching—outputing the words in which the given input word appears as prefixes.
- IMPORTANT: Every one should have the same test file (the file you read words from) name—`test.dat`.

A word is defined as any sequence of ASCII characters, and the words should be case-insensitive. That is, "book" and "Book" and "bOOk" are the same word.

2 Implementation

Implement a program called `prefixStat` with the following usage interface:

`prefixStat <option> <input word>`

where:

- `<prefixStat>` is the executable file of your program.
- `<option>` is "h", "w", "l", or "p".
- `<input word>` is the given word to do prefix matching.
- `prefixStat -h`: help for how a user can run the program and quit.
- `prefixStat -w`: outputing the words and word count.
- `prefixStat -l`: outputing the lines and line count.
- `prefixStat -p <input word>`: outputing the words in which the given input word appears as prefixes.

Your program should check the number of parameters passed to main() to decide what action to take. For example, when the number of parameters is greater than 3, you will print out an error message.
As an example, running `prefixStat` on a test file with the following content:

```
dppet apple applet apt apet
```

prefixStat -w will output:

```
dppet apple applet apt apet
5
```

prefixStat -l will output:

```
dppet apple applet apt apet
1
```

prefixStat -p app will output:

```
applet apple
```

3 Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa1.tar`. To create this file, `cd` into the directory containing your submission and run the following command:

```
tar cf pa1.tar readme.pdf Makefile *.h *.c
```

Note, if you want to hand in additional files that are not `.h` or `.c` files, you will need to modify the above command. To check that you have correctly created the tar file, you should copy it (`pa1.tar`) into an empty directory and run the following command:

```
tar xf pa1.tar
```

This should extract all the files that we are asking for below directly into the empty directory (that is, no sub-directories). Your tar file must contain:

- readme.pdf: this file should describe your design and implementation. In particular, it should detail your design, any design/implementation challenges that you ran into, and an analysis (e.g., big-O analysis) of the space and time performance of your program.
- Makefile: there is a sample Makefile in the assignment package, make any modifications as needed.
- source code: all source code files necessary for building `prefixStat`. Your source code should contain at least 2 files: `prefixStat.c` and `prefixStat.h`.

4 Grading Guidelines

4.1 Functionality

This is a large class consequently a significant part of your grade will be based on programmatic checking of your program. That is, we will build a binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should make sure that we can build your program by just running `make`.
- You should test your code as thoroughly as you can. *In particular, your code should be adept at handling exceptional cases.* For example, `prefixStat` should *not* crash if the argument file does not exist.

Be careful to follow all instructions. If something doesn't seem right, ask.

4.2 Coding Style

Having said the above about functionality, it is also important that you write "good" code. Thus, *part of your grade will depend on the quality of your code.* Here are some guidelines for what we consider to be good:

- Your code is modularized. That is, your code is split into pieces that make sense, where the pieces are neither too small nor too big.
- Your code is well documented with comments. This does not mean that you should comment every line of code. Common practice is to document each function (the parameters it takes as input, the results produced, any side-effects, and the function's functionality) and add comments in the code where it will help another programmer figure out what is going on.
- You use variable names that have some meaning (rather than cryptic names like `i`).

Further, you should observe the following protocols to make it easier for us to look at your code:

- Define prototypes for all functions.
- Place all prototype, `typedef`, and `struct` definitions in header (.h) files.
- Error and warning messages should be printed to `stderr` using fprintf.

4.3 Performance

Finally, part of your grade will depend on your design. That is, we expect you to write reasonably efficient code based on reasonably performing algorithms. For each assignment, you will need to analyze the performance of your code and justify it as part of discussing your design.