CS 345: Programming Assignment 1 (Due: Nov 4, 2013 before 23:59)

Rules. Teams of no more than two students as explained in syllabus (Handout 1). This is to be handed back no later than midnight on the Monday it is due (see information below) in electronic form as a single tar or zip file decompressible on an AFS machine (afsconnect1.njit.edu or afsconnect2.njit.edu), where testing would take place. (At a minimum, the zipfile should contain a text file `HW1_ABCD_EFGH.txt` as explained below.) The code should be compilable/interpretable and executable on one or the other identical AFS machines in any language available there (emphasis C, C++, Java, Python, Perl).

1 A file-based desktop search engine: fdsee

You will build the first components of a small and usefule file-based (the f of fdsee), desktop (the d of fdsee) search engine (the se from search and the e from engine. I am not very good with acronyms so **fdsee** it will be.

1.1 Objectives of the assignment

We assume that a crawler has stored locally in a single directory say `alexg` a collection of files already crawled from the Web; within that directory there might be subdirectories of files.

The first phase of this assigment is to go through all the files and identify from their suffixes which ones are text searchable or not (a directory, or other files that are not searchable). A stream of the text searchable files (path name relative to `alexg` will be generated for verification of the successful completion of this objective.

In a second phase this stream of text searchable files will then be tokenized; in a follow-up assignment they will be indexed and then searched.

1.2 Introduction

The first two phases of an indexing step are undertaken. This involves determining which files are indexable, and then performing the parsing of those files. The first phase consists of Step1 and the second phase of Steps 2 and 3.

Step1: **Searchable Documents**. You will traverse the directory structure of the locally stored Web-pages (stored there as a result of web crawling) and identify the text searchable files among them. Some assistance is provided in the protected area of the course Web-page in the form of program `recdir.c` available as link L5 there. It's up to you whether you want to use it or not (it is only tested on Unix). Searchable files will be uncompressed ASCII text files. After you have identified the searchable text files, you are then about to start the process of parsing these files and tokenizing their contents.

Step 2: **Tokenization**. The second step involves the first part of parsing, that of tokenization. For every searchable document, you identify and extract keywords/tokens of interest along with other useful information (such as word position in the text, font-size, and position in an html document). In this part you also convert keywords to numbers (wordIDs) and also URL locators to numbers (docIDs).

Step 3: **Linguistic Analysis.** The third step involves the next part of parsing, the linguistic analysis of the keywords/tokens that would be turned into index terms. For the sake of this assignment we will concentrate only on elementary `stopword` removal. The output of this phase will be the tokens identified minus the stopwords; this stream of keywords will then be fed into the index (wait for the next assignment).

This is the minimum implementation required to gain you the full points of this assignment. You can enrich this implementation by adding additional features such as searching more file types, do a more thorough parsing or more elaborate linguistic analysis.

1.2.1 Project deliverables

After compilation the executable unit produced should be named `fdsee`. If your code is interpreted the wrapper function should be `fdsee` possibly with an appropriate suffix as needed. We shall refer to that file as `fdsee` in the remainder. Every source file you submit must include in the form of comments in the first 5 lines the names of the members of the group including the last four digits of their NJIT IDs. In addition a file named `HW1_ABCD_EFGH.txt` (or `HW1_ABCD.txt` if the team has only one member) needs to be included (that also conforms to the first-5-line convention) that includes instructions for compilation/interpretation, bugs, and anything else of interest (eg extensions).

1.2.2 Step 1: Searchable Document

The program `fdsee` will read the command line and behave as follows.

```
% ./fdsee searchable filename
% ./fdsee searchable directoryname
```

The first argument in the command line (after the name of the executable) denotes the action. The second argument is an arbitrary file or directory name.

For action `searchable` (searchable document list) you need to decide whether `filename`, or all the files under `directoryname` (and its subdirectories) are searchable text files or not and list them. A searchable text file `SETF` is a file with one of the following suffixes:

`.html , .htm , .txt , .cc , .cpp , .c , .h , .java .`

A `SETF` (for SEarchable Text File) file can be `HTML` or `TEXT`. An `HTML` file is a searchable text file with suffixes `.html` or `.htm`. A `TEXT` file is a searchable text file with any of the remaining suffixes. A file that is either a directory name or a non-searchable text file is `NSTF` (non-SETF abbreviation to four characters).

The command argument `searchable` prints for every file identified as an `SETF` whether it is `HTML` or `TEXT` along with the full path name relative to the directory name listed as the second argument during program invocation. A (partial) output may look like as follows in the case where there are two HTML files in subdirectory `cs345` of directory `courses` which is the only directory of `alexg`. An `END-OF-LISTING` concludes this listing.

```
% ./fdsee searchable  alexg
alexg/courses/cs345/index.html  HTML
alexg/courses/cs345/handouts.html  HTML
END-OF-LISTING
```

In general one of the following labels will be printed `HTML, TEXT` for `SETF` files, and optionally `NSTF` for non-SETF files. (You may only list `HTML, TEXT` files, and ignore NSTF files.)

1.2.3 Step 2: Tokenization

If action is `token` then for every file identified as `SETF` (with `HTML, TEXT` printed in the output) a parsing function is called that tokenizes each file into its individual tokens. You decide what constitutes a token. A `tokendebug` should also be made available!

```
% ./fdsee token filename
% ./fdsee token directoryname

% ./fdsee tokendebug filename
% ./fdsee tokendebug directoryname
```

The tokenization phase is probably the most difficult part of this assignment and the most time consuming. You need to decide how to parse a document and what constitutes a token. You might make the job at hand easier if you use a parser program such as lex or yacc for this part (and the time involved to familiarize yourselves with them, if you have never used them before). An hour or two reading a manual page or the extensive documentation for the GNU equivalents names flex and bison might save you time building a tokenizer from scratch.

Tokenizing a `TEXT` file is easier. Interesting tokens that will become index terms are going to be words (e.g. alphanumeric strings starting with a character) or non-trivial numbers. Single digit numbers can be discarded dates such as 1950 might become useful searchable terms. Collectively we will call all these interesting tokens `words` even if some of them are numbers.
`Words` will be represented by `wordID`s, i.e. a preferably 32-bit unsigned number.
`Documents` will be represented by `docID`s, i.e. a preferably 32-bit unsigned number.
Therefore if word `algorithms` is identified in a `TEXT` document a quadruplet of information is output by the `token`izer (an explanation is provided after the NOTES):
`(docID,wordID,wpos,attr)`

NOTE 1. The tokenizer generates words that are lower case no matter what the original case was.
NOTE 2. You need to have ways to determine the word represented by say `wordID` 25 or the full URL name of the document with `docID` 100, eg `alexg/courses/cs345/handouts.html`.
NOTE 3. For the sake of this and remaining assignments a file in say directory
`alexg/courses/cs345/index.html` has a full URL which is
`http://www.cs.njit.edu/~alexg/courses/cs345/index.html`.

The first element of the quadruplet above is a `docID` and not the full-URL of the document in which the named word appears. The second one is the unique `wordID` for the specified word (e.g. `algorithms`). All files that have been searched should have a common lexicon (e.g. dictionary of words) and a single entry for each unique word. The third attribute `wpos` is positional information about the identified word in the document. It identifies the word position of the word in the document, i.e. whether `algorithms` is the 10-th or 20-th word in the named document. (The value of `wpos` will depend on the tokenization method that you use.) The last element is an `attr`ibute value. For `TEXT` files all words have attribute 0. For `HTML` files certain words may have higher attribute values. Words surrounded by a `<TITLE>` tag will have attribute value 1. Words surrounded by an `<A>` which is an anchor tag will have attribute value 2. Words surrounded by a `<H1> to <H3>` tag have value 3, and `<H4> to <H6>` tags have value 4. Other values are possible; your implementation is open-ended.

So a command such as the one depicted below will generate in the output a stream of parenthesized numbers (the example below shows only a single quadruplet).

```
% ./fdsee token alexg/courses/cs345/index.html
(10,20,30,4)
```

NOTE 4. We do not specify how big the attribute or positional values are going to be. It's up to you to decide so.
If however the command is `tokendebug` instead of `token`, more useful information will be printed.

```
% ./fdsee tokendebug  alexg/courses/cs345/index.html
```

The output will still be a quadraplet with docID, wordID, and attr appropriately replaced
`( http://www.cs.njit.edu/~alexg/courses/cs345/index.html,algorithms,30,<H4>)`

NOTE 5. The `<H4>` refers to attribute values 4. Thus `<H4>` is the class and might imply that the word algorithms was surrounded by one of `<H4>, <H5>, <H6>`, not necessarily `<H4>`.

1.3 Step 3: Linguistic Analysis

The linguistic analysis module will take as input the output of Step 2 and eliminate those keywords that are identified as stopwords. The list of stopwords for the sake of this assignment is given below.

```
    I a about an are as at be by com en for from how in is it  of on or that the
  this to was what when where who will with www

% ./fdsee stopword filename
% ./fdsee stopword directoryname
```

The effect of command `stopword` is incremental. It is equivalent to `token` with the addition of stopword elimination. (In the generated stream, stopwords will not be listed.) It's up to you whether you want to insert them into the lexicon.

1.3.1 Side Effects

Each one of the three commands `stopword`, `token` and `tokendebug` will have the following common side-effect. The generation of a file `fdsee_lexicon` in the directory in which `fdsee` was invoked and the generation of a file `fdsee_doclist` also in the same directory. The first file `fdsee_lexicon` lists all words indexed by wordID and the latter `fdsee_doclist` lists all URL/documents indexed by docID. The two files are ASCII printable/viewable files. For example the output might look like as follows. No indentation or pretty printing is required. However one item per line should be printed. (wordID or docID need not start from zero nor be consecutive in value.)

```
% cat ./lexicon
 0  alex
 ...
 20 algorithms
 ....
```

In addition all tuplets generated should be dumped into a file named `fdsee_dump`. The `stopword, token, tokendebug` components can all use the same file; if not have a filename where the third component is one of those three words separated from `dump` with an underscore as well. (In the file, parentheses and commas should not be printed just include one space for visibility.) ■

Date Posted: 9/16/2013