$Id: asg2j-jroff-queue.mm,v 1.20 2014-01-24 11:45:08-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/asg2j-jroff-queue
URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Assignments/asg2j-jroff-queue/

1. Overview

In this assignment you will write a very simple text formatter in the spirit of **nroff**, which is similar to the more powerful text formatters such as **troff** [1] which is proprietary, and **groff** [2] which is Gnu [3]] free software. ("*Free software is a matter of liberty, not price. To understand the concept, you should think of free as in free speech, not as in free beer.*") similar programs are TEX [4] created by Donald Knuth [5] and LATEX [6]. You will also write code for the simple linear abstract data type **queue**, implemented as a linked list.

[1] **http://www.troff.org/**
[2] **http://www.gnu.org/software/groff/**
[3] **http://www.gnu.org/philosophy/free-sw.html**
[4] **http://www.tug.org/texlive/**
[5] **http://www-cs-staff.stanford.edu/~uno/**
[6] **http://www.latex-project.org/**

Note: The only code you may use is code you or your partner wrote yourselves, or copied from the course directory, or which is available as part of the **java.lang** and **java.io** packages. You may not use any classes from **java.util**, except for those classes specifically listed in the Syllabus.

2. Manual Page and `example-output` Directory

The manual page describing the software you are to implement is in the subdirectory **example-output/**. In that directory, you will find several files:

proff.perl	is a program which is a wrapper around **groff**, which does the work your program has to do.
jroff.rf	is a **jroff** source file containing markup language specifying your project.
jroff.out	is a plain text file which is in readable format for printing. When your program reads **jroff.rf**, it should produce exactly this file.
test*.rf	are other test files (input data).
test*.out	are other output files created from these input files.
test*.err	are some samples of error messages that might be generated.

3. Provided Code

You are also provided with a **code/** subdirectory, which you may use as a source to begin your program:

Makefile	which you can modify to build your **jroff** jar file from Java sources.

`jroff.java`	is a debugging trace starter for your main program. Delete all lines of code marked "`STUB`".
`commands.java`	contains a `HashMap` that will be used to dispatch each of the control sequence commands.
`linkedqueue.java`	is a stub for a linked list implementation of a queue. Finish the code and documentation for this file.
`auxlib.java`	the same file from a previous assignment. use the facilities provided herein to print message, etc.

4. Implementation Strategy

It is important to implement your program a little at a time. Begin by copying the Java programs from the `code` subdirectory into your project development directory. Then proceed in steps. One possible sequence is\(:

(i) Do a recursive copy (`cp -R`) of the `code` subdirectory and begin with the code provided. Your results will be written to `stdout` and error messages to `stderr`.

(ii) Note that each word is printed one word at a time boxed in by brackets for visibility. The function `java.lang.String.split` accepts a regular expression and splits the string into an array of strings. Note that if there are leading spaces, the first word is empty.

(iii) For each empty line in the input, print the stub message "`.sp 1`" as if it were a dot command.

(iv) Implement `linkedqueue`: replace code that throws `UnsupportedOperationException` with working code. Your implementation is a linked list. The function `insert` creates a new node and appends it to the end of the list. The function `remove` unlinks the first node from the list and return its value.

(v) Add the following statement to your main function:

```
linkedqueue<String> wordqueue = new linkedqueue<String> ();
```

(vi) Change your code so that instead of just printing words read, each word read is inserted into the end of the queue. Whenever you see a control line, read each word from the queue and print it.

(vii) Delete the printing code and move it into an auxiliary function which has a local `StringBuffer`. Implement `printparagraph`. The main loop of this function should proceed as follows:

(a) If the string buffer is empty, `append` the new word to it.

(b) If not empty, figure out how many spaces to append to it: A sentence-ending character gets 2 spaces. Any other character gets one space.

(c) If this does not make the buffer exceed the line length, append the space(s) and the word and cycle to the next word. If it does exceed the line length, print the string buffer, clear it out, and put the word at the front of the buffer.

(d) When the queue is exhausted, print the string buffer, unless it is empty.

(viii) The class `commands` implements each of the commands with a stub. Dispatch is done via a switch statement, which is kept deliberately small by only calling functions. Note that Java 1.7 allows the argument to a switch to be a string. Java 1.6 does not allow that, so if you develop on your own computer, you will need at least Java 1.7.

(ix) Implement each of the control commands in some arbitrary sequence:

`.\"` The line is simply discarded and does not break a paragraph. Any other command causes the current paragraph to be dumped and the queue to be emptied.

`.bp` The program switches to top-of-page mode.

`.br` Call dump paragraph. The queue is flushed to the standard output and then cleared.

`.cc C` Change the control character to '*C*', for any character. Your must now retrofit the code that looks for dots in the first position and recognize this character instead.

`.in N` Every line printed is moved to the right by both the page offset and the indentation. This number is remembered for future output.

`.ll N` Set the line length as specified. The dump paragraph routine wraps when a word exceeds this length.

`.mt N` Sets the height of the top margin.

`.pl N` Sets the page length to this number. A page eject occurs whenever this directive is found, or printing a line causes the page to be full.

`.po N` Every line of output except for empty lines is preceded by this number of spaces.

`.sp N` Dump the paragraph. Then remember the number of blank lines to be printed. The next time a paragraph is dumped, this number of empty lines is printed. However, if printing that number of empty lines would fill the page, the program switches to top-of-page mode.

(x) Fix your paragraph output routine: Whenever you print, your program is either in top-of-page mode or in mid-page mode. Initially, it is in top of page mode. In top-of-page mode, before printing an output line, it must print a form feed (`\f`), followed by $N/2$ blank lines, followed by the page title, followed by $N/2-1$ blank lines, followed by the line to print. In mid-page mode, it just prints the line. The line feed is suppressed on the first page. When a line feed is printed, it is not printed on its own line. The page headers is left, mid, and right justified with any occurrence of the character "`%`" replaced by the current page number.

(xi) See the subdirectory `example-output/`. The files `*.out` from your program should `diff`(1) identical to those generated by your program. The files `*.err` should be similar. The files `*.log` should show exactly the same exit status.

5. Other nodes

Normally you are prohibited from using anything from `java.util`, except for those specific classes listed in Figure 1 of the syllabus. In this program, however, `commands.java` does make use of a `HashMap` for eas of looking up commands in lieu of a switch statement, which does not work with strings.

When `proff.perl` runs `groff`, there is a bogus error message that is printed:

```
grotty:<standard input>:5: character above first line discarded
```

Ignore this. It is a bug in `grotty`. Not your problem.

The subdirectory `example-output` contains some sample runs. You may test the correctness of your program by using `diff` to compare your program's output with that of `proff.perl`.

6. What to submit

Submit the files `README`, `Makefile`, and the necessary Java source files. Your `Makefile` should have the targets: `all`, the first target, which builds the jar; `ci` which checks in all files into an `RCS` subdirectory. `submit`, which submits files, `clean`, which deletes any files built by `gmake all`, except for the jar. Verify that your submit works using the command `testsubmit`.

If you are doing pair programming, carefully read the document in `cmps012b-wm/Syllabus/pair-programming/` and submit the `PARTNER` file as required. Pair programming is encouraged in all assignments.

Every file you submit should have as its first line your name and username. Its second line should be an RCS comment `$Id$` string in a comment. ***CAUTION:*** Before submitting anything, carefully read the section in the syllabus that covers cheating.