**CSCE 5533 Advanced Information Retrieval**

**Homework 1**

**Due: Thursday, September 6 at 11:59pm**

**By: Chenglong Lin**

**Objectives**

The objective of this homework is to implement a tokenizer that tokenizes and downcases all words in a collection of HTML documents.

**Approach**

In this project, I decided to use Flex scanner generator to implement my tokenizer. This scanner generator takes in a lex files and produces a C/C++ scanner code file. The lex file contain 3 sections of code. The first section is the definitions section. It is the place to put preprocessor directives and declare global variables/constants. The second section is the rules section where I put the patterns and the corresponding actions. The third section is the user code section where I do function implementations.

To handle punctuations, I wrote a “complement” matching rule. What I mean by “complement” matching is that after all words, urls, emails, and tags are matched and processed by corresponding actions. What is left must be punctuations, so this “complement” matching rule will match them and consume them.



To handle numbers, I wrote two rules.

The first rule is for matching floating numbers.





This rule however would produce some erroneous tokens if used alone, because version numbers may contain multiple dots. I added a version number rule to match these version numbers as single word. Flex will apply action of the rule that match the most characters, so I do not have to worry about these two rules conflicting with each other.

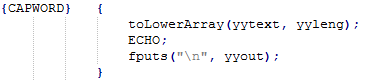


The second rule is for matching numbers that has comma separators or pure integer digits.

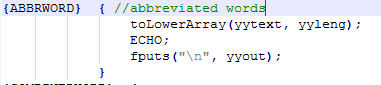


To handle words, I wrote four rules.

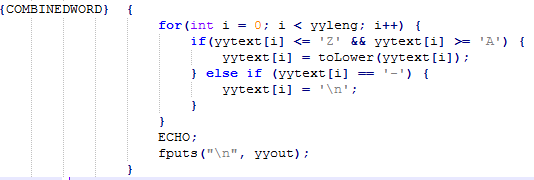
The first rule is for matching all capitalized character words. Once there is a match, it will loop through the yytext array and turn them to lower case letter.



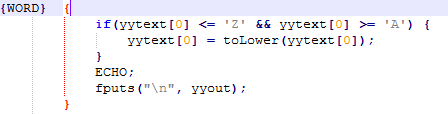
The second rule is for matching abbreviated word. Once there is a match, it will loop through the yytext array and turn capitalized letter to lower case letter.



The third rule is for matching combined words that are joined by hyphens. Once there is a match, it will loop through the yytext array to turn capitalized letter to lower case letter and replace hyphen characters to next line characters.



The fourth rule is for matching normal words. Once there is a match, it will check if the first letter is capitalized. If so, it will turn it to lower case letter.



To handle emails, urls, phone numbers, I created patterns and rules for each of them.

To calculate the frequency of each tokens, I used a Unix command chain.

cat ./output/\* | sort | uniq -c | sort -nr > result.out

What this does is:

1. Combine all output text files together, then it pipes the resulting text to the next command;

2. The sort command sorts the resulting text in alphanumeric order, then it pipes the sorted text to the next command;

3. The Uniq command deletes the duplicate tokens and counts the occurrences of each token, then it pipes the resulting text to the next command;

4. The sort command sorts the tokens in order of descending frequency, then it outputs the resulting text to a file.

**Results**

The results of tokenizing are

Most Frequent tokens (10)

Token, frequency

the 27439

and 17169

of 16688

to 12664

a 10076

in 9033

for 7359

is 6025

on 5081

1996 4398

Least Frequent tokens (10)

Token, frequency

00198 1

000fff 1

0004 1

0000e0 1

0000b0 1

0000a0 1

000088 1

000033 1

00000 1

000 1

Alphabetically first tokens (10)

Token, frequency

0 702

.0 5

00 1586

0.0 14

000 1

0,0,0 2

00.0 3

0000 17

00000 1

000000 33

Alphabetically last tokens (10)

Token, frequency

zwaenepoel 2

zwalm 1

zwatch 1

zwhere 1

zwiener 1

zwilling 1

[zwilling@cs.wisc.edu](mailto:zwilling@cs.wisc.edu) 1

zyda 1

zytkow 1

zyuganov 1

Discussion

In medium.html, the “<b>sgauch</b>@uark.edu” email address isn’t tokenized as a single email address but two tokens: “sgauch” and “uark.edu”. This is because the email matching rule does not handle the case when the email address contains html tags. This case happens rarely, so I decided not to modify my email matching rule to identify this type of email.

In medium.html, the “<b>E</b>lephants” isn’t tokenized as a single word but two tokens: “e” and “lephants”. This is because my word matching rule does not handle words that have html tags in between the characters. This case happens rarely, so I decided not to modify my word matching rule to identify this type of word.

In medium.html, the “H<sub>2</sub>0” chemical compound isn’t tokenized as a single token but three tokens “h”, “2”, and “0”. This is because the chemical compound is a rare occurrence in this file achieve, so I decided not to add a chemical compound rule to identify chemical compounds.

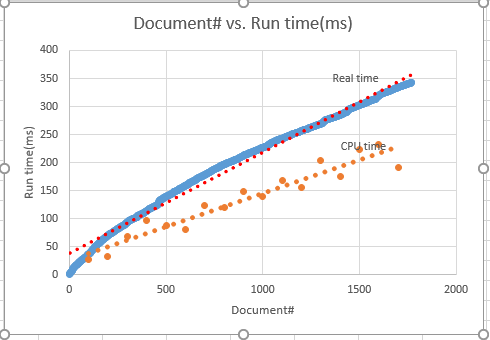
Besides these tokens, all other most frequent tokens are well tokenized.

**Efficiency**

Complexity Analysis

I’m using Flex to implement my scanner. Flex combines all the regex rules into a single DFSA, so the efficiency of my tokenization program is O(n)|n: total number of characters in the documents.

Timings

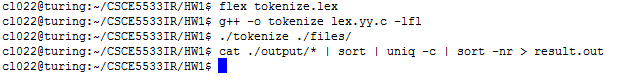


Both function look linear.

**Testing**

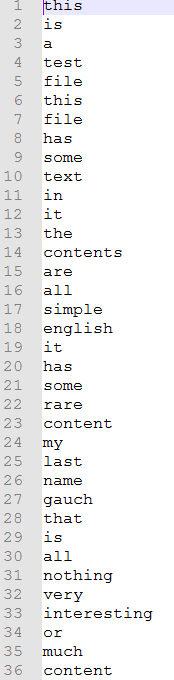
Sample Run

Typescript or screenshot of program running **on turing**.



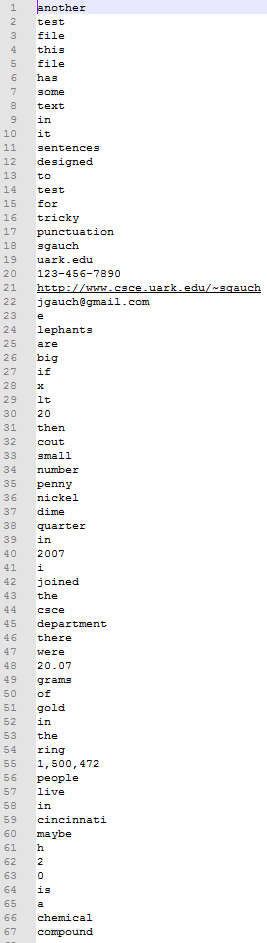
Resulting Tokens: simple.html

Tokens created by your preprocessor



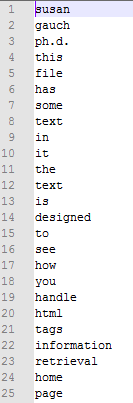
Resulting Tokens: medium.html

Tokens created by your preprocessor



Resulting Tokens: hard.html

Tokens created by your preprocessor



Resulting Tokens: 1632.html

Tokens created by your preprocessor

