

**Faculty of Computer Science, Dalhousie University**  
**CSCI 4152/6509 — Natural Language Processing**

*18-Sep-2024*

**Lecture 5: Basic NLP in Perl**

Location: Carleton Tupper Building Theatre C      Instructor: Vlado Keselj  
 Time: 16:05 – 17:25

**Previous Lecture**

- NFA-to-DFA translation (continued)
  - Review of Regular Expressions
  - Introduction to text processing with Perl
- 

**Perl Regular Expressions: ‘proc...ing’ Example**

*(repeated slide)*

- Similar functionality as grep:

```
#!/usr/bin/perl
# run as: ./re-proc-ing.pl linux.words

while ($r = <>) {
  if ($r =~ /proc...ing/) {
    print $r;
  }
}
```

We can note that in Perl regular expressions are delimited by default with slash (‘/’) character, as in `/proc...ing/`.

**Shorter ‘proc...ing’ Code**

- There are several ways how this program can be made shorter: first, let us use the default variable ‘\$’:

```
while ($_ = <>) {
  if ($_ =~ /proc...ing/) {
    print $_;
  }
}
```

- Shorter version:

```
while (<>) {
  if (/proc...ing/) {
    print;
  }
}
```

**Even Shorter 'proc...ing' Code**

- and shorter:

```
while (<>) {
  print if (/proc...ing/);
}
```

- and shorter:

```
#!/usr/bin/perl -n
print if (/proc...ing/);
```

- or as a one-line command:

```
perl -ne 'print if /proc...ing/'
```

**More Special Character Classes**

\d — any digit  
 \D — any non-digit  
 \w — any word character  
 \W — any non-word character  
 \s — any space character  
 \S — any non-space character

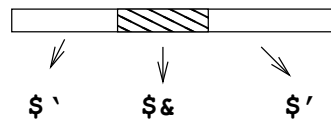
**A More Complete List of Iterators**

\* — zero or more occurrence  
 + — one or more occurrences  
 ? — zero or one occurrence  
 {n} — exactly *n* occurrences  
 {n,m} — between *n* and *m* occurrences  
 {n,} — at least *n* occurrences  
 {,m} — at most *m* occurrences

**Special Perl Variable Assigned After a Match**

\$var =

regular expression match: \$var =~ /re/

**Example: Counting Simple Words**

```
#!/usr/bin/perl

my $wc = 0;
while (<>) {
  while (/\\w+/) { ++$wc; $_ = $'; }
}
```

```
print "$wc\n";
```

### Example: Counting Simple Words (2)

- Consider the following variation:

```
#!/usr/bin/perl

my $wc = 0;
while (<>) {
    while (/\\w+/g) { ++$wc }
}
print "$wc\n";
```

### Counting Words and Sentences

```
#!/usr/bin/perl
# simplified sentence end detection

my ($wc, $sc) = (0, 0);
while (<>) {
    while (/\\w+|[.!?]+/) {
        my $w = $&; $_ = $';
        if ($w =~ /^[.!?]+$/) { ++$sc }
        else { ++$wc }
    }
}
print "Words: $wc Sentences: $sc\n";
```

The following are some more extended features of regular expressions that behave in a similar ways as anchors:

### More on Perl RegEx'es

<code>\G</code>	anchor, end of the previous match
<code>(?=re)</code>	look-ahead
<code>(?!re)</code>	negative look-ahead
<code>(?&lt;=re)</code>	look-behind
<code>(?&lt;!re)</code>	negative look-behind

- Some examples:

```
/foo(?!. *foo)/ — finding last occurrence of 'foo'
s/(?<=\be) (?=mail)/-/g — inserting hyphen
/\\b\\w+(?<!s)\\b/ — a word not ending with 's'
```

The anchor `'\G'` matches the end of the last regular expression match on a string. It can be conveniently used in tokenization as shown in the next example:

### An Example with `\G`

```
while (<>) {
```

```

while (1) {
  if    (/G\w+/gc) { print "WORD: $&\n" }
  elsif (/G\s+/gc) { print "SPACE\n" }
  elsif (/G[.,;?!]/gc)
    { print "PUNC: $&\n" }
  else { last }
}
}

```

- Option `g` must be used with `\G` for global matching
- Option `c` prevents position reset after mismatch

### Back References

- `\1 \2 \3 ...` match parenthesized sub-expressions
- for example: `/(a*)b\1/` matches  $a^nba^n$ ; such as `b, aba, aabaa, ...`
- Sub-expressions are captured in `(...)`
- Aside, in `grep`: `\(...\)`
- `(?:...)` is grouping without capturing

The expressions `'\1'`, `'\2'`, `'\3'`, and so on are so-called *back references*, and they match a repetition of a string previously captured in parentheses in the same regular expression. For example, the expression:

```
/<(\w+)>.*<\1>/
```

would match a string such as `<h1>test</h2>abc</h1>`. As we can see, it will match a string ending with an HTML tag, but only if the tag is the same as the one used in the starting tag. By choosing the number, we can pick which pair of parentheses to match. For example, the regular expression:

```
/(a+(b+))(c+(d+))\4/
 1 2   3 4
```

will match any string of the form  $a^n b^m c^k d^l d^l$ . The numbers below the expression show how we count the opening parentheses to decide which one to match in the back reference. As a note, the superscripts  $n$ ,  $m$ ,  $k$ , and  $l$  denote repetition of the corresponding letter for that number of times. For a comparison, the following regular expression:

```
/(a+(b+))(c+(d+))\3/
 1 2   3 4
```

would match any string of the form  $a^n b^m c^k d^l c^k d^l$ .

### Shortest Match

- default matching: left-most, longest match
- e.g., consider `/\d+/`
- Shortest match is sometimes preferred
  - e.g., consider: `<div>.*</div>/` or `<[>]*>/` vs. `<.*>/`
  - and: `<div>.*?</div>/` and `<.*?>/`
- Shortest match iterators:
  - `*? +? ?? {n}? {n,m}?`

### Regular Expression Substitutions

- syntax: `s/re/sub/options`
- Some substitution options
  - `c` – do not reset search position after `/g` fail
  - `e` – evaluate replacement as expression

- g – replace globally (all occurrences)
- i – case-insensitive pattern matching
- m – treat string as multiple lines
- o – compile pattern only once
- s – treat string as a single line
- x – use extended regular expressions

### 5.3 Text Processing Example: Counting Letters

#### Text Processing Example

- Perl is particularly well suited for text processing
- Easy use of Regular Expressions
- Convenient string manipulation
- Associative arrays
- Example: Counting Letters

#### Experiments on “Tom Sawyer”

- File: TomSawyer.txt:

The Adventures of Tom Sawyer

by

Mark Twain (Samuel Langhorne Clemens)

#### Preface

MOST of the adventures recorded in this book really occurred; one or two were experiences of my own, the rest those of boys who were schoolmates of mine. Huck Finn is drawn from life; Tom Sawyer also, but not from an individual -- he is a combination of the characteristics of three boys whom I knew, and therefore belongs to the composite order of architecture.

The odd superstitions touched upon were all prevalent among children and slaves in the West at the period of this story -- that is to say, thirty or forty years ago.

Although my book is intended mainly for the entertainment of boys and girls, I hope it will not be shunned by men and women on that account, for part of my plan has been to try to pleasantly remind adults of what they once were themselves, and of how they felt and thought and talked, and what queer enterprises they sometimes engaged in.

...

#### Letter Count Total

```
#!/usr/bin/perl
# Letter count total
```

```

my $lc = 0;

while (<>) {
    while (/[a-zA-Z]/) { ++$lc; $_ = $'; }
}
print "$lc\n";

# ./letter-count-total.pl TomSawyer.txt
# 296605

```

### Letter Frequencies

```

#!/usr/bin/perl
# Letter frequencies

while (<>) {
    while (/[a-zA-Z]/) {
        my $l = $&; $_ = $';
        ${$l} += 1;
    }
}

for (keys %f) { print "$_ ${f{$_}}\n" }

```

### Letter Frequencies Output

```

./letter-frequency.pl TomSawyer.txt
S 606
a 22969
T 1899
N 324
K 24
d 14670
Y 214
E 158
j 381
y 6531
u 8901
...

```

### Letter Frequencies Modification

```

#!/usr/bin/perl
# Letter frequencies (2)

while (<>) {
    while (/[a-zA-Z]/) {
        my $l = $&; $_ = $';
        ${lc $l} += 1;
    }
}

```

```

}

for (sort keys %f) { print "$_ $f{$_}\n" }

```

### New Output

```

./letter-frequency2.pl TomSawyer.txt
a 23528
b 4969
c 6517
d 14879
e 35697
f 6027
g 6615
h 19608
i 18849
j 639
k 3030
...

```

We will look now at an implementation where letters and their frequencies are sorted by the frequency, from the highest-frequency letter to the lowest. We will also produce frequencies both as letter counts, and as normalized frequencies; i.e., as proportional frequencies of the letters out of 1.

### Letter Frequencies Modification (3)

```

#!/usr/bin/perl
# Letter frequencies (3)

while (<>) {
    while (/[a-zA-Z]/) {
        my $l = $&; $_ = $';
        $f{lc $l} += 1; $tot ++;
    }
}

for (sort { $f{$b} <=> $f{$a} } keys %f) {
    print sprintf("%6d %.4lf %s\n",
        $f{$_}, $f{$_}/$tot, $_); }

```

### Output 3

```

35697 0.1204 e
28897 0.0974 t
23528 0.0793 a
23264 0.0784 o
20200 0.0681 n
19608 0.0661 h
18849 0.0635 i
17760 0.0599 s
15297 0.0516 r

```

14879 0.0502 d  
 12163 0.0410 l  
 8959 0.0302 u

...

## 6 Elements of Morphology

- Reading: Section 3.1 in the textbook, “Survey of (Mostly) English Morphology”
- *morphemes* — smallest meaning-bearing units
- *stems* and *affixes*; stems provide the “main” meaning, while affixes act as modifiers
- affixes: prefix, suffix, infix, or circumfix
- cliticization — clitics appear as parts of a word, but syntactically they act as words (e.g., ’m, ’re, ’s)
- tokenization, stemming (Porter stemmer), lemmatization

The *morphemes* are the smallest meaning-bearing parts of a word. For example, the word *cats* contains two morphemes *cat* and *s*, the word *unbelievably* contains the four morphemes *un*, *believ*, *ab*, and *ly*, and the word *unmorphologically* contains the six morphemes *un*, *morpho*, *ling*, *uist*, *ical*, and *ly*. It could be sometimes debatable what is the proper way of breaking a word into morphemes, but not having a clear correct answer is not uncommon in analysis of natural languages.

- suffix example: *eats*; prefix example: *unbuckle*; circumfix example from German: *sagen* (to say) and *geesagt* (said, past participle); infix example from Tagalog (Philippine language): *hingi* (borrow) and *humingi*
- stacking multiple affixes is possible: *unbelievably* = *un-believe-able-y*
- English typically allows up to 4 affixes, but some languages allow up to 10 affixes, such as Turkish. Such languages are called *agglutinative* languages.
- *cliticization* is considered to be a morphological process
- Clitics appear as orthographic or phonological parts of the words, but syntactically they act as words.
- Clitic examples: ’m in I’m, ’re in we’re, possessive ’s

### Tokenization

- Text processing in which plain text is broken into words or *tokens*
- Tokens include non-word units, such as numbers and punctuation
- Tokenization may normalize words by making them lower-case or similar
- Usually simple, but prone to ambiguities, as most of the other NLP tasks

**Tokenization** is text processing in which the plain text is broken into words. It may not be a simple process, depending on the type of text and kind of tokens that we want to recognize.

**Stemming** is the type of word processing in which a word is mapped into its *stem*, which is a part of the word that represents the main meaning of the word. For example, *foxes* is mapped to the stem *fox*, or the word *semantically* is mapped to the stem *semanti*.

It is used in Information Retrieval due to the property that if two words have the same stem, they are typically semantically very related. Hence, if words in documents and queries are replaced by their stems, the resulting indices are smaller, and words in a query can be easily matched with their morphological variations.