Faculty of Computer Science, Dalhousie University *7-Oct-2025*

CSCI 4152/6509 — Natural Language Processing

Lecture 4: Basic NLP with Perl

Location: Studley LSC-Psychology P5260 Instructor: Vlado Keselj
Time: 14:35 – 15:55

Previous Lecture

- Finite state automata (continued)
  - Deterministic Finite Automaton (DFA)
  - Non-deterministic Finite Automaton (NFA)
- Review of Deterministic Finite Automata (DFA)
- Non-deterministic Finite Automata (NFA)
- Implementing NFA, NFA-to-DFA translation
- Review of Regular Expressions

5.2 Introduction to Perl

We will first start with an introduction to Perl Programming language. Perl is covered in more details in the labs, and in the lab notes of the course. We will give here a quick overview, but as with any programming language, the best way to learn it is to use it in some exercises, or to solve some programming problems.

- Created in 1987 by Larry Wall
- Interpreted, but relatively efficient
- Convenient for string processing, system admin, CGIs, etc.
- Convenient use of Regular Expressions
- Larry Wall: Natural Language Principles in Perl
- Perl is introduced in lab in more details

Perl: Some Language Features

- interpreted language, with just-in-time semi-compilation
- dynamic language with memory management
- provides effective string manipulation, brief if needed
- convenient for system tasks
- syntax (and semantics) similar to:
  C, shell scripts, awk, sed, even Lisp, C++

Some Perl Strengths

- **Prototyping:** good prototyping language, expressive: It can express a lot in a few lines of code.
- **Incremental:** useful even if you learn a small part of it. It becomes more useful when you know more; i.e., its learning curve is not steep.
- **Flexible:** e.g, most tasks can be done in more than one way
- **Managed memory:** garbage collection and memory management

- **Open-source:** free, open-source; portable, extensible
- **RegEx support:** powerful, string and data manipulation, regular expressions
- **Efficient:** relatively, especially considering it is an interpreted language
- **OOP:** supports Object-Oriented style

Some Perl Weaknesses

- not as efficient as C/C++
- may not be very readable without prior knowledge
- OO features are an add-on, rather than built-in
- competing popular languages
- not a steep learning curve, but a long one
  (which is not necessarily a weakness)

Why Perl?

- We only cover Perl with regular expressions and basic text processing
- Perl is very convenient for these types of tasks
- Perl uses very direct way of using regular expressions
- Perl is still used a lot in text processing, NLP, bioinformatic string processing, etc.
- Perl-style regular expressions are very important in any programming languages for NLP

Perl in This Course

- Examples in lectures, but you are expected to learn used features by yourself
- Labs will cover more details
- Finding help and reading:
  - Web: `perl.com`, `CPAN.org`, `perlmonks.org`, ...
  - `man perl`, `man perlintro`, ...
  - books: e.g., the "Camel" book:
    "Learning Perl, 4th Edition" by Brian D. Foy; Tom Phoenix; Randal L. Schwartz (2005)

Testing Code (as shown in labs)

- Login to timberlea
- Use plain editor, e.g., emacs
- Develop and test program
- Submit assignments
- You can use your own computer, but code must run on timberlea

Perl File Names

- Extension '`.pl`' is common, but not mandatory
- `.pl` is used for programs (scripts) and basic libraries
- Extension '`.pm`' is used for Perl modules

5.2.1 "Hello World" and Other Simple Test Runs

"Hello World" Program

Choose your favorite editor and edit `hello.pl`:

```
print "Hello world!\n";
```

Type "`perl hello.pl`" to run the program, which should produce:

```
Hello world!
```

Another way to run a program

Let us edit again `hello.pl` into:

```
#!/usr/bin/perl
print "Hello world!\n";
```

Change permissions of the program and run it:

```
chmod u+x hello.pl
./hello.pl
```

Simple Arithmetic

```
#!/usr/bin/perl
print 2+3, "\n";
$x = 7;
print $x * $x,"\n";
print "x = $x\n";
```

Output:

```
5
49
x = 7
```

Direct Interaction with Interpreter

- Command: `perl -d -e 1`
- Enter commands and see them executed
- '`q`' to exit
- This interaction is through Perl debugger

5.2.2 Syntactic Elements of Perl

Syntactic Elements

- statements separated by semi-colon '`;`'
- white space does not matter except in strings
- line comments begin with '`#`'; e.g.
  `# a comment until the end of line`
- variable names start with $, @, or % ('sigils'):
  `$a` — a scalar variable
  `@a` — an array variable
  `%a` — an associative array (or hash)
  However: `$a[5]` is 5th element of an array `@a`, and
  `$a{5}` is a value associated with key 5 in hash `%a`

- the starting special symbol is followed either by a name (e.g., `$varname`) or a non-letter symbol (e.g., `$!`)
- user-defined subroutines are usually prefixed with &: `&a` — call the subroutine `a` (procedure, function)

Similarly to C, C++, and Java, the statements in Perl are separated by semi-colon (‘`;`’). White space is generally ignored, except in string literals. The comments are line comments marked by the hash sign (‘`#`’); i.e., text from a hash sign to the end of line is treated as a comment. The variable names start with one of the special symbols, `$`, `@` or `%`, which denote different variable types. For example, the name `$a` denotes a scalar variable, the name `@a` denotes an array variable or a list, and `%a` denotes an associative array or a hash in Perl terminology, and this structure is also called a *map* in Java or *dictionary* in Python. Although, `@a` is an array, the individual elements of the array are denoted as `$a[0]`, `$a[1]`, and so on; and similarly the individual elements of a hash `%a` are `$a{'key'}` and similar.

As in many other programming languages, the variable names start with a letter, followed by letters, digits, and underscore, such as `$some_variable2`; however, in Perl they can also consist of only one special character (non-letter), such as `$_` or `$&`. The variables with a name consisting of a special character usually have some special function in Perl. For example, the variable `$_` is known as the default variable, and many commands take it as an argument when argument is not specified.

When called, user-defined functions (or subroutines) in Perl are prefixed with ampersand (‘`&`’); e.g., `&f(12)`, or simply `&f` if called without arguments. Actually, in many cases we can call a function as `f(1,2)` but there are some differences than using `&f(1,2)`, until you understand them, it is recommended to use `&f(1,2)` for functions defined by you.

Example Program: Reading a Line

```
#!/usr/bin/perl
use warnings;

print "What is your name? ";
$name = <>;       # reading one line of input
chomp $name;      # removing trailing newline
print "Hello $name!\n";
```

`use warnings;` enables warnings — recommended!
`chomp` removes the trailing newline from `$name` if there is one. However, changing the special variable `$/` will change the behaviour of `chomp` too.

The special variable `$/` is called the input fields separator and it is `"\n"` by default. It determines what is a line read by the <> operator, and what is removed by `chomp`. If its value is the null-string (`''`) empty lines are delimiters, and `undef $/` is used if we want to read the whole file at once. Mnemonic for the variable names come from the use of slash (/) to delimit line boundaries when quoting poetry.

Example: Declaring Variables

The declaration “`use strict;`” is useful to force more strict verification of the code. If it is used in the previous program, Perl will complain about variable `$name` not being declared, so you can declare it: `my $name`

We can call this program `example3.pl`:

```
#!/usr/bin/perl
use warnings;
```

```
use strict;

my $name;
print "What is your name? ";
$name = <>;
chomp $name;
print "Hello $name!\n";
```

5.2.3 Regular Expressions in Perl

Regular Expressions in Perl

- We will learn more about Regular Expressions by using Perl in simple text processing.
- Let us start with a program to count lines in a text

Perl Program for Counting Lines

```
#!/usr/bin/perl
# program: lines-count.pl

while (<>) {
++$count;
}

print "$count\n";
```

Regular Expressions in Perl

- Perl provides an easy use of Regular Expressions
- Consider the regular expression: `/pro...ing/`
- Run the following commands on timberlea:
  ```
  cp ~prof6509/public/linux.words .
  grep proc...ing linux.words
  ```
- Output includes 'processing', and more:
  ```
  coprocessing
  food-processing
  microprocessing
  misproceeding
  multiprocessing
  ...
  ```

Perl provides an easy use use of Regular Expressions. Consider doing the following small exercise with the regular expression `/pro...ing/` on the timberlea server. First, you can copy the provided file `linux.word` to your current directory using the command: `cp ~prof6509/public/linux.words .`

Then, we can use grep to find the words matching the considered regular expression using command:
`grep proc...ing linux.words`

The output will include the word 'processing', but also 'proceeding' and more:

```
coprocessing
food-processing
microprocessing
misproceeding
```

```
multiprocessing
...
```

Note About File 'linux.words' and Others

- Some helpful files can be found on timberlea in:
  `~prof6509/public/`
- or, on the web at:
  `http://web.cs.dal.ca/~vlado/csci6509/misc/`
- For example:
  `linux.words`
  `wordlist.txt`
  `Natural-Language-Principles-in-Perl-Larry-Wall.pdf`
  `TomSawyer.txt`

We will not show a short Perl program with a similar functionality as 'grep':

Perl Regular Expressions: 'proc...ing' Example

- 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 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/'
```

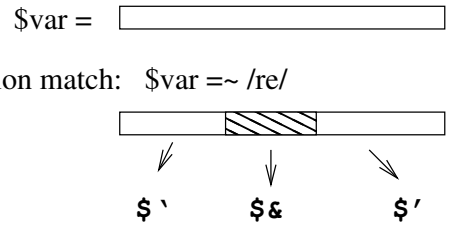
More Special Character Classes

`\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



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

`\G`	anchor, end of the previous match
`(?=re)`	look-ahead
`(?!re)`	negative look-ahead
`(?<=re)`	look-behind
`(?<!re)`	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 $a^n b a^n$; 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, where the parentheses pairs are counted according to the order of the opening parenthesis.

For example let us consider the following problem:

Problem example: Consider the regular expressions `/(a+(b+))(c+(d+))\4/` and `/(a+(b+))(c+(d+))\3/`. Write some examples of strings matched by these regular expressions. Characterize all string matched by these regular expressions.