Faculty of Computer Science, Dalhousie University *14-Sep-2023*

CSCI 4152/6509 — Natural Language Processing

Lecture 4: NFA, Regular Expressions Review, Perl

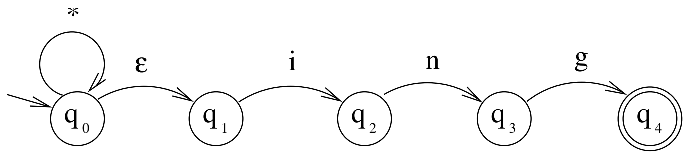
Location: Rowe 1011 Instructor: Vlado Keselj
Time: 16:05 – 17:25

Previous Lecture

- **Part II: Stream-based Text Processing**
- Finite state automata
  - 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 (started)

Example of Translating an NFA to DFA

Let us consider a language of words consisting of all lowercase English letters ending with 'ing', i.e., of all words that can be described with the regular expression `/.*ing/`. These words would be recognized by the following NFA:



In the NFA shown above, we use ε to denote an ε-transition, i.e., a transition that happens without reading any input symbol. The asterisk symbol ('`*`') is used to denote all letters. The shown NFA illustrates how we can conveniently express this language with an NFA. An issue is how to implement an NFA as they are more difficult to implement than DFAs. One solution would be to use backtracking to explore all possible states while reading a string. This is not so easy to implement and it may also be a very inefficient solution depending on an NFA. A more efficient solution usually is to translate this NFA into a DFA. Let us see how we can do this.

The states of an DFA produced from an NFA are sets of states from the NFA. The start state of the DFA is a set containing the start state of the NFA. In this case, it is the state q_0. We also add all states in the so-called 'closure' of q_0, which are all states that can be reached from q_0 via ε-transitions. In our particular case, it implies that q_1 is also a state in the closure of q_0. Hence, the start state of the DFA is the set $\{q_0, q_1\}$. We can start the DFA transition table as follows:

State	`i`	`n`	`g`	other letters (not `i`, `n`, or `g`)
$\rightarrow \{q_0, q_1\}$				

We now look at the states q_0 and q_1 and look at the states reachable from these states after reading '`i`'. These states are q_0, because it is reachable from q_0 via '`*` transition, and the state q_2, which is reachable from q_1. So the states reachable from $\{q_0, q_1\}$ after reading '`i`' character are the states $\{q_0, q_2\}$. We also need to include closure of this set, i.e., all states reachable via ε transitions from these states, so we finally get the set $\{q_0, q_1, q_2\}$. That is how we fill the first entry of the transition table:

State	i	n	g	other letters (not i, n, or g)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$			

We fill the rest of the row of the table in a similar way. These entries are less interesting since the corresponding letters lead only to the state q_0, and we add its closure to get $\{q_0, q_1\}$:

State	i	n	g	other letters (not i, n, or g)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

After finishing a row in the transition table, we need to check that all added table entries, which are sets of NFA states, appear in the first column of the transition table. We can notice that $\{q_0, q_1\}$ is in included in the column, while we have one new set $\{q_0, q_1, q_2\}$, which needs to be added to the table, in the first column. We may sometimes encounter more of these sets and we need to add them all. After adding this new set to the first column, we get the table:

State	i	n	g	other letters (not i, n, or g)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$				

We repeat the process with the new row of the table and obtain:

State	i	n	g	other letters (not i, n, or g)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

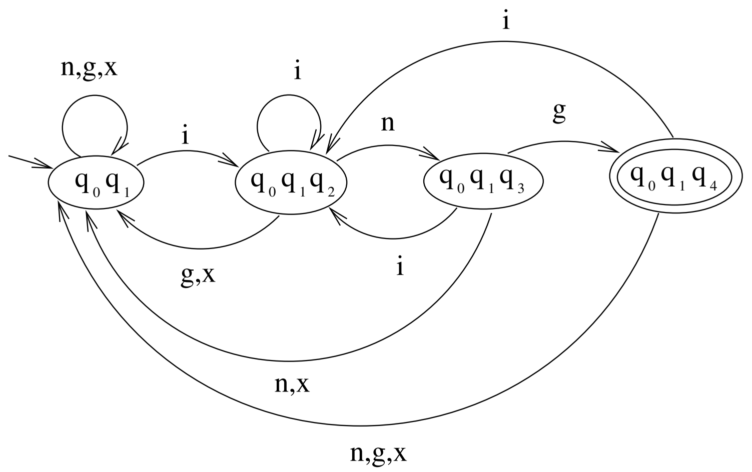
We now add any new sets of states to the first column and repeat the process until now new sets of states appear. We end up with the following table:

State	i	n	g	other letters (not i, n, or g)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_4\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_4\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

We now go through the first column and we mark all sets of states that include at least one final state from the NFA as final states in the DFA. The NFA had only one final state q_4, so we mark only the state $\{q_0, q_1, q_4\}$ as the final state in DFA, and finally obtain the following transition table:

State	i	n	g	other letters (not i, n, or g)
$\rightarrow \{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1, q_4\}$	$\{q_0, q_1\}$
F: $\{q_0, q_1, q_4\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$

If we want to represent this DFA as a graph, we would obtain:



It the above DFA we used ‘x’ to represent any letter other than ‘i’, ‘n’, or ‘g’.

It is now relatively easy to implement DFA in any programming language, and it would be a very efficient solution. A possible issue with this approach is that the new DFA could have exponentially more states that the original NFA. One solution would be to apply DFA minimization, which is a well-known procedure that we will not study here in any detail. Another approach is not to represent the DFA explicitly, as we did, but to implement NFA by keeping track of all reachable states after reading input, and whether any final state is included in this set of reachable states at the end.

Finite Automata in NLP

Slide notes:

Finite Automata in NLP

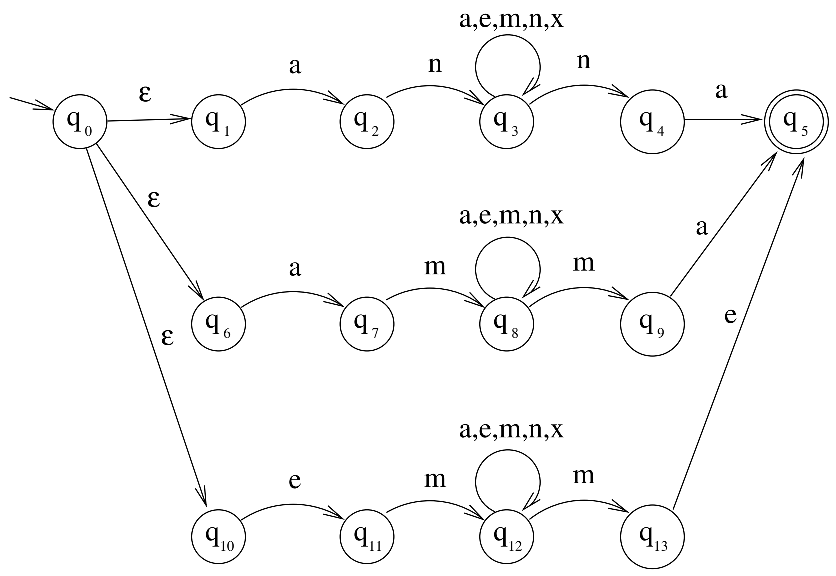
- Useful in data preprocessing, cleaning, transformation and similar low-level operations on text
- Useful in preprocessing and data preparation
- Efficient and easy to implement
- Regular Expressions are equivalent to automata
- Used in Morphology, Named Entity Recognition, and some other NLP sub-areas

Finite Automata—DFA, NFA, and their various variations—are very useful models for describing different kinds of language and string processing. They are easy to implement and they run efficiently. They are a frequently used model in morphological processing and in describing morphological processes of a language.

In addition to being useful as direct models for many NLP tasks, they are also an excellent way to implement regular expressions. Regular expressions are a convenient way of representing language patterns, but it is not obvious how to implement them in an algorithm. A solution is to translate a regular expession to NFA, then translate NFA to DFA, and then implement a DFA. All these steps are well-understood and straightforward.

Similarly to regular expressions, automata are useful in many text preprocessing tasks, such as text filering, cleaning, transformation, and similar.

Exercise Problem: Let us suppose that we want to implement a recognizer of all words that either:
1) start with ‘an’ and end with ‘na’,
2) start with ‘am’ and end with ‘ma’, or
3) start with ‘em’ and end with ‘me’. We will denote any other letters with ‘x’. An easy way to describe these words is with the following NFA:



Translate this NFA into a DFA.

Answer:

State	`a`	`e`	`m`	`n`	`x`
$\rightarrow \{q_0, q_1, q_6, q_{10}\}$	$\{q_2, q_7\}$	$\{q_{11}\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{q_2, q_7\}$	$\emptyset$	$\emptyset$	$\{q_8\}$	$\{q_3\}$	$\emptyset$
$\{q_{11}\}$	$\emptyset$	$\emptyset$	$\{q_{12}\}$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\{q_8\}$	$\{q_8\}$	$\{q_8\}$	$\{q_8, q_9\}$	$\{q_8\}$	$\{q_8\}$
$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$	$\{q_3\}$
$\{q_{12}\}$	$\{q_{12}\}$	$\{q_{12}\}$	$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_{12}\}$
$\{q_8, q_9\}$	$\{q_5, q_8\}$	$\{q_8\}$	$\{q_8, q_9\}$	$\{q_8\}$	$\{q_8\}$
$\{q_3, q_4\}$	$\{q_3, q_5\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$	$\{q_3\}$
$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_5, q_{12}\}$	$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_{12}\}$
F: $\{q_5, q_8\}$	$\{q_8\}$	$\{q_8\}$	$\{q_8, q_9\}$	$\{q_8\}$	$\{q_8\}$
F: $\{q_3, q_5\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3, q_4\}$	$\{q_3\}$
F: $\{q_5, q_{12}\}$	$\{q_{12}\}$	$\{q_{12}\}$	$\{q_{12}, q_{13}\}$	$\{q_{12}\}$	$\{q_{12}\}$

4.2 Regular Expressions

Regular Expressions

– Review (should have been covered in earlier courses as well)
– Used as patterns to match parts of text
– Equivalent to automata, although this may not be obvious
– Provide a compact, algebraic-like way of writing patterns
– Example: `/Submit (the )?file [A-Za-z.-]+/`

Slide notes:

Some References on Regular Expressions

You can find many references on Regular Expressions, including:

- Chapter 2 of the textbook [JM]
- Perl “Camel book” or many resources on Internet
- On timberlea server: ‘`man perlre`’ and ‘`man perlretut`’
- The same effect: ‘`perldoc perlre`’ and ‘`perldoc perlretut`’
- Or on the web: `http://perldoc.perl.org/perlre.html` and `http://perldoc.perl.org/perlretut.html`

Regular Expressions are an important topic in Computer Science, both from a theoretical perspective and as a practical tool. It is expected that it was studied by most students taking this course before, so we will do only a brief review of them here. If you are not familiar with regular expressions, there are many available resources to catch up and learn more. For example, the regular expressions are covered in Chapter 2 of the main course textbook (Jurafsky and Martin), and the Perl “Camel book” includes this topic. If you work on the `timberlea` server, or any other Linux computer, the manual pages command ‘`man perlre`’ includes a lot of information about Perl regular expressions, and ‘`man perlretut`’ presents an excellent tutorial on the topic.

Some interesting developments in the history of Regular Expressions are:

- Research by Stephen Kleene: regular sets, and the name of regular sets and regular expressions (1951),
- Implementation in QED by Ken Thompson (1968),
- Open-source implementation by Henry Spencer (1986),
- Use in Perl by Larry Wall (1987),
- Perl-style Regular Expressions in many modern programming languages.

Example Regular Expressions

- Literal: `/woodchuck/ /Buttercup/`
- Character class: `/./` (any character),
`/[wW]oodchuck/`, `/[abc]/`, `/[12345]/`
(any of the characters)
- Range of characters: `/[0-9]/`, `/[3-7]/`, `/[a-z]/`, `/[A-Za-z0-9_-]/`
- Excluded characters and repetition: `/[^()]+/`
- Grouping and disjunction: `/(Jan|Feb) \d?\d/`
- Note: `\d` is same as `[0-9]`
- Another character class: `\w` is same as `[0-9A-Za-z_]` (‘word’ characters)
- Opposite: `\W` same as `[^0-9A-Za-z_]`

RegEx Examples: Anchors, Grouping, Iteration

```
/^This is a/   # use of anchor
/This^or^that/ # not an anchor
/woodchucks?/
/\bcolou?r\b/       # anchor \b
/is a sentence\.$/  # end of string anchor

# Grouping and iteration:
/This sentence goes on(, and on)*\.$/
/cat|dog/           # disjunction (alternation)
/The (cat|dog) ate the food\./
```

5 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)

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

- 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

5.1 “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 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 calle 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";
```

Perl Program for Counting Lines

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

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

print "$count\n";
```