



University of
St Andrews

CS5012 LANGUAGE AND COMPUTATION

Lecture Notes

APRIL 29, 2019

Lecturer:
Mark-Jan Nederhof

Submitted By:
Sizhe Yuen

Contents

1	Introduction	2
1.1	Motivation	2
1.2	Topics	2
1.3	Ambiguity	2
2	Regular expressions and automata	3
2.1	Basic regular expression patterns	3
2.2	Regular languages	4
2.3	Finite-state automata	5
2.4	Finite-state transducers	6
3	Morphology	6

1 Introduction

Natural language processing is the idea of getting computers to perform interesting tasks involving human languages. The goal is to provide insights in how languages are processed.

1.1 Motivation

There is a wealth of written knowledge, especially on the Web in a machine readable form. To be able to process and understand this knowledge would be greatly beneficial. By improving machine processing of language, both human-machine communication and human-human (translation) communication can be improved.

1.1.1 Example applications

Small scale:

- Spelling correction
- Hyphenation

Medium scale:

- Web analytics - data mining web pages

Large scale:

- Machine translation
- Text summarisation
- Question/answer conversation

1.2 Topics

- Phonology - how sounds make speech
- Morphology - components of words
- Syntax - how to put words together
- Semantics - meaning of words and sentences (lexical and compositional semantics)
- Pragmatics - goals and intentions of the speaker
- Discourse - how to put utterances together to give meaningful communication

1.3 Ambiguity

A major problem that all language processing tasks must try to solve is ambiguity at some level. An input is **ambiguous** if *there are multiple alternative linguistic structures that can be built for it*. Humans generally resolve ambiguity naturally and are unaware of the ambiguity through the use of context, world knowledge, common sense, etc.

For example, the sentence “*I made her duck*” can have multiple meanings:

- I cooked waterfowl for her (to eat)
- I cooker waterfowl belonging to her
- I created the (plaster?) duck she owns

- I caused her to quickly lower her head or body
- I waved my magic wand and turned her into undifferentiated waterfowl

This happens as each word in the sentence can have multiple meanings.

- **duck** - can be either a noun or a verb
- **her** - can be possessive or dative pronoun
- **make** - can mean ‘create’, ‘cook’ or ‘force’
- **make** - can be used as a transitive or ditransitive verb

Models and algorithms used in language processing often try to resolve or disambiguate these cases. For example, deciding whether ‘duck’ is a noun or a verb can be solved by **part-of-speech tagging**. This is a kind of **lexical disambiguation**. In contrast, deciding whether ‘her’ and ‘duck’ are part of the same entity is an example of **syntactic disambiguation**, which can be addressed by probabilistic parsing.

2 Regular expressions and automata

Regular expressions are a powerful way to search texts in strings. Most programming languages support regular expressions natively in some way. They work by searching for patterns and replacing the pattern with another.

For example, the following searches can be performed on the word ‘woodchuck’:

- `/d/` matches the `d` in woodchuck
- `/s/` does not matching anything in woodchuck
- `/ck/` matches the last two letters in woodchuck

2.1 Basic regular expression patterns

Regular expressions are case sensitive, so a lowercase `/s/` will not match words with an upper case `S`. Square brackets can be used to specify a **disjunction** of characters to match. For example, the `/[wW]/` pattern will match either an upper or lower case `w`. A dash (`-`) can also be used to specify any one character in a range. For example, `/[2-5]/` specifies any one of the characters `2`, `3`, `4` or `5`.

RE	Match	Example pattern
<code>/[wW]oodchuck/</code>	Woodchuck or woodchuck	<code>"<u>W</u>oodchuck"</code>
<code>/[abc]/</code>	‘a’, ‘b’ or ‘c’	<code>"In uo<u>m</u>ini, in sold<u>a</u>ti"</code>
<code>/[1234567890]/</code>	a digit	<code>"plenty of <u>7</u> to 5"</code>
<code>/[0-9]/</code>	a digit	<code>"plenty of <u>7</u> to 5"</code>
<code>/[A-Z]/</code>	an uppercase letter	<code>"we should call it <u>D</u>renched"</code>

Table 1: Example regular expressions using square brackets `[]`.

Here are a few basic rules and expressions:

- `/[^abc]/` matches any character other than `a`, `b` or `c` using the caret ‘`^`’
- `/./` is a **wildcard** that matches any character
- `/[abc]+/` matches one or more of `a`, `b` or `c`

- `/[0-9]*/` matches zero or more digits

There are also a number of aliases for sets of characters that can be used as a shortcut.

- `\d` for any digit (same as `[0-9]`)
- `\D` for any character other than a digit (same as `^[0-9]`)
- `\w` for any alphanumeric or underscore
- `\W` converse of the above
- `\s` for any whitespace character (spaces and tabs)
- `\S` converse of the above

Repetition can be matched using curly brackets

- `/\d{2,5}/` matches between 2 and 5 digits
- `/\s{3,}/` matches 3 or more whitespaces
- `/\d{4}/` matches exactly 4 digits

Anchors can be used to match places in a string rather than characters. For example, `\b` can match word boundaries so that an expression such as `/\bthe\b/` does not match the three letters **the** inside **other**.

- `/^/` matches beginning of input string or beginning of line
- `/$/` matches end of input string or end of line
- `/\b/` matches a word boundary

The power of regular expressions can be captured by a small number of operators for concatenation, union and Kleene closure. This defines the class of **regular languages**.

2.2 Regular languages

The class of regular languages is formally defined as:

1. \emptyset and $\{\epsilon\}$ are regular languages (ϵ stands for the **empty string**)
2. $\{a\}$ is a regular language for any symbol a in the ‘alphabet’
3. If L_1 and L_2 are regular languages, then so are:
 - (a) $L_1 \cdot L_2 = \{xy | x \in L_1, y \in L_2\}$ (concatenation)
 - (b) $L_1 \cup L_2$ (union/disjunction)
 - (c) L_1^* (Kleene closure)

Nothing else is a regular language unless it can be obtained through the above rules. Regular languages are the languages characterisable by regular expressions, so all regular expression operators can be implemented by the three operations which define regular languages.

Regular languages are closed under the following operations

- **intersection** - if L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$, the language consisting of the set of strings that are in *both* L_1 and L_2 .
- **difference** - if L_1 and L_2 are regular languages, then so is $L_1 - L_2$, the language consisting of the set of strings that are in L_1 but not in L_2 .
- **complementation** - if L_1 is a regular language, then so is $\Sigma^* - L_1$ where Σ^* is the infinite set of all possible strings formed by the alphabet Σ and $\Sigma^* - L_1$ is the set of all possible strings that aren't in L_1 .

- **reversal** - if L_1 is a regular language, then so is L_1^R , the language consisting of the set of reversals of all the strings in L_1

The proof that regular expressions are equivalent to finite-state automata has two parts: showing that an automaton can be built for any regular expression and showing that a regular language can be built for any automaton.

2.3 Finite-state automata

A finite automata (FA) consists of:

- a finite set Q of **states**
- a finite set Σ of input symbols (**alphabet**)
- an **initial state** $q_0 \in Q$
- a set $F \subseteq Q$ of **final states**
- a set Δ of **transitions**

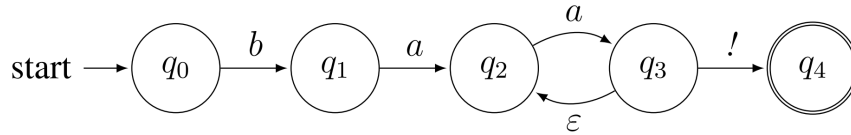


Figure 1: Example of a finite-state automaton with an epsilon transition.

A transition has the form (q, a, q') which says, go to state q' from q by reading the next symbol a . An epsilon transition (q, ϵ, q') allows a transition between the states without reading an input symbol.

A string of input symbols is **accepted** by an automaton if it can go from an initial state to a final state by reading all the symbols left to right and following a valid sequence of transitions. Therefore, the language accepted by a finite-state automaton is the set of strings that is accepted by it.

2.3.1 Deterministic vs. non-deterministic

The difference between deterministic and non-deterministic finite automata is the existence of epsilon transitions and valid transitions - that is to say, for each q and a there is at most one q' such that $(q, a, q') \in \Delta$.

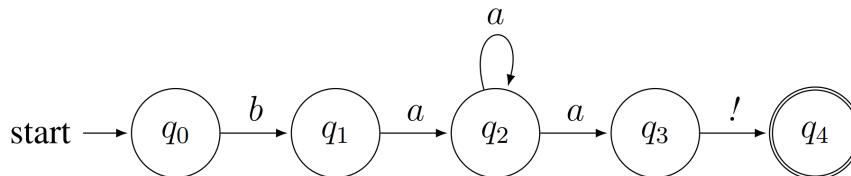


Figure 2: Example of a non-deterministic finite-state automaton. In q_2 when the symbol a is read, it can either move to q_2 or q_3 .

It is not the case that non-deterministic finite-state automata are more powerful than deterministic automata. For any NFSA, there is exactly one equivalent DFSA, although the number of states in the equivalent DFSA may be much larger. This is done through the construction of a powerset, where each state of M' represents a set of states of M .

2.3.2 Recognition

Assuming an input of length n . An input position is a number from 0 to n .

- A **configuration** is defined to be a pair consisting of a state and an input position
- The **initial configuration** is $(q_0, 0)$
- A configuration (q, i) is **accepting** if $q \in F$ and $i = n$
- The **agenda** is a set of configurations, which is initially empty.

Recognition can be run by adding and remove configurations from the agenda on every iteration, then if the agenda ever becomes empty, we have run out of valid configurations, so the recognition can stop and return a failure. If the agenda ever contains an accepting configuration, then we can stop and return a success.

2.3.3 Regular expressions and finite automata

For each regular expression, we can construct an equivalent finite automata.

Proof by induction:

- Show that the FSAs for \emptyset , $\{\epsilon\}$ and $\{a\}$ for every symbol a can be constructed.
- Show that if we have FSAs accepting L_1 and L_2 , then we can construct three FSAs accepting $L_1 \cdot L_2$, $L_1 \cup L_2$ and L_1^* respectively.

Another important point for closure is that FSA languages are closed under concatenation, union and Kleene star.

2.4 Finite-state transducers

3 Morphology

Morphology is the study of the ways in which words are built up of smaller **morphemes**. There are two general units of morphemes:

- **stems** - the core meaning-bearing units
- **affixes** - bits and pieces that adhere to stems to change their meanings and grammatical functions