Computational Linguistics: Introduction

Raffaella Bernardi

KRDB, Free University of Bozen-Bolzano

P.zza Domenicani, Room: 2.28, e-mail: bernardi@inf.unibz.it

Contents

1. Course Info

- **Time**: Thursdays 08:30-10:30 (Lessons) and Thursdays: 14:00-15:00 (Labs –TBC)
- **Office hours**: February-June: Thursdays 10:30-12:30, later by prior arrangement via e-mail.
- **Course Materials**: Slides, Readings (**study the book!**)
- **Reference Material**:
  1. D. Jurasfky and J. H. Martin **Speech and Language Processing**. (see nr. of chapters on the web.).
  2. P. Blackburn and J. Bos (BB1) (see nr. of chapters on the web) **Representation and Inference for Natural Language A First Course in Computational Semantics**
  3. P. Blackburn and J. Bos (BB2) (see nr. of chapters on the web) **Working with Discourse Representation Structures**
  4. P. Blackburn and K. Striegnitz (BS) (online) **Natural Language Processing Techniques in Prolog** (??????)
  5. P. Blackburn, J. Bos and K. Striegnitz. **Learn Prolog Now!**. (?????)
- **Url**: `http://www.inf.unibz.it/~bernardi/Courses/CompLing/06-07.html`

1.1. Grading

1. **Projects**: You are to complete an independent project on some topic in CL that must include a careful write-up or oral presentation (overview of the literature, a critique of a selected paper and a description of your own idea/implementation). [50%]
2. **Final Exam**: Written exercises on the topics discussed in class. [50%]

Calendar Last lecture: May 10th. Last Lab (project presentation): May 17th. Final exam: June 12th (TBC).

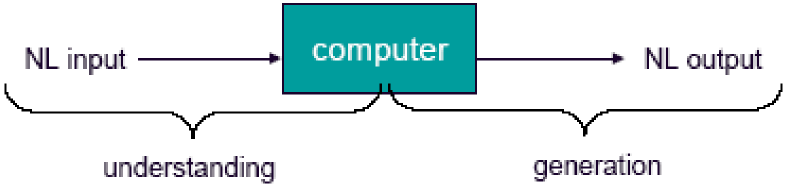
Hours of your work This is a 4 ECTS (100 hs): **36 hs** with me, **64 hs** on your own.

1.2. Program

- **Feb.-March**: Fundamentals of Linguistics and Computational Lingusitics with emphasis on: Morphology, Syntax, Parsing and Semantics.
- **April-May**: Discussion of more challenging linguistic phenomena and analysis of some solution proposed in the literature, recently.

2. Goals of Computational Linguistics

- **Ultimate goal**: To build computer systems that perform as well at using natural language as humans do.
- **Immediate goal** To build computer systems that can process text and speech more intelligently.



where, NL (**Natural Language**) is the language that people use to communicate with one another and **process** means to analyze.

3. The study of Natural Language

Natural Language is studied in several different academic disciplines, and each of them has its set of problems and tools.

Discipline	Typical Problems	Tools
Linguistics	How do words form phrases and sentences? What constrains the possible meanings for a sentence?	Intuitions about well formedness and meaning; mathematical models of structure
Psycoling.	How do people identify the sentence structures? How are word meanings identified? When does understanding take place?	Experimental techniques based on measuring human performance; statistical analysis of observations
Philosophy	What is meaning? How do words and sentences acquire it? How do words identify objects in the world?	Natural language argumentation using intuition about counter-examples; math. models (eg. logic and model the
Com. Ling.	How is the structure of sentences identified? How can knowledge and reasoning be modeled? How can language be used to accomplish specific tasks?	Algorithms, data structures; formal models of representation and reasoning; AI techniques; math. models

4. Why computational models of NL

There are two motivations for developing computational models:

- **Scientific**: To obtain a **better understanding** of how language works. Computational models may provide very specific predictions about human behavior that can then be explored by the phsycholinguist.
- **Technological**: natural language processing capabilities would revolutionize the way computers are used. Computers that could understand natural language could access to all human knowledge. Moreover, **natural language interfaces** to computers would allow complex systems to be accessible to everyone. In this case, it does not matter if the model used reflects the way humans process language. It only matters that it works.

We are interested in **linguistically motivated computational models** of language understanding and production that can be shown to perform well in specific example domains.

5.1. Ambiguity: Phonology

Phonology: It concerns how words are related to the sounds that realize them. It's important for speech-based systems.

1. "I scream"
2. "ice cream"

5.2. Ambiguity: Morphology

Morphology: It's about the inner structure of words. It concerns how words are built up from smaller meaning-bearing units.

1. Unionized (characterized by the presence of labor unions)
2. un-ionized in chemistry

5.3. Ambiguity: Syntax

Syntax: It concerns sentence structure. Different syntactic structure implies different interpretation.

1. I saw the man with the telescope
   - ▶ $[\mathrm{I}[[\mathrm{saw}]_v[\mathrm{the\ man}]_{np}[\mathrm{with\ the\ telescope}]_{pp}]_{vp}]_s$ [(I have the telescope)]
   - ▶ $[I[[\mathrm{saw}]_v[[\mathrm{the\ man}]_{np}[\mathrm{with\ the\ telescope}]_{pp}]_{np}]_{vp}]_s$ [(the man has the telescope)]
2. Visiting relatives can be tiring.

5.4. Ambiguity: Semantics

Semantics: It concerns what words mean and how these meanings combine to form sentence meanings.

1. Visiting relatives can be tiring.
2. Visiting museums can be tiring.

Same set of possible syntactic structures for this sentence. But the meaning of museums makes only one of them plausible.

5.5. Ambiguity: Discourse

Discourse: It concerns how the immediately preceding sentences affect the interpretation of the next sentence

1. **Merck & Co.** formed **a joint venture** with **Ache Group**, of Brazil. **It** will ...?
2. Merck & Co. formed a joint venture with Ache Group, of Brazil. **It**$_i$ will be called Prodome Ltd.

   (**a joint venture!**$_i$)
3. Merck & Co. formed a joint venture with Ache Group, of Brazil. **It**$_i$ will own 50% of the new company to be called Prodome Ltd.

   (**Merck & Co.**$_i$!)
4. Merck & Co. formed a joint venture with Ache Group, of Brazil. **It**$_i$ had previously teamed up with Merck in two unsuccessful pharmaceutical ventures.

   (**Ache Group**$_i$!)

6. NLP Systems: Tokenization

1. Tokenization
2. PoS tagging
3. Morphological analysis
4. Shallow parsing
5. Deep parsing
6. Semantic representation (of sentences)
7. Discourse representation

Tokenization It consists in dividing the sequence of symbols in minimum units called tokens (words, date, numbers, punctation etc..). Many difficulties: e.g.

Sig. Rossi vs. 05.10.05 vs. www.unibz.it;

given up (multi words 1 token).

7. Words: Classes

Traditionally, linguists classify words into different categories:

- **Categories**: words are said to belong to *classes*/categories. The main categories are nouns (n), verbs (v), adjectives (adj), articles (art) and adverbs (adv).

The class of words can be divided into two broad supercategories:

1. **Closed Class**: Those that have relatively fixed membership. E.g. prepositions, pronouns, particles, quantifiers, coordination, articles.
2. **Open Class**: nouns, verbs, adjectives, adverbs.

8. Words: Classes (Cont'd)

A word in any of the four open classes can be used to form the basis for a phrase. This word is called the **head** of the phrase and indicates the type of thing, activity, or quality that the phrase describes. E.g. "dog" is the head in: "The dog", "the small dog", "the small dog that I saw".

- **Constituents**: Groups of categories may form a single *unit or phrase* called constituents. The main phrases are noun phrases (*np*), verb phrases (*vp*), prepositional phrases (*pp*). Noun phrases for instance are: "she"; "Michael"; "Rajeev Goré"; "the house"; "a young two-year child".

Tests like substitution help decide whether words form constituents.

Can you think of another test?

See Jurafsky & Martin, pp. 289-296 for more details on the single categories and phrases.

8.1. Applications of PoS tagging

More recently, linguists have defined classes of words, called **Part-of-Speech** (PoS) tagsets with much larger numbers of word classes. PoS are used to label words in a given collection of written texts (Corpus). These labels turn out to be useful in several language processing applications.

- **Speech synthesis**: A word's PoS can tell us something about how the word is pronounced. E.g. "content" can be a noun or an adjective, and it's pronounced differently: CONtent (noun) vs. conTENT (adjective).
- **Information Retrieval**: A word's PoS can tell us which morphological affixes it can take, or it can help selecting out nouns or other important words from a document.
- **Theoretical Linguistics**: Words' PoS can help finding instances or frequencies of particular constructions in large corpora.

PoS tagging techniques is a topic of the "Text Processing" course (ITC-irst: Bernardo Magnini)

9. Morphology

Morphology is the study of how words are built up from smaller meaning-bearing units, morphemes. It concerns the inner structure of words.

For instance,

- **fog**: it's one morphem
- **cats**: it consists of two morphemes: cat + -s.

9.1. Morphemes

Morphemes are divided into:

1. stems: they are the main morpheme of the word, supplying the main meaning.
2. affixes: they add additional meanings of various kinds. They are further divided into:
   - **prefixes**: precede the stem (English: unknown= un + known)
   - **suffixes**: follow the stem (English: eats= eat + -s)
   - **circumfixes**: do both (German: gesagt (said)= ge + sag + t)
   - **infixes**: are inserted inside the stem (Bontoc -Philippines - fikas (strong), fumikas (to be strong))

A word can have more than one affixes (e.g. re+write+s, unbelievably= believe (stem), un-, -able, -ly).

9.2. Ways of forming new words

There are two basic ways used to form new words:

1. **Inflectional forms**: It is the combination of a word stem with a grammatical morpheme, usually resulting in a word of the **same class as the original stem**, and usually filling some syntactic function like agreement. E.g. in English,

   past tense on verbs is marked by the suffix “-ed”, form by “-s”, and participle by “-ing”.

2. **Derivational forms**: It is the combination of a word stem with a grammatical morpheme, usually resulting in a word of a **different class**, often with a meaning hard to predict exactly. E.g.

   Adverbs from noun: friendly from friend.

   Noun from verbs: killer from kill.

   Adjectives from nouns: “computational” from “computation”, “unreal” from “real”.

10. Computational Morphology

We want to build a system able to provide the stem and the affixes given a word as input (e.g. cats $\longrightarrow \{\text{cat} + N + PL\}$), or able to generate all the possible words made of a given stem (e.g. cat $\longrightarrow \{\text{cats}, \text{cat}\}$). To this end, we first of all need to have a way to formally represent Morphology Theory studied by Linguists.

10.1. Modules

To build a morphological recognizer/generator, we'll need at least the following:

lexicon: the list of stems and affixes, together with basic information about them (e.g. Noun stem or Verb stem).

morphotactics: the model of the morpheme ordering, e.g. English plural morpheme follows the noun rather than preceding it.

orthographic rules: spelling rules used to model the changes that occur in a word, e.g. city becomes cities, i.e. “y” $\rightsquigarrow$ “ie”.

10.2. The Lexicon and Morphotactics

Lexicon: It's a repository of words. Having an explicit list of every word is impossible, hence the lexicon is structured with a list of each of the stems and affixes of the language.

Morphotactics: One of the most common way to model morphotactics is by means of **Finite State Automata** (FSA).

11. Background Notions

Before looking at how FSA are used to recognize/generate natural language morphology we need to introduce some background notions, namely **Formal Languages** and **FSA**.

Remark: The topics of this section are treated in details in Nievergelt's course **Formal Languages** (2nd year BSc) and in Calvanese's course **Theory of Computing** (1st year MSc). I just repeat some of their slides and give the intuitions for the students who have not attended their courses.

11.1. Formal Languages

Formal Language Theory considers a Language as a mathematical object.

- A Language is just a **set of strings**. To formally define a Language we need to formally define what are the strings admitted by the Language. Formal notions:

  1. **Alphabet**: A set of symbols, indicated by V (e.g., $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$).
  2. **String**: A string over an alphabet, V, is a sequence of symbols belonging to the alphabet (e.g., "518" is a string over the above V). The empty string is denoted by ϵ.
  3. **Linguistic Universe**: Indicated by V^*, denotes the set of all possible strings over V, including ϵ. The set V^+ denotes the set $V - \{\epsilon\}$.

- To characterize a Language means to find a finite representation of all admissible strings.

11.1.1. Concatenation

We have said that a language is a set of strings. An important operation on strings is concatenation.

- At syntactic level, strings are words that are concatenated together to form phrases.
- At morphological level, strings are morphemes that are concatenated to form words. E.g.

Stem Language:	$\{work, talk, walk\}$.
Suffix Language:	$\{\epsilon, -ed, -ing, -s\}$.

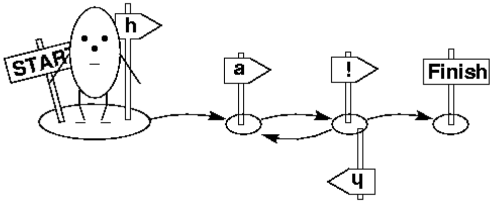
The concatenation of the Suffix language after the Stem language, gives:

$$\{work, worked, working, works, talk, talked, talking, talks, \\ walk, walked, walking, walks\}$$

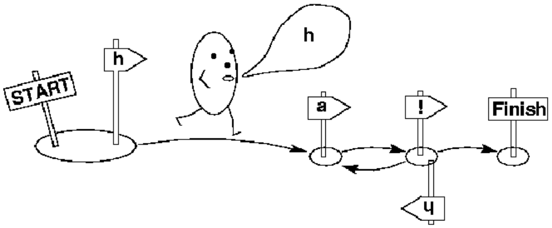
11.2. Finite State Automata

A finite state **generator** is a simple computing machine that **outputs** a sequence of symbols.

It starts in some **initial state** and then tries to reach a **final state** by making transitions from one state to another.



Every time it makes such a transition it emits (or writes or generates) a symbol.



It has to keep doing this until it reaches a **final state**; before that it cannot stop.

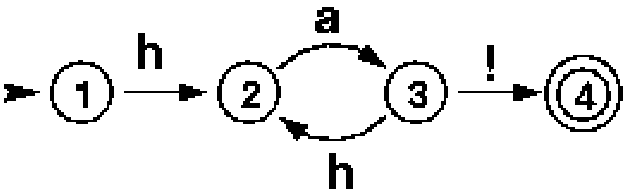
So, what does the generator in the pictures say?

It laughs: It generates sequences of symbols of the form `ha!` or `haha!` or `hahaha!` or `hahahaha!` and so on.

Why does it behave like that? Well, it first has to make a transition emitting `h`. The state that it reaches through this transition is not a final state. So, it has to keep on going emitting an `a`. Here, it has two possibilities: it can either follow the `!` arrow, emitting `!` and then stopping in the final state or it can follow the `h` arrow emitting an `h` and going back to the state where it just came from.

11.2.1. FSA as directed graph

Finite state generators can be thought of as **directed graphs**. And in fact finite state generators are usually drawn as directed graphs. Here is our laughing machine as we will from now on draw finite state generators:



The nodes of the graph are the states of the generator. We have numbered them, so that it is easier to talk about them. The **arcs** of the graph are the transitions, and the **labels** of the arcs are the symbols that the machine emits. A double circle indicates that this state is a final state and the one with the black triangle is the start.

11.2.2. Finite State Recognizer

Finite state recognizers are simple computing machines that **read** (or at least try to read) **a sequence of symbols** from an input tape. That seems to be only a small difference, and in fact, finite state generators and finite state recognizers are exactly the same kind of machine. Just that we are using them to output symbols in one case and to read symbols in the other case.

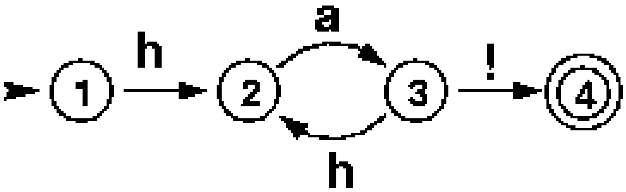
An FSA recognizes (or accepts) a string of symbols if starting in an intial state it can read in the symbols one after the other while making transitions from one state to another such that the transition reading in the last symbol takes the machine into a final state.

That means an FSA **fails** to recognize a string if:

- it cannot reach a final state; or
- it can reach a final state, but when it does there are still unread symbols left over.

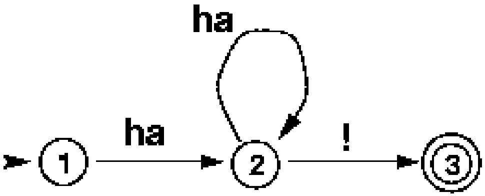
11.2.3. Recognizer: an example

So, this machine recognizes a laughter.

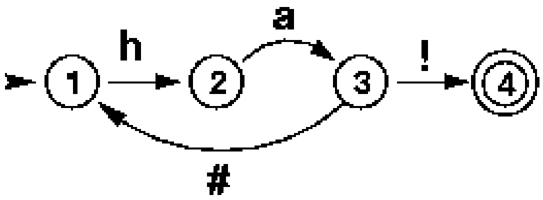

$$\begin{tikzcd} \arrow[r] & 1 \arrow[r, "h"] & 2 \arrow[r, bend left, "a"] & 3 \arrow[l, bend left, "h"] \arrow[r, "!"] & \boxed{4} \end{tikzcd}$$

For example, it accepts the word `ha!` by going from state 1 via state 2 and state 3 to state 4. At that point it has read all of the input and is in a final state. It also accepts the word `haha!` by making the following sequence of transitions: state 1, state 2, state 3, state 2, state 3, state 4. Similarly, it accepts `hahaha!` and `hahahaha!` and so on. However, it does not accept the word `haha?`. Although it will be able to read the whole input (state 1, state 2, state 3, state 2, state 3), it will end in a non-final state without anything left to read that could take it into the final state. So, when used in recognition mode, this machine **recognizes exactly the same words that it generates**, when used in generation mode. This is something which is true for all finite state automata.

11.2.4. Finite State Automata Try to think of what language is recognized or generated by the FSA below.



11.2.5. Finite State Automata with jumps



It has a strange transition from state 3 to state 1 which is reading/emitting **#**. We will call transitions of this type **jump arcs** (or ϵ transitions). Jump arcs let us jump from one state to another **without emitting or reading a symbol**. So, **#** is really just there to indicate that this is a jump arc and the machine is not reading or writing anything when making this transition.

This FSA accepts/generates the same language as our first laughing machine, namely sequences of `ha` followed by a `!`. Try it yourself.

11.2.6. Important properties of FSA

- All in all, finite state generators can only have a **finite number of different states**, that's where the name comes from.
- Another important property of finite state generators is that **they only know the state they are currently in**. That means they cannot look ahead at the states that come and also don't have any memory of the states they have been in before or the symbols that they have emitted.
- An FSA can have several intial and final states (it must have at least one initial and one final state, though).

11.3. Summing up: Formal Language & FSA

- A formal language is a set of strings. E.g. $\{a, b, c\}$, $\{the, a, student, students\}$.
- Strings are by definition finite in length.
- The language accepted (or recognized) by an FSA is the set of all strings it recognizes when used in recognition mode.
- The language generated by an FSA is the set of all strings it can generate when used in generation mode.
- The language accepted and the language generated by an FSA are exactly the same.
- FSA recognize/generate Regular Language.

11.3.1. Regular Language

Recall: V^* denotes the set of all strings formed over the alphabet V. A^* denotes the set of all strings obtained by concatenating strings in A in all possible ways.

Given an alphabet V,

1. $\{\}$ is a regular language
2. For any string $x \in V^*$, $\{x\}$ is a regular language.
3. If A and B are regular languages, so is $A \cup B$.
4. If A and B are regular languages, so is AB.
5. If A is a regular language, so is A^*.
6. Nothing else is a regular language.

Examples For example, let $V = \{a, b, c\}$. Then since aab and cc are members of V^* by 2, $\{aab\}$ and $\{cc\}$ are regular languages. By 3, so is their union, $\{aab, cc\}$. By 4, so is their concatenation $\{aabcc\}$. Likewise, by 5 $\{aab\}^*$ $\{cc\}^*$ are regular languages.

11.3.2. Pumping Lemma

For instance, a non-regular language is, e.g., $L = \{a^n b^n \mid n \geqslant 0\}$. More generally, FSA cannot generate/recognize balanced open and closed parentheses.

You can prove that L is not a regular language by means of the **Pumping Lemma**.

Roughly note that with FSA you cannot record (**no memory**!) any arbitrary number of a's you have read, hence you cannot control that the number of a's and b's has to be the same. In other words, you cannot account for the fact that there exists a **relation of dependency** between a^n and b^n.

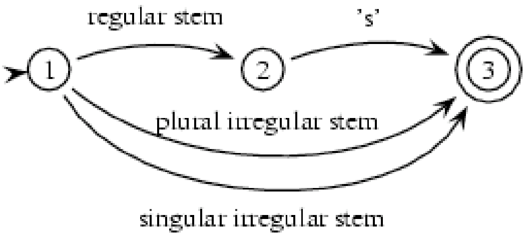
See Calvanese' s course for formal details.

12. FSA for Morphology Recognition/Generation

12.1. FSA for English Inflectional Morphology

Let's build an FSA that recognizes English nominal inflection. Our lexicon is:

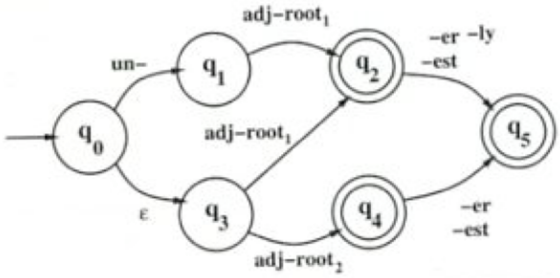
reg-stem	**plural**	**pl-irreg-stem**	**sing-irreg-stem**
fox	-s	geese	goose
cat		sheep	sheep
dog		mice	mouse



12.2. FSA for English Derivational Morphology

Let's build an FSA that recognizes English adjectives. Our lexicon is:

adj-root1	**adj-root2**	**Suffix-1-2**	**Suffix-1**	**Affix-1**
clear	big	-er	-ly	un-
happy	cool	-est		
real				


adj-root$_1$
un-
-er -ly
-est
adj-root$_1$
ε
-er
-est
adj-root$_2$
q_0
q_1
q_2
q_3
q_4
q_5

13. Recognizers vs. Parsers

We have seen that we can give a word to a recognizer and the recognizer will say "yes" or "no". But often that's not enough: in addition to knowing that something is accepted by a certain FSA, we would like to have an explanation of why it was accepted. Finite State Parsers give us that kind of explanation by returning the sequence of transitions that was made.

This distinction between recognizers and parsers is a standard one:

- **Recognizers** just say "yes" or "no", while
- **Parsers** also give an analysis of the input (e.g. a parse tree).

This distinction does not only apply to FSA, but also to all kinds of machines that check whether some input belongs to a language and we will make use of it throughout the course.

13.1. Morphological Parsers

The goal of morphological parsing is to find out what morphemes a given word is built from. For example, a morphological parser should be able to tell us that the word “cats” is the plural form of the noun stem “cat”, and that the word “mice” is the plural form of the noun stem “mouse”. So, given the string “cats” as input, a morphological parser should produce an output that looks similar to {cat N PL}.

Project Students who know about Finite State Transducers could carry out a project on their use as Morphological Parsers. See BS for more information. You should speak with me before submitting the project (better before starting it!).

14. What are FSA good for in CL?

Finite-state techniques are widely used today in both research and industry for natural-language processing. The software implementations and documentation are improving steadily, and they are increasingly available. In CL they are mostly "lower-level" natural language processing:

- Tokenization
- Spelling checking/correction
- Phonology
- Morphological Analysis/Generation
- Part-of-Speech Tagging
- "Shallow" Syntactic Parsing

Finite-state techniques cannot do everything; but for tasks where they do apply, they are extremely attractive.

In fact, the flip side of their **expressive weakness** being that they usually behave very well computationally. If you can find a solution based on finite state methods, your implementation will probably be **efficient**.

15. Practical Info

- Labs:

  1. Some paper-and-pencile exercises.
  2. Exercises with Prolog? (depending on your background)
  3. Reading groups (depending on your interests)

- Information Sheet, please fill it in and give it to me now.