Logic Programming (ITSE301)

Introduction to
Natural Language Processing

What is NLP?

- Natural Language Processing (NLP) is the study of human languages using computers.
- Human languages are studied by many research groups.
- As computer scientists we are interested in the algorithm and data structures that are useful for analyzing human languages.

NLP Goals

- Computers would be a lot more useful if they could handle our email, do our library search, talk to us …
- But this is not an easy task.
- How can we make computers handle human languages?

Some NLP Applications

- Spelling correction, grammar checking …
- Better search engines
- Information Extraction (IE)
- New interfaces:
  - Speech recognition (and text-to-speech)
  - Dialogue systems
  - Machine translation

Objectives

- The main goal of this serious of lectures is to introduce you to the NLP problems & solutions
- At the end you should:
  - Agree that NLP is interesting
  - Write small programs that analyze human languages

Language Levels
مستويات تحليل اللغة

- Phonetics/phonology: The study of sounds that make words علم الصوتيات
- Morphology: The study of written words and their structure.دراسة الكلمات المكتوبة
- Syntax: The study of the structure of phrases and sentences.دراسة تركيب الجمل والعبارات
- Semantics: The study of the literal meaning? دراسة المعاني
- Pragmatics: The study of sentences in their context "It's cold in here!" دراسة الجمل مع السياق الذي تذكر فيه

NLP problems: Ambiguity الغموض

- If there are more than one interpretation of a sentence then it is ambiguous.
- Ambiguity can arise at all levels of language processing
  - Morphology (can has many meaning)
  - Syntax: The girl eats the apple with a smile –The girl eats the apple with a bruise.
  - Semantics: Water runs down the hill. vs. The river runs down the hill.
  - Pragmatics: Can you pass the salt. (Sue!, Yes, No). It’s cold in here!

Syntax النحو

- Syntax is the study of the structure of sentences
- Syntactic objects are words, groups of words, syntactic categories such as NOUN and NOUN PHRASE, and syntactic roles such as SUBJECT and MODIFIER

Structure in Strings

- Some words: *the, a, small, nice, big, very, boy, girl, sees, likes, apples*
- Some good sentences:
  - the boy likes apples
  - the small girl likes the big girl
  - a very small nice boy sees a very nice boy
- Some bad sentences:
  - *the boy the apple
  - *small boy likes nice apples
- Can we find subsequences of words (**constituents**) which in some way behave alike?

Structure in Strings
Proposal 1

- Some words: *the a small nice big very boy girl sees likes cat*
- Some good sentences:
  - (the) boy (likes the cat)
  - (the small) girl (likes the big girl)
  - (a very small nice) boy (sees a very nice boy)
- Some bad sentences:
  - *(the) boy (the girl)
  - *(small) boy (likes the nice girl)

Structure in Strings
Proposal 2

- Some words: *the a small nice big very boy girl sees likes*
- Some good sentences:
  - (the boy) likes (the cat)
  - (the small girl) likes (the big girl)
  - (a very small nice boy) sees (a very nice boy)
- Some bad sentences:
  - *(the boy) (the cat)
  - *(small boy) likes (the nice girl)

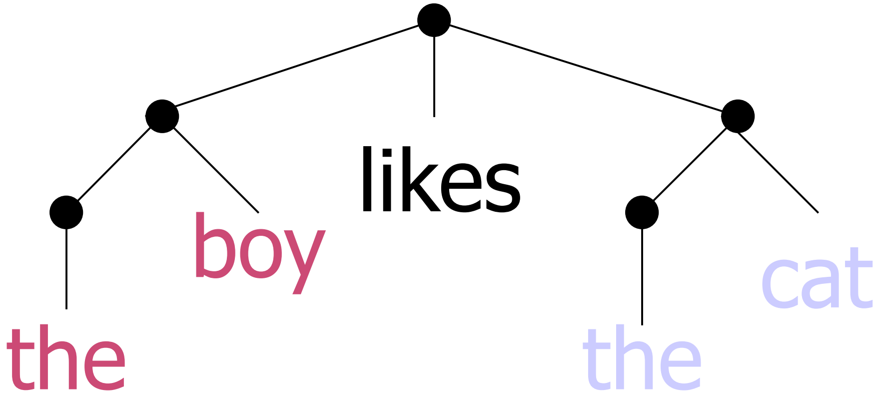
- This is better proposal: fewer types of constituents

More Structure in Strings

- Some words: *the a small nice big very boy girl sees likes cat*
- Some good sentences:
  - ((the) boy) likes ((the) cat)
  - ((the) (small) girl) likes ((the) (big) girl)
  - ((a) ((very) small) (nice) boy) sees ((a) ((very) nice) girl)
- Some bad sentences:
  - *((the) boy) ((the) cat)
  - *((small) boy) likes ((the) (nice) girl)

From Substrings to Trees

- (((the) boy) likes ((the) cat))

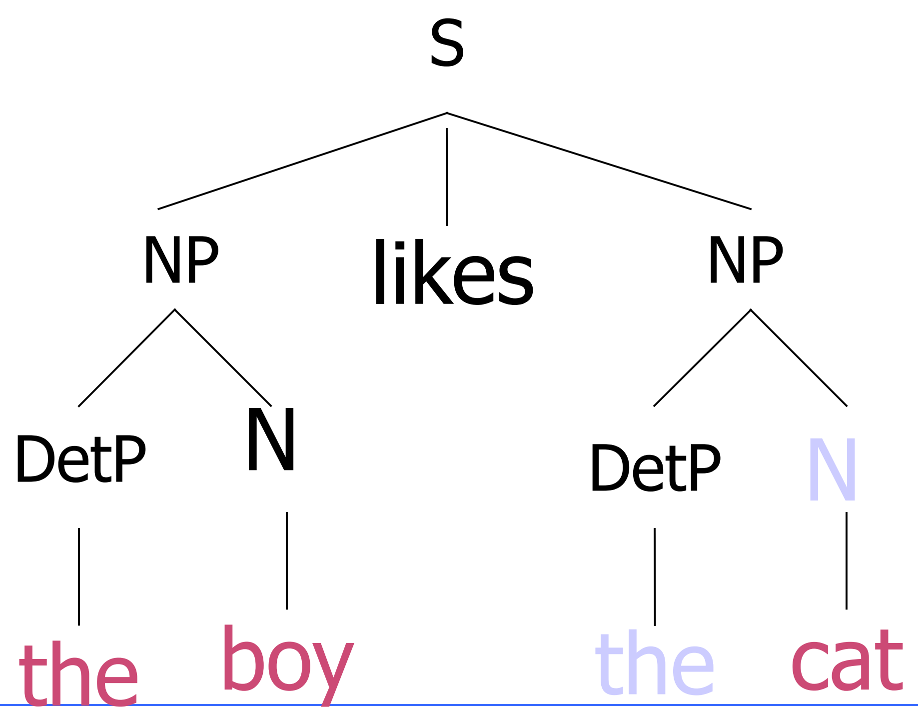


Node Labels?

- ((the) boy) likes ((the) cat)
- Group words by their part-of-speech (POS):
  - Noun (N), verb (V), adjective (Adj), adverb (Adv), determiner (Det)
- Category of constituent: XP, where X is POS
  - NP, AdjP, AdvP, VP, and S

Node Labels

- (((the/Det) boy/N) likes/V ((the/Det) cat/N))



Word Classes = POS

- Possible basic set: N, V, Adj, Adv, Prep, Det, Aux
- 2 supertypes: open- and closed-class
  - Open: N, V, Adj, Adv
  - Closed: Prep, Det, Aux
- Many subtypes:
  - eats/V $\Rightarrow$ eat/VB, eat/VBP, eats/VBZ, ate/VBD, eaten/VBN, eating/VBG,

Can we use prolog to write some simple grammar rules?

- Yes! Prolog comes with a grammar called Definite Clause Grammar (DCG)

Definite Clause Grammars

- A *grammar* is a precise definition of which sequences of words or symbols belong to some *language*.
- Grammars are particularly useful for natural language processing
- But they can be used to process any precisely defined 'language', such as the commands allowed in some human-computer interface.

Grammar rules

- In general, a grammar is defined as a collection of *grammar rules.* These are sometimes called *rewrite rules*, since they show how we can rewrite one thing as something else.
- In linguistics, a typical grammar rule for English might look like this:

  sentence → noun_phrase, verb_phrase

  e.g “ The man ran.”

- This would show that, in English, a *sentence* could be constructed as a *noun phrase*, followed by a *verb phrase*. More example:

  noun_phrase → noun

  noun_phrase → determiner, noun

  verb_phrase → intransitive_verb

  verb_phrase → transitive_verb, noun_phrase

Terminals and non-terminals

- In these rules, symbols like *sentence*, *noun*, *verb*, etc., are used to show the structure of the language
- Such symbols are called *non-terminal symbols*, because they can be further decomposed
- In defining grammar rules for *noun*, we can write:

  noun → [ball]

  noun → [dog]

  noun → [stick]

  These are called the *terminal symbols*, because they can't be expanded any more.

Grammar rules in Prolog

- Prolog allows us to directly implement grammars of this form.
- So, we can write the same rules as:

```
sentence  -->  noun_phrase, verb_phrase.
noun_phrase -->  noun.
noun_phrase -->  determiner, noun.
verb_phrase -->  intransitive_verb.
verb_phrase -->  transitive_verb, noun_phrase.
```

- Here, each non-terminal symbol is like a predicate with no arguments.

Grammar rules in Prolog

- Terminal symbols are represented as lists containing one atom

```
noun --> [ball].
noun --> [dog].
noun --> [stick].
noun --> ['Tripoli'].
```

How Prolog uses grammar rules

- Prolog converts DCG rules into an internal representation which makes them conventional Prolog clauses.
  - This can be seen by 'listing' the consulted code.
- Non-terminals are given two extra arguments, so:

```
sentence --> noun_phrase, verb_phrase.
```

becomes:

```
sentence(In, Out) :-
        noun_phrase(In, Temp),
        verb_phrase(Temp, Out).
```

How Prolog uses grammar rules

- This means: some sequence of symbols In, can be recognised as a sentence, leaving Out as a remainder, if
  - a noun phrase can be found at the start of In, leaving Temp as a remainder,
  - and a verb phrase can be found at the start of Temp, leaving Out as a remainder.

How Prolog uses grammar rules (2)

- Terminal symbols are represented using the special predicate 'C', which has three arguments. So:

```
noun --> [ball].
```

becomes:

```
noun(In, Out) :-
        'C'(In, ball, Out).
```

- This means: some sequence of symbols In can be recognised as a noun, leaving Out as a remainder, if the atom *ball* can be found at the start of that sequence, leaving Out as a remainder.

How Prolog uses grammar rules (2)

- The built-in predicate 'C' is very simply defined:

```
'C'( [Term|List], Term, List ).
```

where it succeeds if its second argument is the head of its first argument, and the third argument is the remainder.

A very simple grammar

- Here's a very simple little grammar, which defines a very small subset of English:

```
sentence --> noun, verb_phrase.
verb_phrase --> verb, noun.
noun --> [ali].
noun --> [salem].
noun --> [apples].
verb --> [likes].
verb --> [hates].
verb --> [runs].
```

A very simple grammar

- We can now use the grammar to test whether some sequence of symbols *belongs to* the language:

```
| ?- sentence([bob, likes, apples], []).
 yes
| ?- sentence([bob, runs], []).
 no
```

A very simple grammar (2)

- By specifying that the remainder is an empty list we can use the grammar to generate all of the possible sentences in the language:

```
| ?- sentence(X, []).
 X = [bob,likes,bob] ? ;
 X = [bob,likes,david] ? ;
 X = [bob,likes,apples] ? ;
 X = [bob,hates,bob] ? ;
 X = [bob,hates,david] ? ;
:
```

Adding Arguments

- We can add our own arguments to the non-terminals in DCG rules to improve our grammar.
- As an example, we can very simply add *number* agreement (singular or plural) between the subject of an English sentence and the main verb.

```
sentence --> noun(Num), verb_phrase(Num).
verb_phrase(Num) --> verb(Num), noun(_).
noun(singular) --> [bob].
noun(plural) --> [students].
verb(singular) --> [likes].
verb(plural) --> [like].
```

Adding Arguments

- So now we can ask prolog:

```
| ?- sentence([bob, likes, students], []).
 yes
| ?- sentence([students, likes, bob], []).
  no
```