Syntactic Analysis based on Constituents

(with CFG grammars)

- **CFG Grammars (Context-free grammars): By convention, the lefthand side of the first production is the start-symbol of the grammar, typically S. All well-formed trees must have this symbol as their root label.**

- **In NLTK context-free grammars are defined in the `nltk.grammar` module.**

- **Example of CFG:**

```
grammar1 = nltk.parse_cfg("""
    S-> NP VP
    VP-> V NP| V NP PP
    PP -> P NP
    V -> "saw" | "ate" |"walked"
    NP -> "John" |"Mary" | "Bob" | Det N
          | Det N PP
    Det -> "a" | "an" | "the" | "my"
```

```
N -> "man" | "dog" | "cat"
|"telescope" | "park"

P -> "in" |"on" | "by" | "with"

""")
```

- **Note that a production like**

  ```
  VP -> V NP | V NP PP
  ```

  has a disjunction on the righthand side, shown by the |, and is an abbreviation for the two productions `VP -> V NP` and `VP -> V NP PP`.

- **A CFG grammar is formed of <u>productions</u> (representing the phrase-structure rules) and the <u>lexicon</u> (identical to the vocabulary).**

- **<u>Parsing a sentence</u> admitted by the grammar (using the recursive descent parser):**

```
>>> sent = "Mary saw Bob".split()

>>>rd_parser=
nltk.RecursiveDescentParser(grammar1)

>>>for tree in rd_parser.nbest_parse(sent):

…  print tree

(S (NP Mary) (VP (V saw) (NP Bob)))
```

- **A grammar is said to be recursive if a category occurring on the lefthand side of a production also appears on the righthand side of the production.**

Parsing with Context-Free Grammars

- **A parser processes input sentences according to the productions of a grammar, and builds one or more constituent structures that conform to the grammar.**

- **A grammar is a declarative specification of well-formedness, while a parser is a procedural interpretation of the grammar.**

- **The parser searches through the space of trees licensed by the grammar in order to find one that has the required sentence along its fringe (frontiera).**

- **Classical parsing algorithms:**
  - **the top-down method called "recursive descent parsing"**
  - **the bottom-up method called "shift-reduce parsing"**

- **More evolved parsing techniques:**
  - **a top-down method with bottom-up filtering called "left-corner parsing"**
  - **a dynamic programming technique called "chart parsing"**

Recursive descent parsing
(top-down)

- **Each production rule gets a procedural interpretation:**

  `S -> NP VP`

  can be interpreted as: "in order to analyze (isolate) an S, first analyze (isolate) an NP, then a VP".

  `N -> mother`

  can be interpreted as: "in order to analyze (isolate) an N (noun), accept the word *mother* from the input string".

- **The resulting process is a *recursive descent* (each production rule is turned into a procedure and the resulting procedures call each other).**

- **The top-down strategy starts with the S symbol and tries to rewrite the symbols until it reaches the input sentence (if this sentence is accepted by the given grammar).**

- **The parser processes the input string by accepting its words on a one-by-one basis.**

Analyzing a constituent of type C

If C represents the type of constituent that the parser is looking for at a given time of the analysis (S, NP, V etc.), then the algorithm for analyzing a constituent of type C is the following:

Algorithm 1

- **If C represents an individual word, it is looked up in the lexical rules of the grammar and it is accepted from the input string.**
- **Otherwise, the phrase-structure rules of the grammar are browsed, in order to rewrite C as a list of constituents. These constituents are then analyzed on a one-by-one basis.**

The State of the Analysis

- **The state of the analysis is a pair consisting of the symbol list and a number indicating the current position within the input string.**

- **The symbol list indicates the result of the processing operations having taken place until the current moment.**

  Example: The parser starts in the state (S). After applying the phrase-structure rule `S -> NP VP`, the symbol list becomes (NP, VP). If immediately after the rule `NP -> ART N` is applied, then the symbol list becomes (ART N VP).

- **Example of the state of the analysis**

  Input string:

  $_1$ ***Colentina*** $_2$ ***angajează*** $_3$ ***surori*** $_4$

  State of the analysis :

  ((N VP)1)

  Significance of this state of the analysis :

  The parser must find an N followed by a VP starting from position 1.

Top-down Parser

- **The parser must systematically explore all possible new states of the analysis (obtained by applying all phrase-structure rules and lexical rules of the grammar).**

- **All possible new states will be generated at each step. One of them will be chosen as representing the next state. The others will be saved as backup states.**

- **If the current state can not lead to a solution, a new current state is chosen from the list of backup states.**

- **The algorithm works with a list of possible states – the list of possibilities. The first element of this list is the current state, representing the state of the analysis. The other elements of the list represent the backup states.**

- **This type of syntactic analysis based on constituents can be viewed as a search problem. In the case of a depth-first strategy, the list of possibilities is organized as a stack. In the case of a breadth-first strategy, the list of possibilities is organized as a queue.**

EXAMPLE:

Consider the following context-free grammar (CFG):

$$S \rightarrow NP\ VP$$

$$NP \rightarrow ART\ \ N$$
$$NP \rightarrow ART\ \ ADJ\ \ N$$
$$VP \rightarrow V$$
$$VP \rightarrow V\ \ NP$$

together with the lexicon

latră: V

câine: N

un: ART

and the input string

$_1$ ***Un*** $_2$ ***câine*** $_3$ ***latră*** $_4$

We perform top-down parsing in this case, implementing a search strategy of type depth-first. The steps of the analysis are the following :

$_1$ *Un* $_2$ *câine* $_3$ *latră* $_4$

Step	Current state	Backup states	Comments
1.	**((S)1)**	**-**	**Initial position**
2.	**((NP VP)1)**	**-**	**Rewrite S with rule 1**
3.	**((ART N VP)1)**	**((ART ADJ N VP)1)**	**Rewrite NP with rules 2 and 3**
4.	**((N VP)2)**	**((ART ADJ N VP)1)**	**ART is reduced with "Un"**
5.	**((VP)3)**	**((ART ADJ N VP)1)**	**N is reduced with "câine"**
6.	**((V)3)**	**((V NP)3) ((ART ADJ N VP)1)**	**Rewrite VP with rules 4 şi 5**
7.	**(()4)**	**((V NP)3) ((ART ADJ N VP)1)**	**V is reduced with "latră"**
8.	**STOP**	**-**	**Correct sentence**

Algorithm 2 (top-down parsing)

Se pleacă din starea iniţială ((S)1), fără stări backup. Paşii algoritmului sunt:

1. *Selectează* starea curentă: se scoate prima stare din lista de posibilităţi; fie ea starea C. Dacă lista de posibilităţi este vidă, atunci algoritmul eşuează (i.e. nu se poate face cu succes analiza sintactică, în sensul că secvenţa de intrare este respinsă ca nefiind conformă cu gramatica limbajului) şi STOP.

2. Dacă C are o listă de simboluri vidă, iar contorul indică poziţia de la sfârşitul propoziţiei, atunci algoritmul are succes (i.e. propoziţia este acceptată ca fiind corectă) şi STOP.

3. Altfel, *generează* următoarele stări posibile. (Adăugarea acestor stări în lista posibilităţilor se va face în funcţie de modul de organizare al listei, adică de modul în care se efectuează căutarea).

3.1. Dacă primul simbol din lista de simboluri a lui C este unul lexical şi următorul cuvânt al propoziţiei

aparţine acelei categorii lexicale, atunci se creează o nouă stare prin înlăturarea primului simbol din lista de simboluri şi repoziţionarea contorului de poziţie. Starea nou creată se adaugă listei posibilităţilor.

3.2. Altfel (i.e. primul simbol din lista de simboluri a lui C este un neterminal), se generează o nouă stare corespunzător fiecărei reguli a gramaticii care poate rescrie acest neterminal. Se adaugă listei posibilităţilor noile stări generate.

The Need for More Parsing Techniques

Problem: When performed from left to right, top-down parsing encounters trouble with rules that exhibit *left recursion*.

Example:

`VP -> VP NP` (rule for a complex verb phrase)

This rule is left recursive in that the first symbol on the right hand side (RHS) is the same as the left hand side (LHS).

With left-recursive rules in a grammar, a left-to-right, top-down parser will get into an infinite loop if it makes the wrong choices. Additionally, if we are asking for all possible parses, it will have to try all possible choices – and will in the end get into an infinite loop.

BOTTOM-UP PARSING (SHIFT-REDUCE PARSER)

Parser cu deplasare-reducere

- **Bottom-up parsers try to find (within the input string) sequences of words and phrases that correspond to the righthand side of a grammar production and replace them with the lefthand side, until the whole sentence is reduced to an S.**

- **A simple kind of bottom-up parser is the shift-reduce parser.**

- **Action: the shift-reduce parser repeatedly pushes the next input word onto a stack – this is the *shift* operation. If the top *n* items on the stack match the *n* items on the righthand side of some production (phrase-structure rule of the grammar), then they are all popped off the stack, and the item on the lefthand side of the production is pushed onto the stack. This replacement of the top *n* items with a single item is the *reduce* operation.**

- **The *reduce* operation may be applied only to the top of the stack; reducing items lower in the stack must be done before later items are pushed onto the stack.**

- **<u>End of parse</u>: the parser finishes when all the input is consumed and there is only one item remaining on the stack, a parse tree with an S node as its root or just the S node.**

Example of shift-reduce parsing

Un câine latră.

Pas	Acţiune	Stivă	Şir de intrare
	START		*un câine latră*
1	**Shift**	*un*	*câine latră*
2	**Reduce**	*ART*	*câine latră*
3	**Shift**	*ART câine*	*latră*
4	**Reduce**	*ART N*	*latră*
5	**Reduce**	*NP*	*latră*
6	**Shift**	*NP latră*	-
7	**Reduce**	*NP V*	-
8	**Reduce**	*NP VP*	-
9	**Reduce**	*S*	-

- **There are two kinds of <u>choices</u> to be made by the parser:**
- **which reduction to do when more than one is possible**
- **whether to shift or to reduce when either action is possible**

- **A shift-reduce parser can be extended to implement policies for resolving such conflicts. For example:**

- **Shift-reduce conflicts - shifts only when no reductions are possible.**
- **Reduce-reduce conflicts - favors the reduction operation that removes the most items from the stack.**

<u>Advantages</u> of shift-reduce parsers over recursive descent parsers:

- **they only build structure that corresponds to the words in the input;**
- **they only build each substructure once.**

Advantages and Disadvantages

- **Shift-Reduce parsing can deal with productions displaying left recursion, unlike Recursive descent parsing. This is so because its actions are determined solely by words actually occurring in the input sentence.**

- **Shift-Reduce parsing suffers from another limitation: it is not able to deal with productions of type**

$$A \longrightarrow \phi$$

since it has no way of reacting when faced with a null constituent (a constituent that is missing).

Example for Romanian:

NP -> Det N

Det -> ϕ

corresponding to

"*un baiat*" and "*baiatul*", respectively.

- **A *better solution*:**

the left-corner parser – representing a top-down parser with bottom-up filtering

Shift-Reduce Parsing with NLTK

- **NLTK provides**

  `ShiftReduceParser()`

  which is a simple implementation of a shift-reduce parser.

- **This parser does not implement any backtracking, so it is not guaranteed to find a parse for a text, even if one exists. Furthermore, it will only find at most one parse, even if more parses exist.**

- **Recommended programming language for shift-reduce parsing: PROLOG – which incorporates the backtracking process.**