Natural Language Processing

Unit 1: Syntactic analysis

January 2020

Syntactic analysis

Definition

The Syntactic analysis in NLP deals with the correct formation of the text. Usually deals with the concept of parsing study at the level of how sentences are formed with respect to a grammar and it's structure. This level of linguistic processing utilizes a parsers to extract the structure fo the phrases.

Why is important?

- Syntactic analysis let us discover the parts of the sentences and the categories they have to build a meaning

Why is important?

- Syntactic analysis let us discover the parts of the sentences and the categories they have to build a meaning
- Some subsets of the natual language have regular and contet-free structures that can be analyzed using syntactic parsing

Why is important?

- Syntactic analysis let us discover the parts of the sentences and the categories they have to build a meaning
- Some subsets of the natual language have regular and contet-free structures that can be analyzed using syntactic parsing
- Let the designers to detect malformed and potentially harmful information

Why is important?

- Syntactic analysis let us discover the parts of the sentences and the categories they have to build a meaning
- Some subsets of the natual language have regular and contet-free structures that can be analyzed using syntactic parsing
- Let the designers to detect malformed and potentially harmful information
- A good syntactic analysis simplifies the task to assign a meaning of a sentence

Context free grammars

Concept

Context-free grammars are mathematical tools which define the rules and elements of a context-free language. They offer a way to undarstand the key elements of a sentence and how they are related.

Conext free grammars

Why are they useful?:

Conext free grammars

Why are they useful?:

- Were used in the study of human languagesto understand the relationship of terms such as *noun*, *verb*, and *preposition* among others.
- They can represeent complex structures which lead to natural recursion because noun phrases may appear inside verb phrases and vice versa

Context free grammars

A grammar consists of a collection of substitution rules, also called **productions**

- Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow.
- The symbol is called a variable. The string consists of variables and other symbols called **terminals**.
- The variable symbols often are represented by capital letters.
- The terminals are analogous to the input alphabet and often represented by lowercase letters, numbers, or special symbols.
- One variable is designated as the start variable. It usually occurs on the left-hand side of the topmost rule.

Context free grammars

How powerful they are?

Context free grammars

How powerful they are?

Build a regular expression to represent the language 0^n1^n

Context free grammars

How powerful they are?

Build a regular expression to represent the language 0^n1^n

It's not possible!

Context free grammars

How powerful they are?

Build a regular expression to represent the language 0^n1^n

It's not possible!

You nead a free-context grammar to represent that language:

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Context free grammar example

Consider the grammar G_1:
(1) $A \rightarrow 0A1$
(2) $A \rightarrow B$
(3) $B \rightarrow \#$

- Has three rules: (1),(2),(3)
- Has two variables: A, B
- A is the start variable
- Has three terminals: 0,1,#

We can check if an element belongs to a language if we can generate that element starting wih the start variable and applying a sequence of substitutions defined by the grammar rules. This is called a **derivation**.

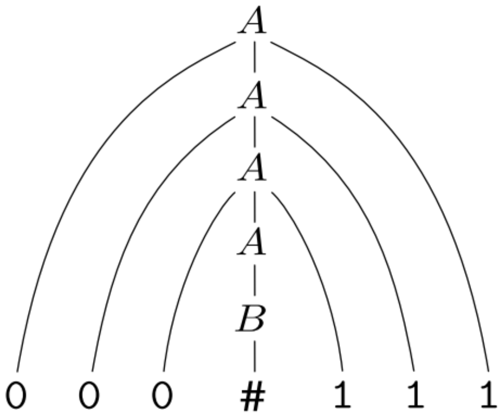
Context free grammar example

As an example we can use G_1:
(1) $A \rightarrow 0A1$
(2) $A \rightarrow B$
(3) $B \rightarrow \#$
to generate the string 000#111 using the following derivation:

$A \Longrightarrow 0A1$
$0A1 \Longrightarrow 00A11$
$00A11 \Longrightarrow 000A111$
$000A111 \Longrightarrow 000B111$
$000B111 \Longrightarrow 000\#111$

Parse Tree

It also produces a parse tree:



Free context languages

What kind of languages can be recognized using a free-context grammar?

- The collection of languages represented by context-free grammars are also called context-free languages.
- Context-free languages include all the regular languages and many additional languages.
- Context free grammars can be recognized using pushdown automata, a class of machines which give us to gain additional insight into the power of context-free grammars.

Context free grammar

Definition

A context-free grammar is a 4-tuple (V, Σ, R, S), where:

- V is a finite set called the variables,
- Σ is a finite set, disjoint from V, called the terminals,
- R is a finite setof rules, with each rule being a variable and a string of variables and terminals, and
- $S \in V$ is the start variable.

Example

Consider the following grammar G_1:

(1) $(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$
(2) $(NOUN\_PHR) \rightarrow (CMP\_NOUN) \mid (CMP\_NOUN)(PREP\_PHR)$
(3) $(VERB\_PHR) \rightarrow (CMP\_VERB) \mid (CMP\_VERB)(PREP\_PHR)$
(4) $(PREP\_PHR) \rightarrow (PREP)(CMP\_NOUN)$
(5) $(CMP\_NOUN) \rightarrow (ARTICLE)(NOUN)$
(6) $(CMP\_VERB) \rightarrow (VERB) \mid (VERB)(NOUN\_PHR)$
(7) $(ARTICLE) \rightarrow a \mid the$
(8) $(NOUN) \rightarrow boy \mid girl \mid flower$
(9) $(VERB) \rightarrow touches \mid likes \mid sees$
(10) $(PREP) \rightarrow with$

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)
$(NOUN\_PHR)(VERB\_PHR) \rightarrow (CMP\_NOUN)(VERB\_PHR)$ (2)

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)
$(NOUN\_PHR)(VERB\_PHR) \rightarrow (CMP\_NOUN)(VERB\_PHR)$ (2)
$(CMP\_NOUN)(VERB\_PHR) \rightarrow$
$(ARTICLE)(NOUN)(VERB\_PHR)$ (5)

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)
$(NOUN\_PHR)(VERB\_PHR) \rightarrow (CMP\_NOUN)(VERB\_PHR)$ (2)
$(CMP\_NOUN)(VERB\_PHR) \rightarrow$
$(ARTICLE)(NOUN)(VERB\_PHR)$ (5)
$(ARTICLE)(NOUN)(VERB\_PHR) \rightarrow a\ (NOUN)(VERB\_PHR)$ (7)

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)
$(NOUN\_PHR)(VERB\_PHR) \rightarrow (CMP\_NOUN)(VERB\_PHR)$ (2)
$(CMP\_NOUN)(VERB\_PHR) \rightarrow$
$(ARTICLE)(NOUN)(VERB\_PHR)$ (5)
$(ARTICLE)(NOUN)(VERB\_PHR) \rightarrow a\ (NOUN)(VERB\_PHR)$ (7)
$a\ (NOUN)(VERB\_PHR) \rightarrow a\ girl\ (VERB\_PHR)$ (8)

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)
$(NOUN\_PHR)(VERB\_PHR) \rightarrow (CMP\_NOUN)(VERB\_PHR)$ (2)
$(CMP\_NOUN)(VERB\_PHR) \rightarrow$
$(ARTICLE)(NOUN)(VERB\_PHR)$ (5)
$(ARTICLE)(NOUN)(VERB\_PHR) \rightarrow a\ (NOUN)(VERB\_PHR)$ (7)
$a\ (NOUN)(VERB\_PHR) \rightarrow a\ girl\ (VERB\_PHR)$ (8)
$a\ girl\ (VERB\_PHR) \rightarrow a\ girl\ (CMP\_VERB)$ (3)

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)
$(NOUN\_PHR)(VERB\_PHR) \rightarrow (CMP\_NOUN)(VERB\_PHR)$ (2)
$(CMP\_NOUN)(VERB\_PHR) \rightarrow$
$(ARTICLE)(NOUN)(VERB\_PHR)$ (5)
$(ARTICLE)(NOUN)(VERB\_PHR) \rightarrow a\ (NOUN)(VERB\_PHR)$ (7)
$a\ (NOUN)(VERB\_PHR) \rightarrow a\ girl\ (VERB\_PHR)$ (8)
$a\ girl\ (VERB\_PHR) \rightarrow a\ girl\ (CMP\_VERB)$ (3)
$a\ girl\ (CMP\_VERB) \rightarrow a\ girl\ (VERB)$ (6)

Example

Using the grammar G_1 we can perform the following derivations:

$(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$ (1)
$(NOUN\_PHR)(VERB\_PHR) \rightarrow (CMP\_NOUN)(VERB\_PHR)$ (2)
$(CMP\_NOUN)(VERB\_PHR) \rightarrow$
$(ARTICLE)(NOUN)(VERB\_PHR)$ (5)
$(ARTICLE)(NOUN)(VERB\_PHR) \rightarrow a\ (NOUN)(VERB\_PHR)$ (7)
$a\ (NOUN)(VERB\_PHR) \rightarrow a\ girl\ (VERB\_PHR)$ (8)
$a\ girl\ (VERB\_PHR) \rightarrow a\ girl\ (CMP\_VERB)$ (3)
$a\ girl\ (CMP\_VERB) \rightarrow a\ girl\ (VERB)$ (6)
$a\ girl\ (VERB) \rightarrow a\ girl\ sees$ (9)

Ambiguity

Sometimes a grammar can generate the same string in several different ways. Such a string will have several different parse trees and thus several different meanings. This result may be undesirable for certain applications, such as programming languages, where a program should have a unique interpretation.

Ambiguity

If a grammar generates the same string in several different ways, we say that the string is derived ambiguously in that grammar. If a grammar generates some string ambiguously, we say that the grammar is ambiguous.

Definition

A string w is derived ambiguously in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

Example of ambiguity

Using the following grammar:

$$EXPR \rightarrow EXPR \; + \; EXPR \mid EXPR \; x \; EXPR \mid (EXPR) \mid a$$

Example of ambiguity

Using the following grammar:

$EXPR \rightarrow EXPR\ +\ EXPR \mid EXPR\ x\ EXPR \mid (EXPR) \mid a$

We can derive the string: a + a x a in two different ways:

Example of ambiguity

Using the following grammar:

$EXPR \rightarrow EXPR \ + \ EXPR \mid EXPR \ x \ EXPR \mid (EXPR) \mid a$

We can derive the string: a + a x a in two different ways:

First derivation:

$EXPR \Longrightarrow EXPR \ x \ EXPR \Longrightarrow EXPR \ + \ EXPR \ x \ EXPR \Longrightarrow$
$a + EXPR \ x \ EXPR \Longrightarrow a + a \ x \ EXPR \Longrightarrow a + a \ x \ a$

Example of ambiguity

Using the following grammar:

$EXPR \rightarrow EXPR\ +\ EXPR \mid EXPR\ x\ EXPR \mid (EXPR) \mid a$

We can derive the string: a + a x a in two different ways:

First derivation:

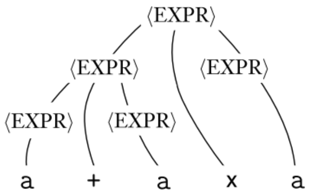
$EXPR \Longrightarrow EXPR\ x\ EXPR \Longrightarrow EXPR\ +\ EXPR\ x\ EXPR \Longrightarrow$
$a + EXPR\ x\ EXPR \Longrightarrow a + a\ x\ EXPR \Longrightarrow a + a\ x\ a$

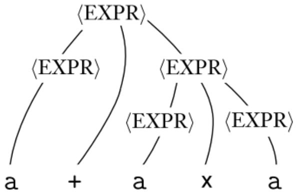
Second derivation:

$EXPR \Longrightarrow EXPR + EXPR \Longrightarrow a + EXPR \Longrightarrow$
$a + EXPR\ x\ EXPR \Longrightarrow a + a\ x\ EXPR \Longrightarrow a + a\ x\ a$

Example of ambiguity

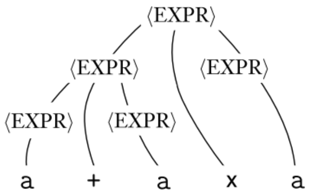
Here we generate two parse tree for the derivation:

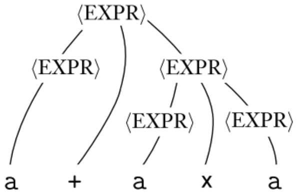




Example of ambiguity

Here we generate two parse tree for the derivation:

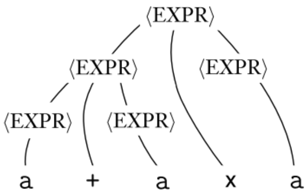




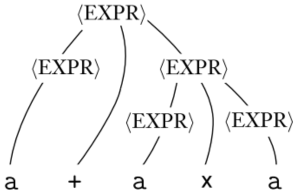
If we assign a value for a, let's say 3 and perform the opreations in the generated parse trees what result we will have?

Example of ambiguity

Here we generate two parse tree for the derivation:



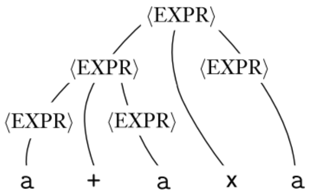
$(3+3)x3 = 18$



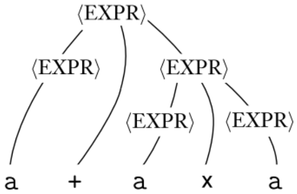
$3+(3x3) = 12$

Example of ambiguity

Here we generate two parse tree for the derivation:



$(3+3)x3 = 18$



$3+(3x3) = 12$

The expression has two different meanings!!

Pushdown automata

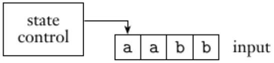
These automata are like nondeterministic finite automata but have an extra component called **stack**. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

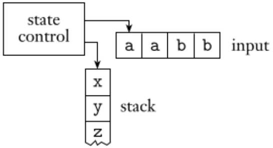
Pushdown automata

Pushdown automata are equivalent in power to **context-free grammars**. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it.

Pushdown automata

Schemas of a finite automata vs. pushdown automata





Pushdown automata

Definition

A pushdown automata is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where:

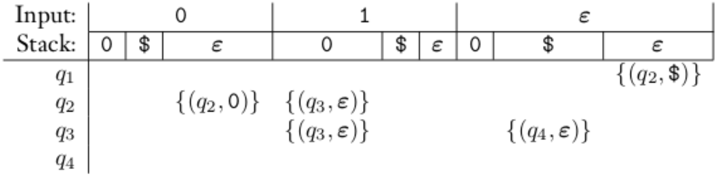
- Q is the set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $\delta : Qx\Sigma_\epsilon x\Gamma_\epsilon \rightarrow P(Qx\Gamma_\epsilon)$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accept states.

Pushdown automata Example

PDA that recognizes the language $\{0^n1^n | n \geq 0\}$.
Let M_1 be $(Q, \Sigma, \Gamma, \delta, q_1, F)$, where:

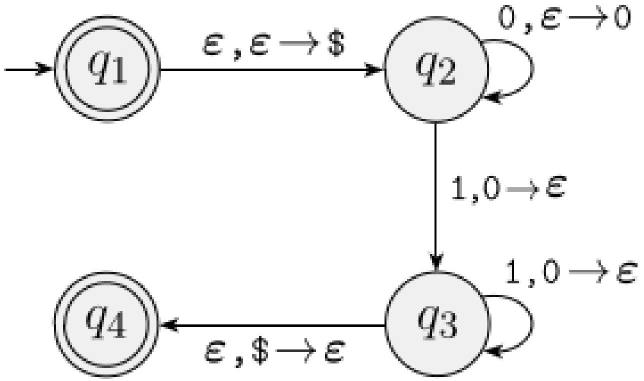
- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, \$\}$
- $F = \{q_1, q_4\}$

Pushdown automata Example



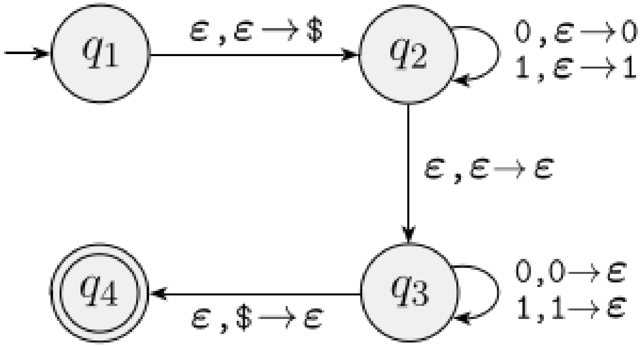
Input:	0			1			ε		
Stack:	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

Pushdown automata Example


q_1
$\varepsilon, \varepsilon \rightarrow \$$
q_2
$0, \varepsilon \rightarrow 0$
$1, 0 \rightarrow \varepsilon$
q_3
$1, 0 \rightarrow \varepsilon$
q_4
$\varepsilon, \$ \rightarrow \varepsilon$

Design a pushdown automata which recognize the language: $\{ww^R|w \in \{0,1\}^*\}$ where w^R means w written backwards:

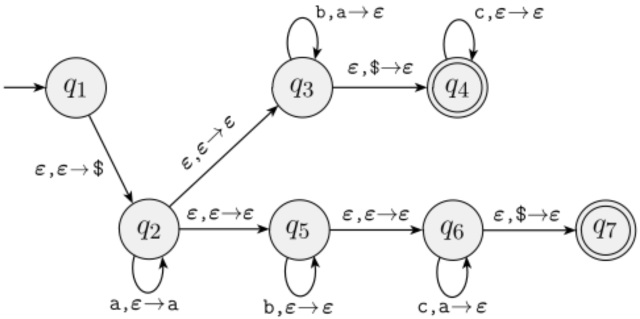
Design a pushdown automata which recognize the language: $\{ww^R | w \in \{0,1\}^*\}$ where w^R means w written backwards:



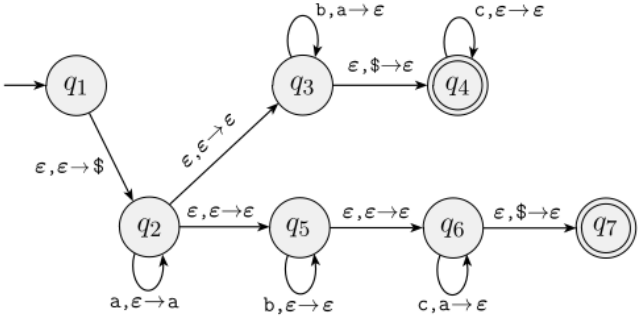
q_1
$\varepsilon, \varepsilon \rightarrow \$$
q_2
$0, \varepsilon \rightarrow 0$
$1, \varepsilon \rightarrow 1$
$\varepsilon, \varepsilon \rightarrow \varepsilon$
q_3
$0, 0 \rightarrow \varepsilon$
$1, 1 \rightarrow \varepsilon$
$\varepsilon, \$ \rightarrow \varepsilon$
q_4

Design a pushdown automata which recognize the language:
$\{a^i b^j c^k | i, j, k \geq 0$ and $i = j$ or $i = k$

Design a pushdown automata which recognize the language:
$\{a^i b^j c^k | i, j, k \geq 0 \text{ and } i = j \text{ or } i = k$



Design a pushdown automata which recognize the language: $\{a^i b^j c^k | i, j, k \geq 0$ and $i = j$ or $i = k$



How would you modify the automata in order to accept the language $\{a^i b^j c^k | i, j, k \geq 0$ and $i = j$ or $j = k$?

Chomsky normal form

Definition

A context-free grammar is in Chomsky normal form if every rule is of the form:

$A \rightarrow BC$

$A \rightarrow a$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \rightarrow \epsilon$, where S is the start variable.

Chomsky normal form

Definition

A context-free grammar is in Chomsky normal form if every rule is of the form:

$A \rightarrow BC$

$A \rightarrow a$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \rightarrow \epsilon$, where S is the start variable.

ANY context-free language is generated by a **context-free grammar in Chomsky normal form** (That's a theorem)

Chomsky normal form conversion

We can convert any grammar G into Chomsky normal form:

- Add a new start variable
- Eliminate $\epsilon - \mathit{rules}$ of the form $A \rightarrow \epsilon$
- Eliminate all unit rules of the form $A \rightarrow B$
- Convert the remaining rules to the proper form

Chomsky Normal Form Conversion Example

1.- Add a new start variable:

$$\begin{aligned} S &\to ASA \mid \mathtt{a}B \\ A &\to B \mid S \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

$$\begin{aligned} \boldsymbol{S_0} &\to \boldsymbol{S} \\ S &\to ASA \mid \mathtt{a}B \\ A &\to B \mid S \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

2.- Eliminate $\epsilon -$ *rules* of the form $A \to \epsilon$

$$\begin{aligned} S_0 &\to S \\ S &\to ASA \mid \mathtt{a}B \mid \boldsymbol{\mathtt{a}} \\ A &\to B \mid S \mid \boldsymbol{\varepsilon} \\ B &\to \mathtt{b} \mid \varepsilon \end{aligned}$$

$$\begin{aligned} S_0 &\to S \\ S &\to ASA \mid \mathtt{a}B \mid \mathtt{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \mid \boldsymbol{S} \\ A &\to B \mid S \mid \varepsilon \\ B &\to \mathtt{b} \end{aligned}$$

Chomsky Normal Form Conversion Example

3.- Eliminate all unit rules of the form $A \rightarrow B$

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \mid S \\ A &\rightarrow B \mid S \\ B &\rightarrow \mathtt{b} \end{aligned}$$

$$\begin{aligned} S_0 &\rightarrow S \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\ S &\rightarrow ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\ A &\rightarrow B \mid S \\ B &\rightarrow \mathtt{b} \end{aligned}$$

Chomsky Normal Form Conversion Example

4.- Transform the remaining rules to the form $A \rightarrow BC$

$$\begin{aligned}
S_0 &\rightarrow ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow B \mid S \mid \mathbf{b} \\
B &\rightarrow \mathtt{b}
\end{aligned}
\qquad
\begin{aligned}
S_0 &\rightarrow ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow ASA \mid \mathtt{a}B \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow S \mid \mathtt{b} \mid \boldsymbol{ASA} \mid \mathbf{a}\boldsymbol{B} \mid \mathbf{a} \mid \boldsymbol{SA} \mid \boldsymbol{AS} \\
B &\rightarrow \mathtt{b}
\end{aligned}$$

Chomsky Normal Form Conversion Example

4.- Transform the remaining rules to the form $A \rightarrow BC$

$$
\begin{aligned}
S_0 &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
S &\rightarrow AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A &\rightarrow \mathtt{b} \mid AA_1 \mid UB \mid \mathtt{a} \mid SA \mid AS \\
A_1 &\rightarrow SA \\
U &\rightarrow \mathtt{a} \\
B &\rightarrow \mathtt{b}
\end{aligned}
$$

Parser

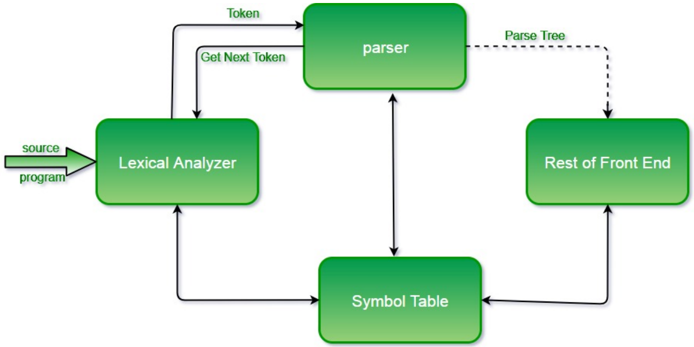
A natural language parser is a program that works out the grammatical structure of sentences, for instance, which groups of words go together (as "phrases") and which words are the subject or object of a verb. Probabilistic parsers use knowledge of language gained from hand-parsed sentences to try to produce the most likely analysis of new sentences.

Parser

Parser

It is used to implement the task of parsing. It may be defined as the software component designed for taking input data (text) and giving structural representation of the input after checking for correct syntax as per formal grammar. It also builds a data structure generally in the form of parse tree or abstract syntax tree or other hierarchical structure.

Parser

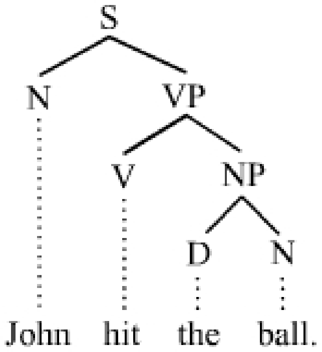


Retrieved from https://www.geeksforgeeks.org/parsing-set-1-introduction-ambiguity-and-parsers/

A parse tree or parsing tree or derivation tree or concrete syntax tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context free grammar. The term parse tree itself is used primarily in computational linguistics; in theoretical syntax, the term syntax tree is more common.

Given the following grammar:
$S \rightarrow N\ VP$
$VP \rightarrow V\ NP$
$NP \rightarrow D\ N$
$V \rightarrow hit \mid sees$
$D \rightarrow the \mid a$
$N \rightarrow John \mid ball$

Parse tree example



Constituency-based parse tree

Parsing methods

- Top Down Parsing: The parser starts constructs the parse tree starting from the start symbol and then tries to transform it to the input. The most common form of topdown parsing uses recursive procedure to process the input. The main disadvantage of recursive descent parsing is backtracking.
- Bottom Up parsing: In this kind of parsing, the parser starts with the input symbol, i.e terminal symbols and tries to construct the parse tree up to the start symbol using the grammar rules.

Recursive descent parser

Interprets a grammar as a specification of how to break a high-level goal into several lower-level subgoals. The top-level goal is to find an S. The rules of the form: $S \rightarrow X\ Y$ permits the parser to replace this goal with subgoals: finding the symbols produced by S. Each of these subgoals can be replaced in turn by sub-sub-goals, using productions of the intermediate symbols until we can derive the input sentence.

Recursive descent parser shortcomings

- Left-recursive productions like $NP \rightarrow NP\ PP$ send it into an infinite loop
- Waste a lot of time considering words and structures that do not correspond to the input sentence
- The backtracking process may discard parsed constituents that will need to be rebuilt again later

Shift reduce parser

Bottom-up parser tries to find sequences of words and phrases that correspond to the right hand side of a grammar production, and replace them with the left-hand side, until the whole sentence is reduced to an S.

Shift reduce parser

- Repeatedly pushes the next input word onto a stack; this is called the shift operation
- If the top n items on the stack match the n items on the right hand side of some production, then they are all popped off the stack, and the item on the left-hand side of the production is pushed on the stack. This replacement of the top n items with a single item is the reduce operation
- This operations may only be applied to the top of the stack.

Shift reduce parser

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. The shift-reduce parser builds a parse tree during the above process. Each time it pops n items off the stack it combines them into a partial parse tree, and pushes this back on the stack.

CYK algorithm

Cocke–Younger–Kasami algorithm (alternatively called CYK, or CKY) is a parsing algorithm for context-free grammars, named after its inventors, John Cocke, Daniel Younger and Tadao Kasami. It employs bottom-up parsing and dynamic programming. The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However any context-free grammar may be transformed to a CNF grammar expressing the same language.

CYK algorithm

The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Big O notation, the worst case running time of CYK is $O(n^3.|G|)$, where n is the length of the parsed string and $|G|$ is the size of the CNF grammar G. This makes it one of the most efficient parsing algorithms in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios.

CYK example

Given the following grammar G:

$S \rightarrow AB \mid BB$

$A \rightarrow CC \mid AB \mid a$

$B \rightarrow BB \mid CA \mid b$

$C \rightarrow BA \mid AA \mid b$

Determine if the string aabb can belongs tot he language expressed by G:

CYK example

Look for all the variables that can derive every single terminal for the first row:

a **A**	a **A**	b **B,C**	b **B,C**
$t_{2,1}$	$t_{2,2}$	$t_{2,3}$	
$t_{3,1}$	$t_{3,2}$		
$t_{4,1}$			

CYK example

Look for all the two combination variables that can derived in this case aa, ab and bb. aa can be derived by AA, ab can be derived from AB or AC and bb can be derived from BB, BC, CB and CC. Find the rules that produce those and see possible derivations:

a **A**	a **A**	b **B,C**	b **B,C**
aa **C**	ab **S, A**	bb **S, B, A**	
$t_{3,1}$	$t_{3,2}$		
$t_{4,1}$			

CYK example

Look for all the combination variables that can derive three letters. To form the string aab we can use a ab or aa b. a A and ab can be derived by A thus a ab only can be derived by AA which is produced by C. An anologous process is done for aa b and for the sequences a bb and ab b:

a **A**	a **A**	b **B,C**	b **B,C**
aa **C**	ab **S, A**	bb **S, B, A**	
aab **C, A**	abb **S, A, C**		
$t_{4,1}$			

CYK example

Look for the combinations to form the 4 terminal string aabb. We can form aabb using a aab, aa bb, aab b. a can be produceb by A and aab can be produced by A,S. The same process is done for the rest of the cases:

a **A**	a **A**	b **B,C**	b **B,C**
aa **C**	ab **S, A**	bb **S, B, A**	
aab **C, A**	abb **S, A, C**		
aabb **C,B,A,S**			

Since the start variable can derive the entire input the sentence is accepted in the language.

CYK algorithm example 2

$S \rightarrow NP\ VP$
$NP \rightarrow A\ B$
$VP \rightarrow C\ NP$
$A \rightarrow det$
$B \rightarrow n$
$NP \rightarrow n$
$VP \rightarrow vi$
$C \rightarrow vt$
Parse the sentence: “the cat eats fish”
the (det) cat(n) eats(vt,vi) fish(n)

CYK algorithm example 2

the cat eats fish **S**			
the cat eats **S**	cat eats fish **S**		
the cat **NP**	cat eats **S**	eats fish **VP**	
the (det) **A**	cat (n) **B, NP**	eats (vt,vi) **C, VP**	fish (n) **B, NP**

CYK algorithm

```
**function** CKY-PARSE(*words, grammar*) **returns** *table*

  **for** $j \leftarrow$ **from** 1 **to** LENGTH(*words*) **do**
      **for all** $\{A \mid A \rightarrow words[j] \in grammar\}$
            $table[j-1, j] \leftarrow table[j-1, j] \cup A$
      **for** $i \leftarrow$ **from** $j-2$ **downto** 0 **do**
          **for** $k \leftarrow i+1$ **to** $j-1$ **do**
              **for all** $\{A \mid A \rightarrow BC \in grammar$ **and** $B \in table[i,k]$ **and** $C \in table[k,j]\}$
                    $table[i,j] \leftarrow table[i,j] \cup A$
```

Example

Convert G_1 to CNF and check if the string: **“the girl sees”** is accepted:

(1) $(SNTC) \rightarrow (NOUN\_PHR)(VERB\_PHR)$
(2) $(NOUN\_PHR) \rightarrow (CMP\_NOUN) \mid (CMP\_NOUN)(PREP\_PHR)$
(3) $(VERB\_PHR) \rightarrow (CMP\_VERB) \mid (CMP\_VERB)(PREP\_PHR)$
(4) $(PREP\_PHR) \rightarrow (PREP)(CMP\_NOUN)$
(5) $(CMP\_NOUN) \rightarrow (ARTICLE)(NOUN)$
(6) $(CMP\_VERB) \rightarrow (VERB) \mid (VERB)(NOUN\_PHR)$
(7) $(ARTICLE) \rightarrow a \mid the$
(8) $(NOUN) \rightarrow boy \mid girl \mid flower$
(9) $(VERB) \rightarrow touches \mid likes \mid sees$
(10) $(PREP) \rightarrow with$

Example

G_1 in CNF:

(1) $S \rightarrow (NP)(VP)$
(2) $NP \rightarrow (A)(N) \mid (CN)(PP)$
(3) $VP \rightarrow touches \mid likes \mid sees \mid (V)(NP) \mid (CV)(PP)$
(4) $PP \rightarrow (P)(CN)$
(5) $CN \rightarrow (A)(N)$
(6) $CV \rightarrow touches \mid likes \mid sees \mid (V)(NP)$
(7) $A \rightarrow a \mid the$
(8) $N \rightarrow boy \mid girl \mid flower$
(9) $V \rightarrow touches \mid likes \mid sees$
(10) $P \rightarrow with$

CYK example

Since the start variable can derive the entire input the sentence **“the girl sees”** is accepted in the language.

the **A**	girl **N**	sees **V, CV, VP**
the girl **CN, NP**	girl sees -	
the girl sees **S**		

Let’s code

Assignments

Assignment 4: Implement a simple CYK algorithm

- Implement a class which recieve the rules of a grammar as a list of tuples, and the string to parse, and outputs the parsing table of CYK algorithm
- The class must perform a pre processing using the methods developed before
- The class must perform POS tagging to the input string and the tags will be the teminal symbols of the grammar.
- Include a jupyter notebook explaining the functionality of the class.
- Add an example of the processing using the simple twiter database provided.

References

[1] Jurafsky D., Martin J.: Speech and Language Processing 2nd. ed. (2009).

[2] Bird S., Klein E., Loper, E.: Natural Language Processing with Python, (2009). ISBN: 978-0-596-51649-9

[3] Indurkhya N., Damerau F. Handbook of Natural Language Processing, Second Edition (2010). ISBN: 978-1-4200-8593-8

[4] Kao, A., Poteet S. Natural Language Processing and Text Mining (2007). ISBN: 78-1-84628-175-4

[5] Sipser, M. Introduction to the theory of computation (2013). ISBN: 978-1-133-18779-0

[6] https://www.geeksforgeeks.org/parsing-set-1-introduction-ambiguity-and-parsers/