PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE

FACULTAD DE LETRAS

Introducción a la lingüística computacional

César Antonio Aguilar

Facultad de Lenguas y Letras

10/10/2017

Cesar.Aguilar72@gmail.com

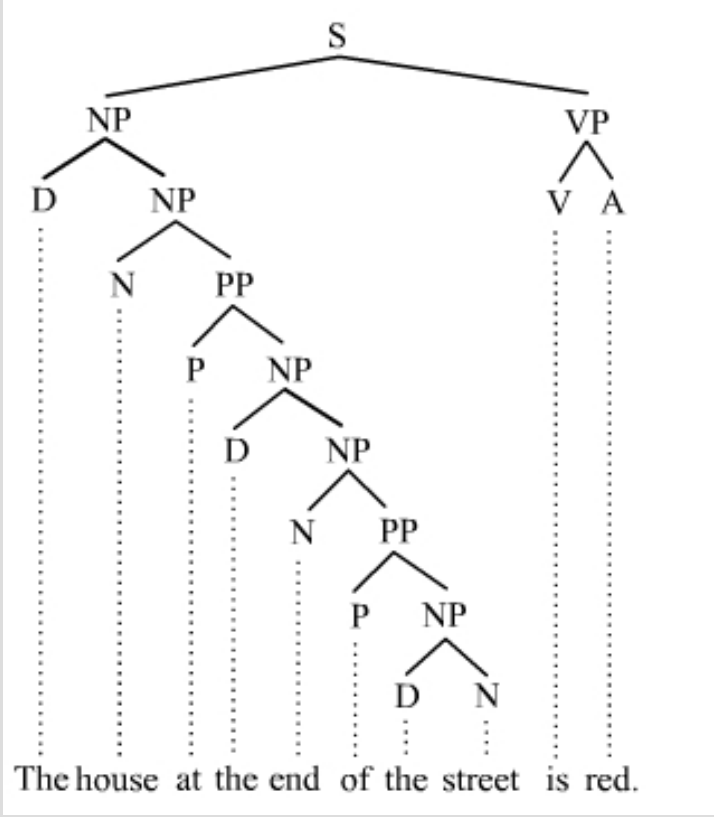
Analizando frases con NLTK (1)

En esta sesión vamos a empezar a emplear algunas de las funciones que ofrece NLTK para el análisis sintáctico. Antes de esto, recordemos algunas cuestiones vistas la clase pasada.

Primero, empecemos por definir lo que es una frase:

Conjunto de palabras que conforman secuencias específicas configuradas en torno a una serie de reglas.

Dicha configuración puede ser esquematizada a través de un árbol sintáctico, p.e.:



Analizando frases con NLTK (2)

Toda palabra dentro de la estructura de una frase cumple únicamente dos funciones: o ser el **núcleo de ésta, o ser un modificador.**

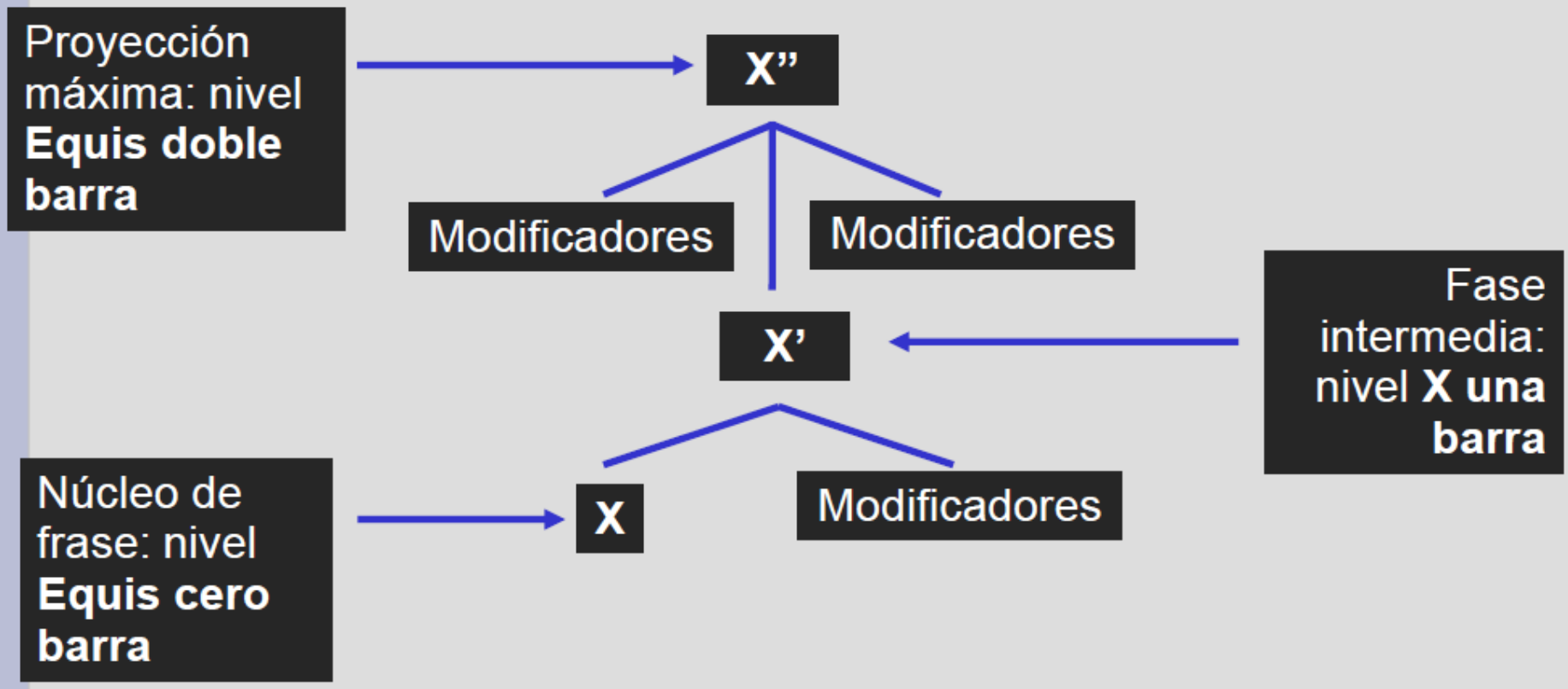
Así, la parte central (esto es, la *cabeza*) de una frase es entonces su núcleo, de modo que éste tiene la propiedad de lograr una **proyección máxima**, dependiendo de cuántos modificadores se le puedan agregar.

Por lo general, este nivel de proyección máxima se logra cubriendo tres fases:

1. Una fase 0, en la que simplemente contamos con un núcleo aislado. At esta fase se le conoce como **nivel X^0** (equis cero barra).
2. Una fase intermedia que configura una estructura conformada por el núcleo más un primer modificador, que puede ser una palabra, una frase, una oración o varias oraciones. A este nivel se le conoce como **nivel X'** (equis una barra).
3. Una fase final en donde la frase se completa en su totalidad, al añadirse un modificador más. Este nivel se le conoce como **nivel X"** (equis doble barra).

Analizando frases con NLTK (3)

El esquema es el siguiente:



Analizando frases con NLTK (4)

Podemos realizar esta clase análisis en NLTK, para lo cual seguiremos el siguiente ejemplo. Primero, redactemos la siguiente oración:

Sentence01 = =[("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]

Analizando frases con NLTK (5)

Posteriormente, requerimos desarrollar una gramática como la siguiente:

```
Grammar01 ="NP: {<DT>?<JJ>*<NN>}”
```

Lo que hemos formulado en nuestra gramática es el patrón que sigue una frase nominal (en inglés, NP), de tal suerte que esta secuencia se compone de un determinante (DT), un adjetivo (JJ) y un nombre en función de núcleo. Una vez definida nuestra gramática, aplicamos la siguiente función:

```
cp = nltk.RegexpParser(Grammar01)
```

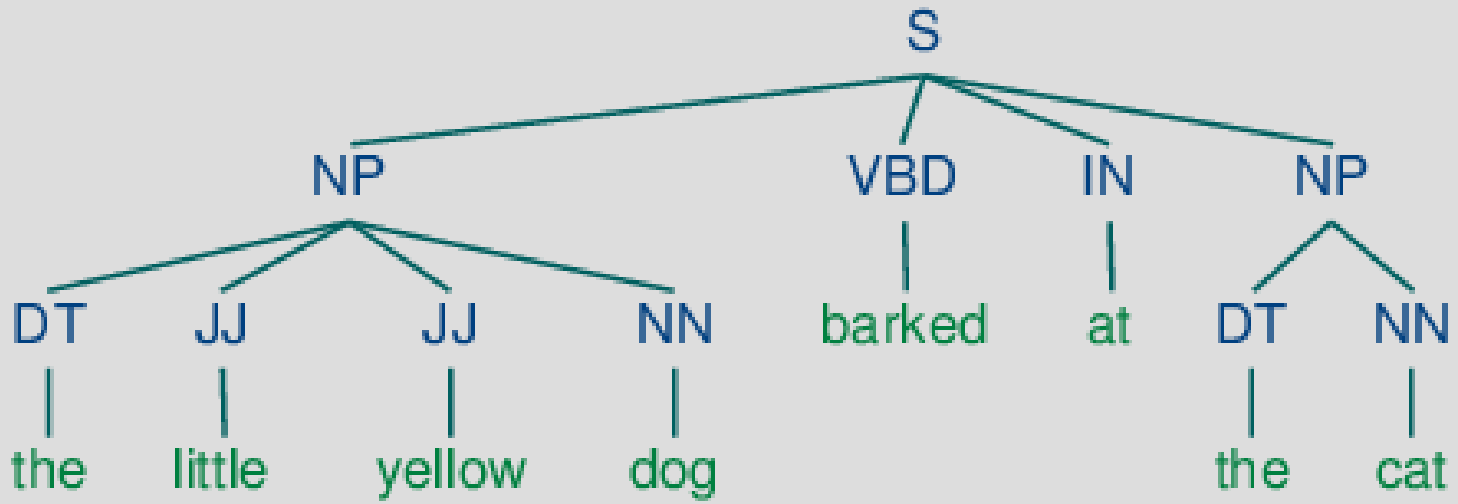
Analizando frases con NLTK (6)

Veamos el resultado:

Result01 = cp.parse(Sent01)

Finalmente, grafiquemos nuestro árbol:

Result01.draw()



Analizando frases con NLTK (7)

Si se dan cuenta, nuestra gramática es una secuencia de etiquetas con expresiones regulares, las cuales simplemente indican aquí:

1. *Busca toda secuencia que inicie con un determinante (o **DT**), más cualquier otra palabra subsecuente*.

2. *A esta cadena se debe ligar un adjetivo (o **JJ**), el cual puede tener una recurrencia de 0 a infinito**.***

3. *Finalmente, a esta secuencia se liga un nombre común (o **NN**)*.

Analizando frases con NLTK (8)

Las reglas y patrones que podamos construir con nuestras gramáticas pueden ser tan simples o complejos como lo creamos conveniente. Veamos el siguiente ejemplo:

```
Sentence02 = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"), ("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair", "NN")]
```

Nuestra gramática va a ser la siguiente:

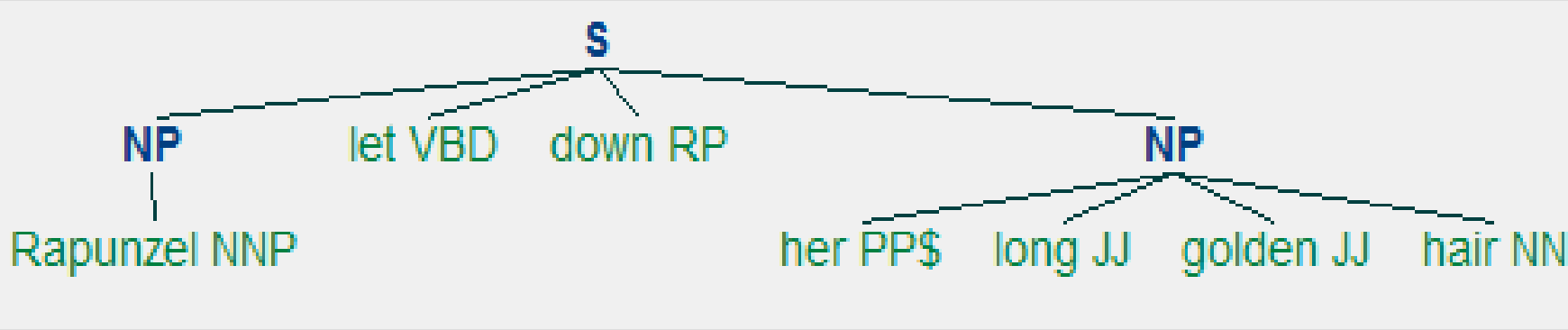
```
Grammar02 = r""""
NP: {<DT|PP\$>?<JJ>*<NN>}   # Frases que tengan determinante/posesivo, adjetivo y nombre
    {<NNP>+}                # Frases con nombres propios
    """

cp = nltk.RegexpParser(Grammar02)
Result02 = cp.parse(Sentence02)
print Result02
result02.draw
```

Analizando frases con NLTK (9)

Veamos los resultados:

```
>>> print Result02
(S
  (NP Rapunzel/NNP)
  let/VBD
  down/RP
  (NP her/PP$ long/JJ golden/JJ hair/NN))
```



Explorando corpus (1)

Como hemos comentado antes, NLTK cuenta con una serie de corpus etiquetados que podemos aprovechar para hacer análisis sintáctico.

Un ejemplo: tratemos de identificar frases que sigan la secuencia *Verbo + TO + Verbo* en el Brown Corpus. Para ello, requerimos de una gramática, así como importar el Brown Corpus a nuestro intérprete. Veamos:

```
cp = nltk.RegexpParser("CHUNK: {<V.*> <TO> <V.*>}")
brown01 = nltk.corpus.brown
```

Explorando corpus (2)

Ahora, lo que vamos a hacer es elaborar un ciclo, el cual nos permita reconocer cuáles son las oraciones que, dentro del Brown, cumplan con nuestra regla. Una vez que las detectemos, las agruparemos en un listado que se desplegará en pantalla. Veamos:

```
>>> for sent in  brown.tagged_sents():
    tree = cp.parse(sent)
    for subtree in  tree.subtrees():
        if subtree.node == 'CHUNK': print subtree
```

Y el resultado es:

Explorando corpus (3)

```
IDLE 2.6.5      ==== No Subprocess ====
>>>
(CHUNK combined/VBN to/TO achieve/VB)
(CHUNK continue/VB to/TO place/VB)
(CHUNK serve/VB to/TO protect/VB)
(CHUNK wanted/VBD to/TO wait/VB)
(CHUNK allowed/VBN to/TO place/VB)
(CHUNK expected/VBN to/TO become/VB)
(CHUNK expected/VBN to/TO approve/VB)
(CHUNK expected/VBN to/TO make/VB)
(CHUNK intends/VBZ to/TO make/VB)
(CHUNK seek/VB to/TO set/VB)
(CHUNK like/VB to/TO see/VB)
(CHUNK designed/VBN to/TO provide/VB)
(CHUNK get/VB to/TO hear/VB)
(CHUNK expects/VBZ to/TO tell/VB)
(CHUNK expected/VBN to/TO give/VB)
(CHUNK prefer/VB to/TO pay/VB)
(CHUNK required/VBN to/TO obtain/VB)
(CHUNK permitted/VBN to/TO teach/VB)
(CHUNK designed/VBN to/TO reduce/VB)
(CHUNK Asked/VBN to/TO elaborate/VB)
(CHUNK got/VBN to/TO go/VB)
(CHUNK raised/VBN to/TO pay/VB)
(CHUNK scheduled/VBN to/TO go/VB)
(CHUNK cut/VBN to/TO meet/VB)
(CHUNK needed/VBN to/TO meet/VB)
(CHUNK hastened/VBD to/TO add/VB)
(CHUNK found/VBN to/TO prevent/VB)
(CHUNK continue/VB to/TO insist/VB)
(CHUNK compelled/VBN to/TO make/VB)
(CHUNK made/VBN to/TO remove/VB)
(CHUNK revamped/VBN to/TO give/VB)
(CHUNK want/VB to/TO risk/VB)
(CHUNK appear/VB to/TO spark/VB)
(CHUNK fails/VBZ to/TO consider/VB)
(CHUNK plans/VBZ to/TO call/VB)
(CHUNK going/VBG to/TO examine/VB)
(CHUNK plans/VBZ to/TO name/VB)
```

El resultado, como pueden ver, es un listado de oraciones extraído del corpus Brown que siga el patrón *Verbo* + *TO* + *Verbo*.

En aras de hacer cálculos de frecuencias, lo que pueden hacer es seleccionar todo el archivo de salida, copiarlo, y finalmente pegarlo en archivo TXT.

Gramáticas netamente sintácticas(1)

En lingüística, existen muchas perspectivas teóricas para abordar el estudio de la sintaxis. En nuestro caso, vamos a contrastar dos:

1. Estudiar el nivel sintáctico de una lengua natural como si fuera un módulo independiente, el cual se encarga de generar y procesar frases y oraciones a partir de la construcción de un modelo gramatical específico.

2. Estudiar el nivel sintáctico de una lengua natural como parte de una **interfaz** que se relaciona con otros niveles, p.e., morfológico, léxico, semántico, pragmático, etc. Así, en la configuración de una frase, intervienen varios factores, no sólo aquellos que pertenecen a la sintaxis.

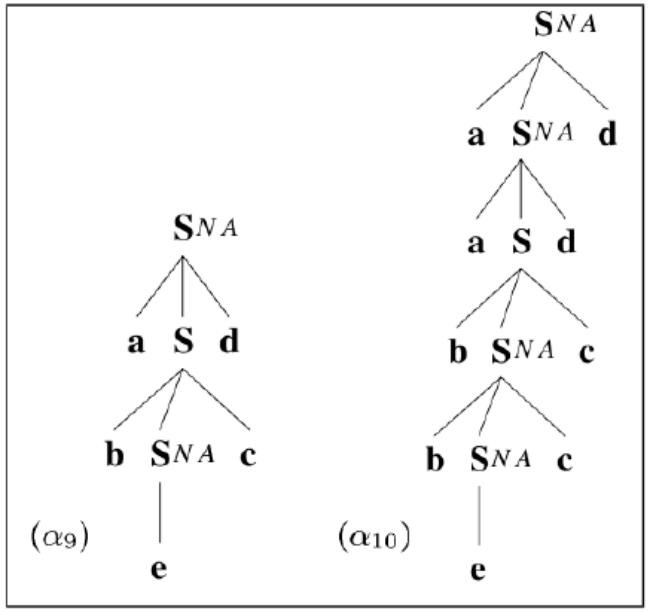
Gramáticas netamente sintácticas(2)

Un ejemplo prototípico de una gramática netamente sintáctica es un modelo llamado **Tree Adjoining Grammar** (o**TAG**).

Aravind K. Joshi (1929)

De acuerdo con Joshi, una TAG es un sistema de generación de árboles, los cuales representan cadenas sintácticas de N-tipo, conforme a un conjunto de reglas y parametros previamente dados.

Las TAGs ofrecen mejores soluciones si se comparan con las GLC, debido a que una TAG no se concentra únicamente en describir la secuencia de una cadena, sino que delinea un árbol capaz de esquematizar la estructura de dicha cadena.



Tree Adjoining Grammar (1)

TAG es una gramática formal que puede ser modelada como una gramática libre de contexto, con la diferencia que es mucho más potente, ya que en lugar de enfocarse en la generación de reglas que describan una frase, TAG formula reglas que permiten generar un árbol abstracto que sirve como armazón de *n* tipos de frases.

V • T • E **Automata theory: formal languages and formal grammars** [hide]

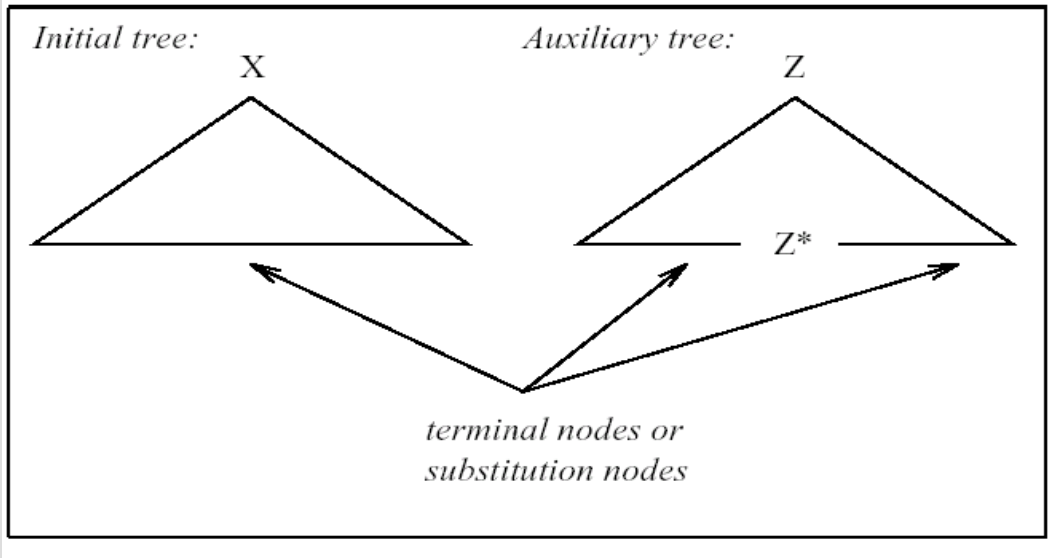
Chomsky hierarchy	Grammars	Languages	Minimal automaton
Type-0	Unrestricted	Recursively enumerable	Turing machine
—	(no common name)	Recursive	Decider
Type-1	Context-sensitive	Context-sensitive	Linear-bounded
—	Indexed	Indexed	Nested stack
—	Linear context-free rewriting systems etc.	Mildly context-sensitive	Thread automaton
—	**Tree-adjoining** etc.	**Tree-adjoining**	Embedded pushdown
Type-2	Context-free	Context-free	Nondeterministic pushdown
—	Deterministic context-free	Deterministic context-free	Deterministic pushdown
—	Visibly pushdown	Visibly pushdown	Visibly pushdown
Type-3	Regular	Regular	Finite
—	—	Star-free	Counter-free (with aperiodic finite monoid)

Each category of languages is a proper subset of the category directly above it. Any automaton and any grammar in each category has an equivalent automaton or grammar in the category directly above it.

Tree Adjoining Grammar (2)

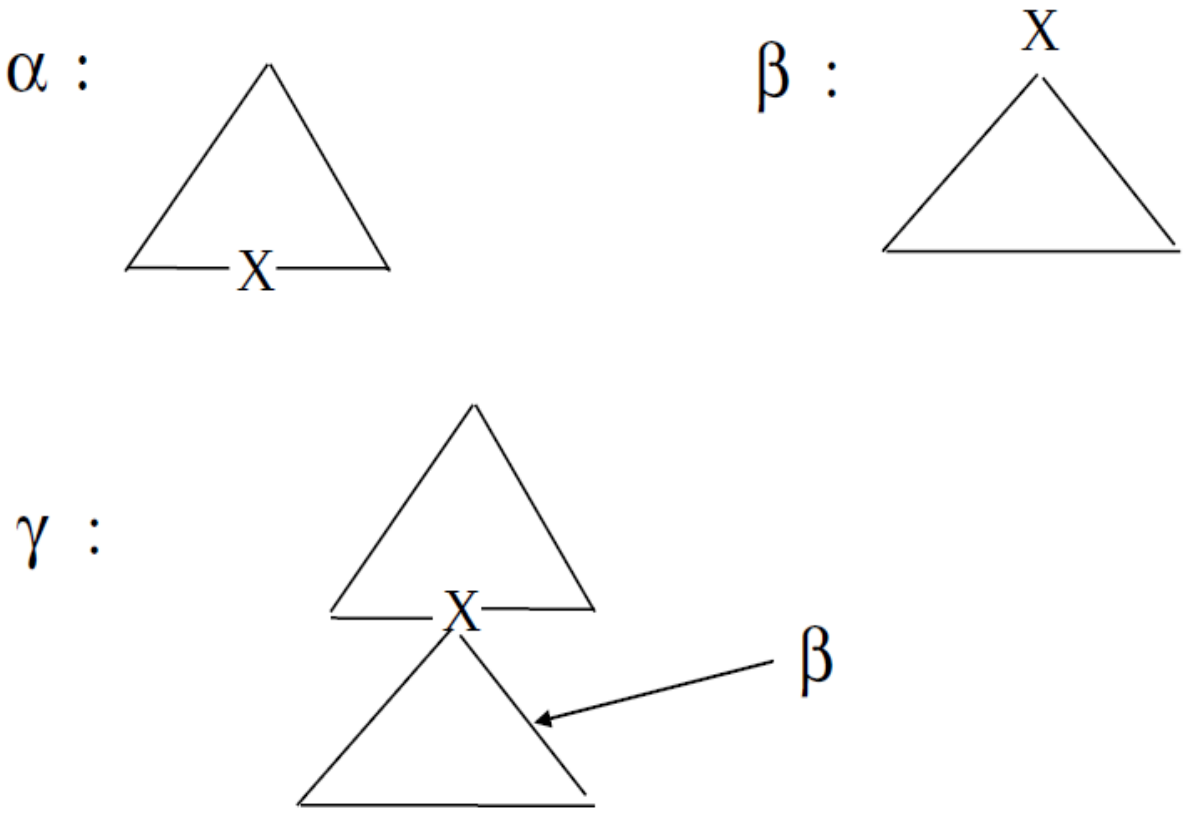
Conforme a lo anterior, entonces podemos definir una TAG del siguiente modo:

1. Un conjunto finito de símbolos terminales (Σ).
2. Un conjunto finito de símbolos no terminales (NT)
3. Cada símbolo no terminal puede ser etiquetado como (S).
4. Un conjunto finito de *árboles iniciales* (AIni).
5. Un conjunto de *árboles auxiliares* (AAux).



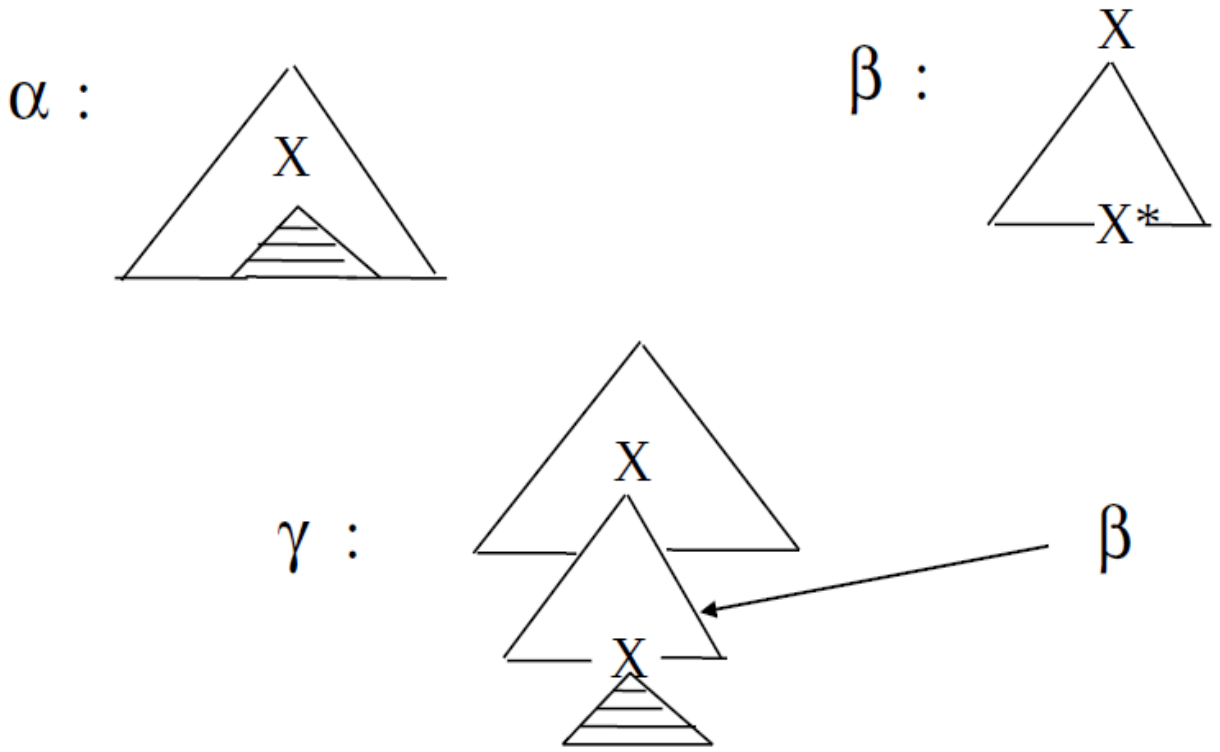
Nota: el signo (*) señala un *nodo pie* dentro de un árbol auxiliar, lo que equivale a un signo no terminal dentro de dicho árbol.

Tree Adjoining Grammar (3)



Las relaciones entre árboles son muy simples: básicamente, podemos sustituir...

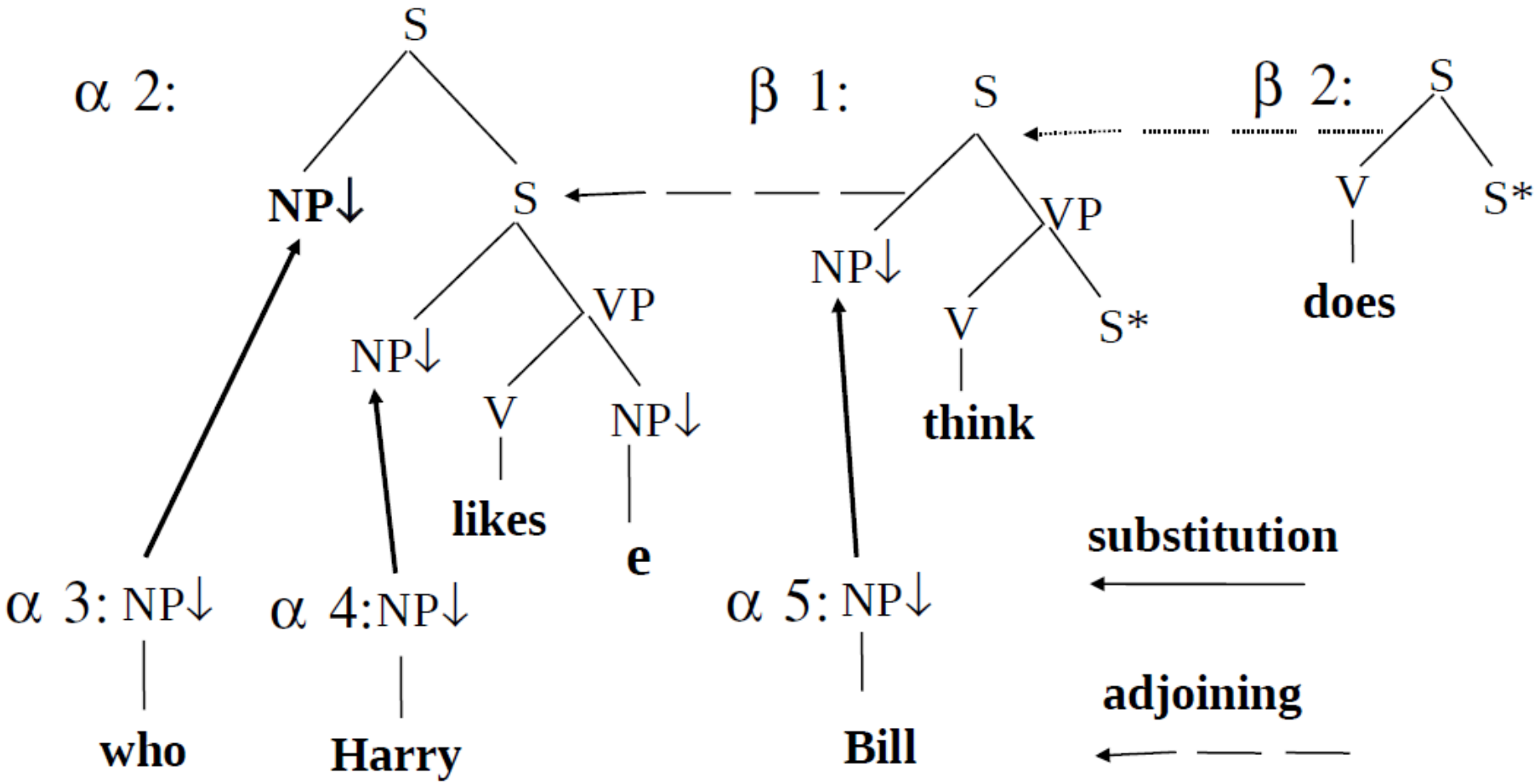
Tree Adjoining Grammar (4)



E ir adicionando árboles en los nodos correspondientes.

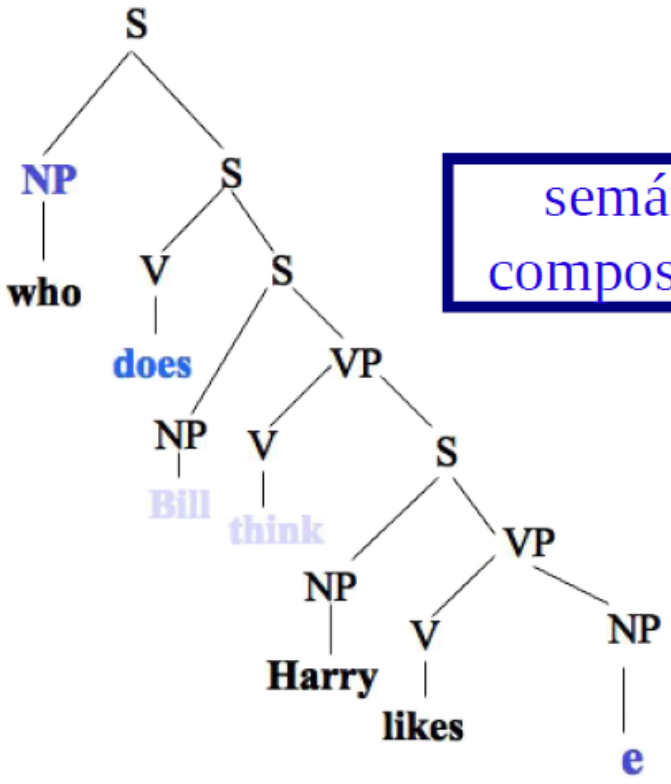
Tree Adjoining Grammar (5)

Podemos ir sustituyendo y sumando árboles conforme lo necesitemos.



Tree Adjoining Grammar (6)

De hecho, con un poco más de imaginación, una TAG ya es un buen camino para modelar una semántica composicional basada en lógica de predicados.



semántica composicional

?x:think(bill,likes(harry,x))

O cualquier otra representación posible

Tree Adjoining Grammar (7)

Más o menos podemos generar matrices semánticas del siguiente tipo, asociadas a un cierto tipo de oración.

Oración	Lógica de primer orden
John loves Mary	loves(john,mary)
Somebody sleeps	∃x sleeps(x)
Somebody loves everybody	1. ∃x ∀y loves(x,y) 2. ∀y ∃x loves(x,y)

Gracias por su atención

Blog del curso:
http://cesaraguilar.weebly.com/introduccioacuten-a-la-linguumliacutestica-computacional.html