Formalismes sintàctics

Antoni Oliver

PID_00155232

UOC
Universitat Oberta
de Catalunya
www.uoc.edu

Aquesta obra és llicencia sota la següent llicència Creative Commons: *Reconeixement - CompartirIgual 3.0* (by-sa): es permet l’ús comercial de l’obra i de les possibles obres derivades, la distribució de les quals s’ha de fer amb una llicència igual a la que regula l’obra original.

Índex

Introducció 5

Objectius 9

1. Gramàtiques lliures de context 11

2. Analitzadors per a gramàtiques lliures de context 16
- 2.1. Analitzador descendent recursiu 17
- 2.2. L'analitzador *shift-reduce* 24
- 2.3. L'analitzador *left-corner* 29
- 2.4. Analitzadors amb taules auxiliars (*Chart parsers*) 30

3. Gramàtica de trets 35
- 3.1. Processament d'estructures de trets 37

4. Les gramàtiques lliures de context probabilístiques 40
- 4.1. Inducció de gramàtiques 43
- 4.2. Analitzadors per a gramàtiques probabilístiques: l'algorisme de Viterbi 47

5. Gramàtiques de dependències 51
- 5.1. *Treebanks* de dependències de l'NLTK 52

Resum 55

Bibliografia 56

Introducció

En aquest mòdul estudiarem els formalismes i tècniques computacionals per a poder portar a terme l'anàlisi sintàctica d'una oració. En el mòdul "Anàlisi fragmental (*chunking*)" vam veure com es pot fer una anàlisi superficial (o *chunking*), que ens permetia detectar els constituents principals de l'oració, però no la relació entre els constituents.

Veurem com podem escriure gramàtiques que ens permetin analitzar oracions i també com podem aprendre a analitzar oracions a partir de corpus analitzats sintàcticament (que es coneixen també amb el nom de *treebanks*).

En l'exemple següent (`programa-8-1.py`) observem un programa que defineix una gramàtica senzilla que permet analitzar una oració.

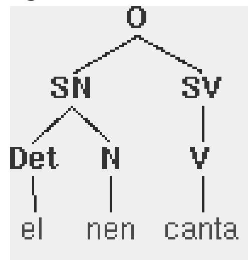
```
import nltk

gramatica_senzilla=nltk.parse_cfg("""
    O -> SN SV
    SN -> Det N
    SV -> V
    Det -> 'el'
    N -> 'nen'
    V -> 'canta'
    """)

frase=['el', 'nen', 'canta']
parser = nltk.ChartParser(gramatica_senzilla)
arbres = parser.nbest_parse(frase)
for arbre in arbres:
    print arbre
    arbre.draw()
```

Si executem aquest programa obtenim l'anàlisi en mode text i també gràficament (figura 1):

```
(O (SN (Det el) (N nen)) (SV (V canta)))
```

No sempre fer l'anàlisi sintàctica d'una oració serà tan senzill; poden aparèixer diversos problemes. En l'exemple següent intentem analitzar les oracions

Figura 1. Anàlisi sintàctica



següents: "el nen menja", "el nen menja un entrepà de formatge" i "el nen menja un entrepà de formatge a casa" (`programa-8-2.py`):

```
#! -*- coding= utf-8 -*-

import nltk

gramatica_senzilla=nltk.parse_cfg("""
    O -> SN SV
    SP -> Prep N
    SN -> Det N | Det N SP
    SV -> V | V SN | V SP | V SN SP
    Prep -> 'de' | 'a'
    Det -> 'el' | 'un'
    N -> 'nen' | 'formatge' | 'casa' | 'entrepà'
    V -> 'canta' | 'menja'
    """)
parser = nltk.ChartParser(gramatica_senzilla)
frase=['el', 'nen', 'menja']
arbres = parser.nbest_parse(frase)
for arbre in arbres:
    print arbre
    arbre.draw()
frase=['el', 'nen', 'menja', 'un', 'entrepà', 'de', 'formatge']
arbres = parser.nbest_parse(frase)
for arbre in arbres:
    print arbre
    arbre.draw()
frase=['el', 'nen', 'menja', 'un', 'entrepà', 'de', 'formatge', 'a', 'casa']

arbres = parser.nbest_parse(frase)
for arbre in arbres:
    print arbre
    arbre.draw()
```

Quan executeu aquest programa aniran apareixent anàlisis per pantalla. Quan aparegui una anàlisi el programa s'atura fins que tanquem la finestra gràfica.

A la figura 2 podem veure l'anàlisi de la frase "el nen menja", que no presenta cap problema especial. En canvi, la frase "el nen menja un entrepà de formatge" presenta el problema de determinar d'on penja el sintagma preposicional "de formatge". La nostra gramàtica ofereix dues possibilitats, que podem observar a les figures 3 i 4.

Figura 2. Anàlisi sintàctica de la frase "el nen menja"

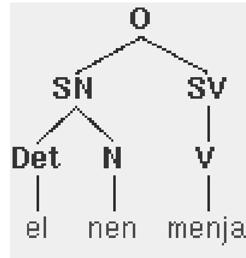


Figura 3. Primera anàlisi sintàctica possible de la frase "el nen menja un entrepà de formatge"

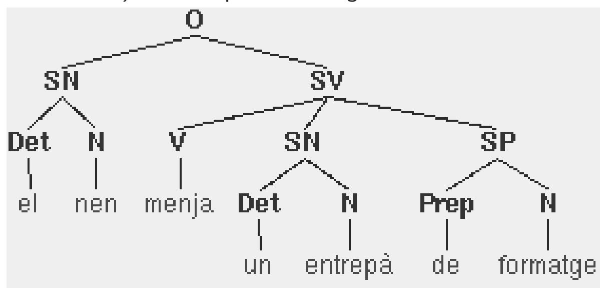
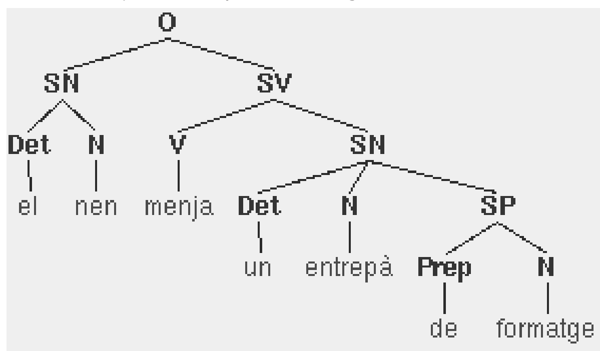


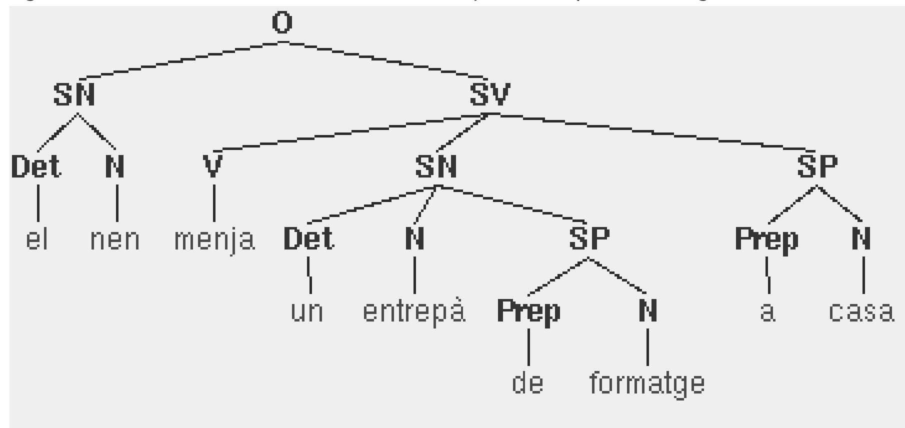
Figura 4. Segona anàlisi sintàctica possible de la frase "el nen menja un entrepà de formatge"



Per una altra banda, la frase "el nen menja un entrepà de formatge a casa" no presenta cap problema, ja que la nostra gramàtica explicita que el SV pot tenir un SP, però no dos (podem observar aquesta anàlisi a la figura 5).

Les regles de la gramàtica sovint s'anomenen *produccions* de la gramàtica.

Figura 5. Anàlisi sintàctica de la frase "el nen menja un entrepà de formatge a casa"



Objectius

Els objectius bàsics que ha d'haver aconseguit l'estudiant una vegada treballats els continguts d'aquest mòdul són els següents:

1. Conèixer els principals formalismes i tècniques computacionals per a fer l'anàlisi sintàctica d'oracions.
2. Saber crear gramàtiques senzilles amb NLTK.
3. Saber què és un *treebank* i quina utilitat poden tenir en l'entrenament de sistemes d'anàlisi sintàctica.
4. Comprendre el tipus d'anàlisi que ofereixen les gramàtiques de dependències.

1. Gramàtiques lliures de context

El nombre de possibles produccions d'una llengua és infinit, ja que no hi ha un límit superior en la llargària d'una oració. Tot i això, volem escriure programes que siguin capaços de processar (ja sigui analitzar o generar) totes les possibles oracions ben formades. Per a poder descriure mitjançant gramàtiques de mida finita un conjunt infinit caldrà fer ús de la recursivitat.

Una gramàtica és un sistema formal que especifica quines seqüències de paraules estan ben formades en una llengua i que proporciona una o més estructures de frase per a les seqüències ben formades.

Les gramàtiques lliures de context es defineixen per:

- Un conjunt de símbols inicials que cobreixen les oracions per analitzar. Aquest conjunt pot estar format per un únic símbol, com per exemple "oració"; o per més d'un símbol, com per exemple "oracio_declarativa" i "oracio_interrogativa".

- Un conjunt de símbols no terminals que permeten la representació de les categories sintàctiques.

- Un conjunt de símbols terminals que representen el diccionari: paraules del diccionari o morfemes.

- Un conjunt de regles en què el símbol de la part esquerra es reescriu en la seqüència de símbols de la part dreta. Aquestes regles sovint reben el nom de regles de reescriptura o produccions de la gramàtica.

Vegem un exemple molt senzill de gramàtica lliure de context que és capaç d'analitzar l'oració "el noi menja un entrepà".

```
O -> SN SV
SN -> Det N
SV -> V
SV -> V SN
Det -> 'el'|'un'
N -> 'noi'|'entrepà'
V -> 'menja'
```

En aquesta gramàtica el conjunt de símbols inicials està compost pel símbol "O".

El símbols no terminals són: SN, SV, Det, N, V. Els símbols terminals són: el, un, nen, entrepà, menja.

En el programa següent (`programa-8-2b.py`) demanem a NLTK que ens retorni les produccions de la gramàtica anterior:

```
#! -*- coding= utf-8 -*-

import nltk

gramatica_senzilla=nltk.parse_cfg("""
    O -> SN SV
    SN -> Det N
    SV -> V
    SV -> V SN
    Det -> 'el'|'un'
    N -> 'noi'|'entrepà'
    V -> 'menja'
    """)
for produccio in gramatica_senzilla.productions():
    print produccio
```

Si executem aquest programa obtindrem la sortida següent:

```
O -> SN SV
SN -> Det N
SV -> V
SV -> V SN
Det -> 'el'
Det -> 'un'
N -> 'noi'
N -> 'entrep\xc3\xa0'
V -> 'menja'
```

En els exemples anteriors ja hem vist com podem implementar una gramàtica senzilla com aquesta amb NLTK. Per a fer més pràctic el programa el que farem serà modificar-lo perquè llegeixi la gramàtica d'un fitxer (que indicarem per la línia d'instruccions) i que li donem la frase també per línia d'instruccions. Quan cridem el programa el primer paràmetre serà el fitxer de gramàtica i el segon la frase per analitzar. El programa queda de la manera següent (`programa-8-3.py`):

```
# -*- coding: utf-8 -*-
import sys
import nltk

print " Indica el nom de la gramatica: ",
fgramatica=raw_input()

print "Escriu la frase per analitzar: ",
frase=raw_input()

gramatica=nltk.data.load('file:%s' % fgramatica)

parser=nltk.ChartParser(gramatica)

arbres=parser.nbest_parse(frase.split())

for arbre in arbres:
    print arbre
    arbre.draw()
```

Si el nostre fitxer de gramàtica (`gramatica1.cfg`) conté:

```
O -> SN SV
SN -> Det N
SV -> V
SV -> V SN
Det -> 'el'|'un'
N -> 'noi'|'entrepà'
V -> 'menja'
```

Ara ja podrem executar el programa i indicar-li el fitxer de gramàtica i la frase que volem que analitzi. A partir d'aquest moment ens podem centrar en la creació de gramàtiques creant els arxius corresponents i cridant al programa que acabem de presentar. Una gramàtica tan senzilla com l'anterior no ens permet fer distincions, per exemple, entre verbs transitius i verbs intransitius. Ampliem una mica els símbols terminals per a afegir una mica de lèxic (`gramatica2.cfg`).

```
O -> SN SV
SN -> Det N
SV -> V
SV -> V SN
Det -> 'el'|'un'|'les'
N -> 'noi'|'entrepà'|'notes'|'professor'
V -> 'menja'|'diu'
```

Aquesta gramàtica pot analitzar tant “el professor diu les notes” com “*El professor diu”. Aquesta segona oració no és gramatical, ja que el verb *dir* requereix un complement directe. Amb NTLK podem escriure fàcilment totes les produccions de la nostra gramàtica.

Podem modificar el conjunt de símbols no terminals per a especificar la transitivitat i intransitivitat dels verbs. Així, tindrem un símbol VI per als verbs intransitius i un símbol VT per als verbs transitius. Aquí definirem verbs transitius com aquells verbs que exigeixen un complement directe i intransitius els que no n’exigeixen. Així, la gramàtica quedarà (`gramatica3.cfg`).

```
O -> SN SV
SN -> Det N
SV -> VI
SV -> VI SN
SV -> VT SN
Det -> ’el’|’un’|’les’
N -> ’noi’|’entrepà’|’notes’|’professor’
VI -> ’menja’
VT -> ’diu’
```

Aquesta gramàtica pot analitzar “el professor diu les notes”, però no “*el professor diu”. En canvi pot analitzar tant “el professor menja un entrepà” com “el professor menja”.

Ara bé, en la gramàtica anterior no hem expressat la concordança determinant-nom i pot analitzar una oració de l’estil “les noi menja les entrepà”.

Per a poder solucionar aquest problema haurem de crear encara més símbols no terminals per a poder preveure el fenomen de la concordança nom-determinant. Ho podem observar a la gramàtica següent (`gramatica4.cfg`).

```
O -> SN SV
SN -> Det_m_s N_m_s
SN -> Det_f_s N_f_s
SN -> Det_m_p N_m_p
SN -> Det_f_p N_f_p

SV -> VI
SV -> VI SN
SV -> VT SN
Det_m_s -> ’el’|’un’
Det_f_s -> ’la’|’una’
Det_m_p -> ’els’|’uns’
Det_f_p -> ’les’|’unes’
```

```
N_m_s -> 'noi'|'entrepà'|'professor'
N_f_s -> 'noia'|'professora'
N_m_p -> 'nois'|'entrepans'|'professors'
N_f_p -> 'noies'|'professores'
VI -> 'menja'|'mengen'
VT -> 'diu'|'diuen'
```

Exercici

Modifiqueu la gramàtica `gramatica4.cfg` per a introduir-hi la concordança subjecte-verb, de manera que pugui analitzar les oracions:

el noi menja un entrepà
els nois mengen un entrepà

però no les oracions

*el noi mengen un entrepà
*els nois menja un entrepà

2. Analitzadors per a gramàtiques lliures de context

Una gramàtica és una especificació declarativa que diu quines oracions estan ben construïdes. Des del punt de vista informàtic són unes cadenes de text, no són uns programes que facin cap tasca. Per a poder fer les anàlisis sintàctiques necessitem uns algorismes capaços d'interpretar aquestes gramàtiques i arribar a trobar l'anàlisi o anàlisis possibles. Aquests algorismes s'anomenen *analitzadors* o *parsers*.

Els analitzadors es poden classificar segons la metodologia que facin servir per a trobar l'anàlisi o anàlisis possibles. Una primera classificació es pot establir sobre si la cerca de l'anàlisi la fan de dalt a baix o bé de baix a dalt. D'aquesta manera poden tenir dos tipus de *parsers*:

- ***Parsers* descendents**: parteixen del símbol inicial *O* i van cap a baix fins que arriben als símbols terminals.

- ***Parsers* ascendents**: parteixen dels símbols terminals de la frase per analitzar i intenten arribar al símbol inicial *O*.

Un **analitzador descendent** parteix de la descripció gramatical (la gramàtica) i busca en el text d'entrada les seqüències que es corresponen amb les permeses per la descripció gramatical. Un **analitzador descendent**, en canvi, parteix del text i busca a la gramàtica regles que permetin les seqüències presents en el text.

A part de la direcció bàsica del procés d'anàlisi (ascendent o descendent), hi ha altres factors que determinen l'estratègia final d'un analitzador:

- Una primera consideració és si l'anàlisi es desenvolupa **en profunditat** (*depth-first*) o **en amplada** (*breadth-first*). En l'estratègia en profunditat cada una de les hipòtesis s'explora completament abans d'iniciar l'explotació d'una nova hipòtesi. És a dir, es va recorrent l'arbre fins que s'arriba a l'anàlisi o bé ja no es pot continuar. En canvi, l'estratègia per amplada manté més d'una hipòtesi (de fet totes les possibles) al mateix temps, i avança pas per pas cada una de les hipòtesis. D'aquesta manera les hipòtesis o recorreguts per a l'arbre d'anàlisi es van descartant o bé una o més arriben fins a l'anàlisi final.

- Un altre factor que cal tenir en compte és la manera com els analitzadors recorren l'entrada. Les opcions bàsiques són d'esquerra a dreta, de dreta a esquerra i fins i tot des del mig cap a la dreta i cap a l'esquerra alternativament.

En aquest apartat veurem quatre tipus d'*analitzadors*:

Lectures recomanades

S'han desenvolupat un gran nombre d'estratègies d'anàlisi. Es pot trobar una comparació pràctica de diverses estratègies a Slocum (1981). Qui vulgui ampliar aquest tema pot consultar Carroll (2003) i Jurafsky i Martin (2000).

1) *Analitzador descendent recursiu* (*recursive descent parser*): com el seu nou indica, es tracta d'un analitzador descendent.

2) *Shift-reduce parser*: és un analitzador ascendent.

3) *Left-corner parser*: és un analitzador descendent, però que fa servir un filtratge ascendent.

4) *Chart parser*: és un analitzador que fa servir taules auxiliars

2.1. Analitzador descendent recursiu

NLTK disposa d'un analitzador descendent recursiu*, el RecursiveDescentParser. Si en una sessió interactiva de Python fem:

*En anglès, *recursive descent parser.*

```
help(nltk.RecursiveDescentParser)
```

Obtindrem una bona explicació de com funciona, que tradueixo aproximadament en les línies següent:

> El `RecursiveDescentParser` és un analitzador de gramàtiques lliures de context que analitza els textos expandint de manera recursiva els extrems de l'arbre i comparant-los amb el text d'entrada.

El RecursiveDescentParser fa servir una llista d'ubicacions de l'arbre per a recordar quins subarbres encara no s'han expandit i quines fulles (extrems de l'arbre) encara no han coincidit amb el text d'entrada. Cada ubicació de l'arbre consisteix en una llista d'índexs dels fills que especifiquen la ruta des de l'arrel de l'arbre fins al subarbre o fins a la fulla; podeu consultar la documentació de referència de *Tree* per a obtenir més informació sobre sobre les ubicacions de l'arbre.

Quan l'analitzador comença l'anàlisi construeix un arbre que conté únicament el símbol inicial i una frontera que conté la ubicació del node arrel de l'arbre. Llavors expandeix l'arbre per a cobrir el text d'entrada fent servir el procediment recursiu següent:

- Si la frontera està buida i el text està cobert per l'arbre, retorna l'arbre com a una anàlisi possible.

- Si la frontera està buida, i el text no queda cobert per l'arbre d'anàlisi, llavors no retorna cap anàlisi.

- Si el primer element de la frontera és un subarbre fa servir les produccions de la gramàtica lliure de context per a expandir-lo. Per a cada producció aplicable, afegeix el fill del subarbre expandit a la frontera, i troba de manera recursiva totes les anàlisis que poden ser generades pel nou arbre i frontera.

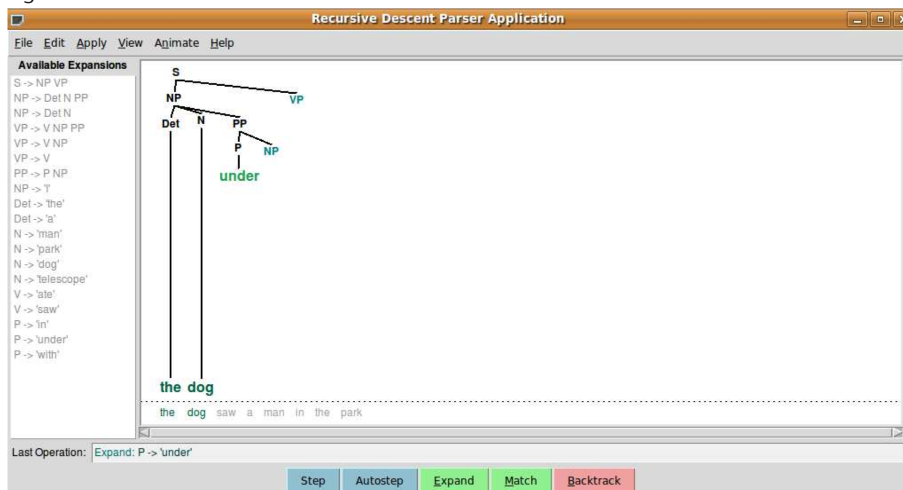
- Si el primer element de la frontera és un símbol terminal (paraula) llavors mira si coincideix amb la paraula següent del text d'entrada. Elimina la paraula de la frontera i troba de manera recursiva totes les anàlisis que poden ser generades pel nou arbre i frontera.

Disposem d'una demostració interactiva d'aquest analitzador; la podem executar fent, en una sessió interactiva:

```
nltk.app.rdparser()
```

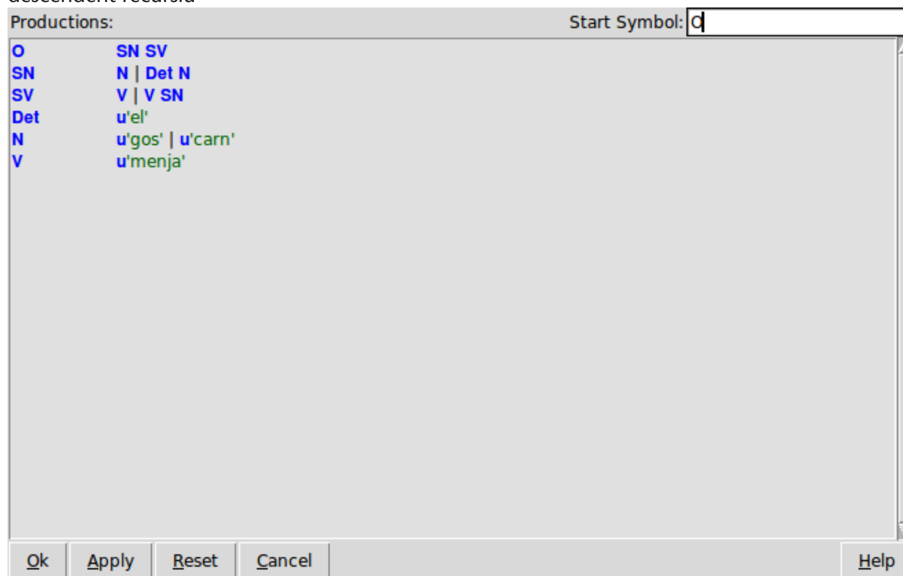
Podem observar l'aspecte d'aquesta versió de demostració a la figura 6. Si feu clic al botó "Autostep" podreu observar tot el procés fins a arribar a una anàlisi possible.

Figura 6. *Demo* de l'analitzador descendent recursiu



En aquesta versió de demostració podem modificar la gramàtica i posar una altra frase a analitzar. Ara analitzarem una frase molt senzilla amb una gramàtica molt senzilla i anirem veient els passos que fa l'analitzador. Per a editar la gramàtica feu "Edit > Edit Grammar". Escriviu la gramàtica que es mostra a la figura 7. No oblideu de posar que el símbol inicial és *O*. Per a escriure les regles feu servir -> tot i que l'editor dibuixarà un tabulador.

Figura 7. Gramàtica catalana senzilla per a provar la versió de demostració de l'analitzador descendent recursiu



Ara editem la frase per analitzar fent "Edit > Edit text" i posant "el gos menja carn". En les figures següents veiem el procés d'anàlisi fins a arribar a una anàlisi vàlida. El procés encara continuaria fent tornada enrere i analitzant altres possibilitats, tot i que no arribaria a cap altra anàlisi vàlida.

Nota

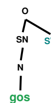
Les funcionalitats d'edició de la gramàtica i de la frase per analitzar no estan molt desenvolupades. Per aquest motiu he inclòs en la distribució dels programes el programa `rdparser_app-cat.py`, que podeu executar directament amb la gramàtica i la frase modificades.

1. Comencem pel símbol inicial

2. Expandeix la regla O -> SN SV

3. Expandeix la regla SN -> N

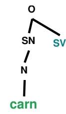
4. Prova amb N -> gos



5. Fa tornada enrere

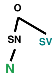
el gos menja carn

6. Prova amb N -> carn



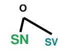
el gos menja carn

7. No hi ha cap altra possibilitat amb SN -> N



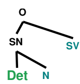
el gos menja carn

8. Fa tornada enrere



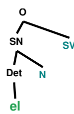
el gos menja carn

9. Expandeix la regla SN -> Det N



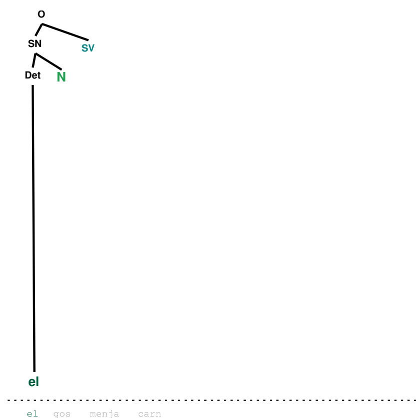
el gos menja carn

10. Prova amb Det -> el

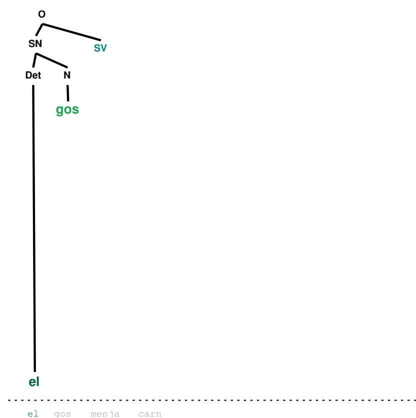


el gos menja carn

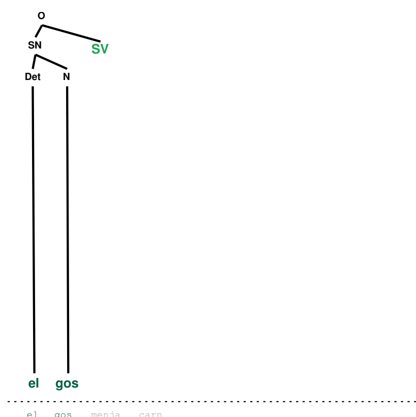
11. Coincideix amb la frase d'entrada



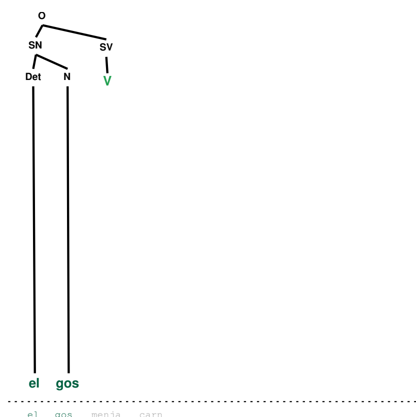
12. Prova amb N -> gos



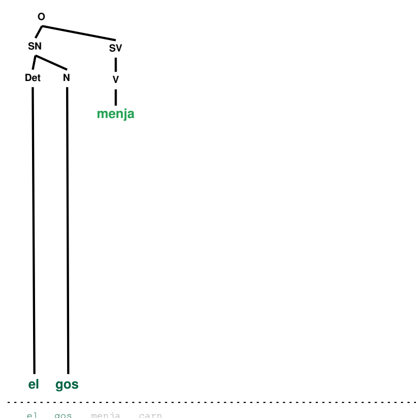
13. Coincideix amb la frase d'entrada



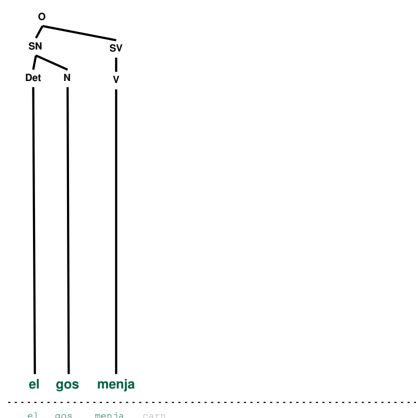
14. Expandeix SV -> V



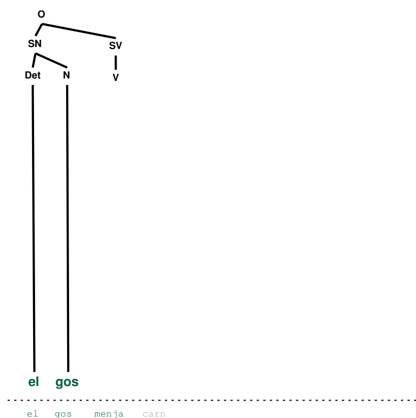
15. Prova amb V -> menja



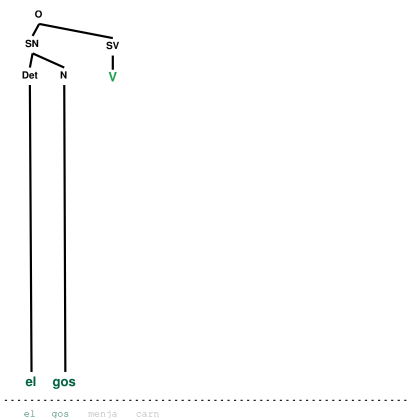
16. Coincideix amb la frase d'entrada



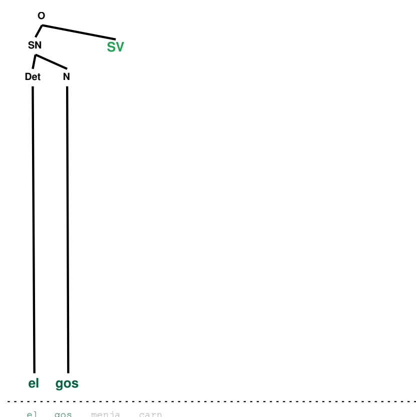
17. No assoleix una anàlisi completa



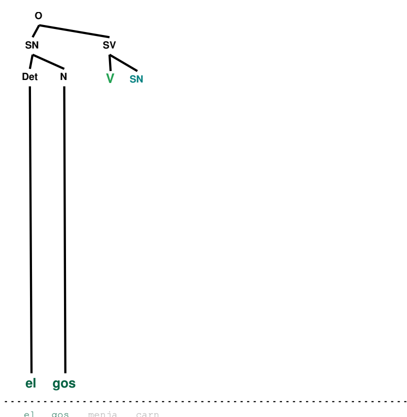
18. No hi ha més V



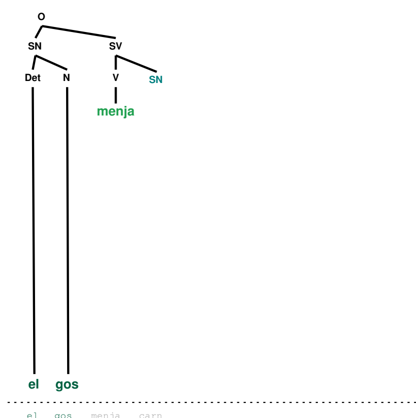
19. Fa tornada enrere



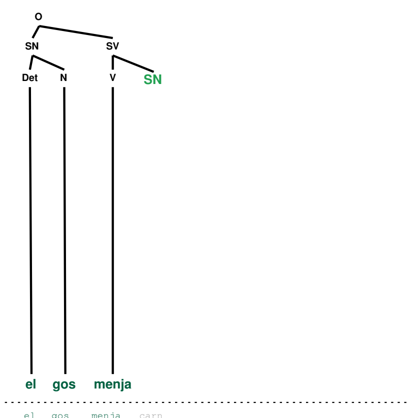
20. Expandeix la regla SV -> V SN



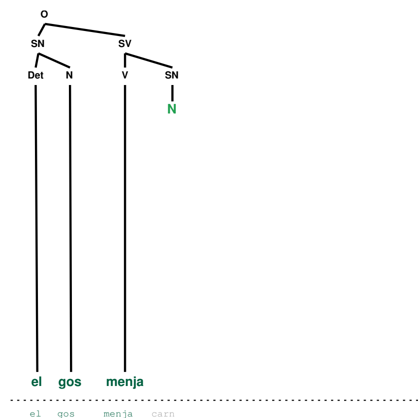
21. Prova amb V -> menja



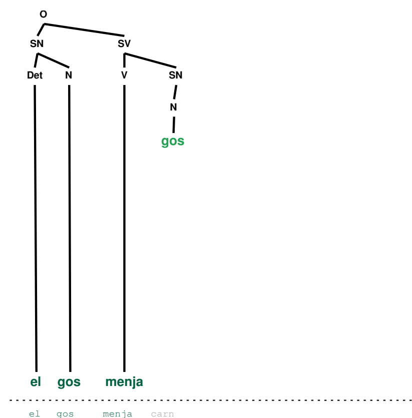
22. Coincideix amb la frase d'entrada



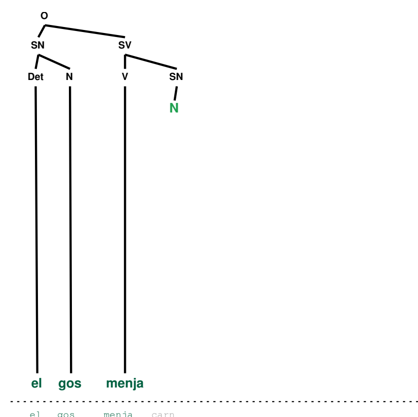
23. Expandeix la regla SN -> N



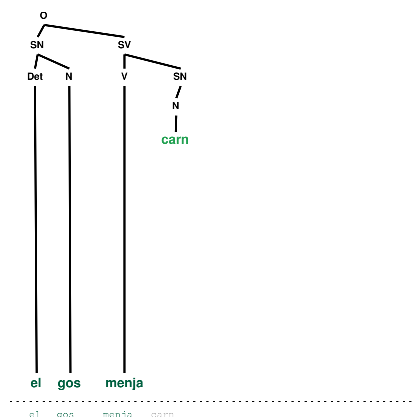
24. Prova amb N -> gos



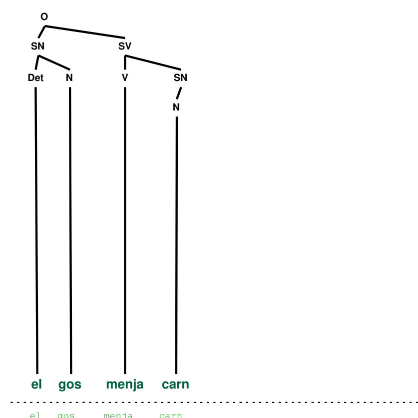
25. No coincideix amb la frase d'entrada



26. Prova amb N -> carn



27. Assoleix una anàlisi completa

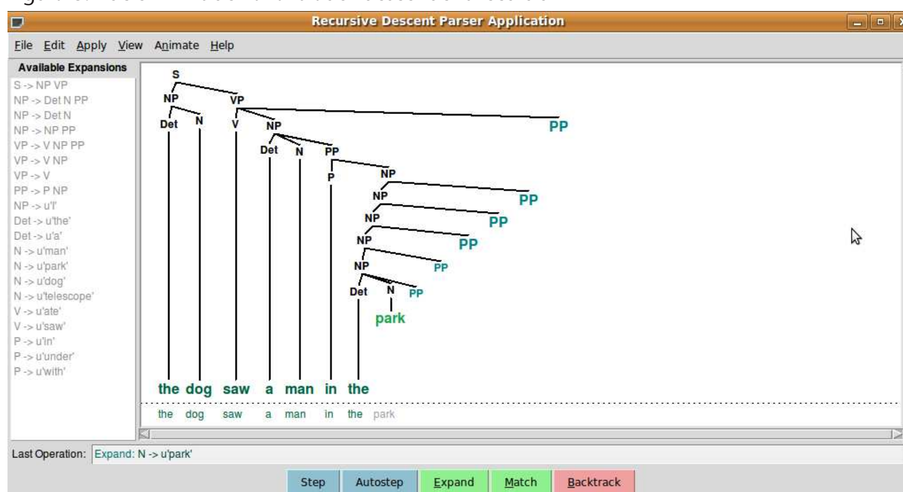


Tot i que ha assolit una anàlisi completa, el procés d'anàlisi no acabaria aquí, ja que l'analitzador intentarà buscar altres anàlisis possibles.

Els analitzadors descendents recursius tenen una sèrie de problemes importants:

- Les produccions recursives a l'esquerra, com per exemple, `NP -> NP PP`, fan que l'analitzador entri un un bucle infinit. Per a verificar això, executeu la versió de demostració de l'analitzador descendent recursiu i modifiqueu la gramàtica que ve per defecte afegint una regla com l'anterior. Executeu la versió de demostració i feu clic en "Autostep" perquè s'executi automàticament. Veureu com entra en aquest bucle infinit. A la figura 8 es pot observar l'estat de l'anàlisi després d'una estona.

Figura 8. Bucle infinit de l'analitzador descendent recursiu



- L'analitzador perd molt de temps en tenir en compte paraules i estructures que no es corresponen amb la frase d'entrada.

- El procés de tornada enrere descarta alguns constituents que ja estan analitzats i que potser s'han de tornar a fer servir més tard.

Així doncs, els analitzadors descendents fan servir la gramàtica per a predir com serà l'entrada, quan l'entrada ja està disponible des del principi. Els analitzadors ascendents, que veurem en els propers subapartats, tenen en compte la frase per analitzar des del principi.

2.2. L'analitzador *shift-reduce*

NLTK implementa un analitzador *shift-reduce*, el ShiftReduceParser. Si en una sessió interactiva de Python fem:

```
help(nltk.ShiftReduceParser)
```

obtenim tota l'ajuda sobre aquest analitzador, que traduïm aproximadament aquí:

> El ShiftReduceParser és un analitzador ascendent senzill per a gramàtiques lliures de context que fa servir dues operacions, *shift* (traslladar) i *reduce* (reduir), per a trobar una única anàlisi d'un text.

El ShiftReduceParser manté una pila (*stack*), que registra l'estructura d'una part del text. Aquesta pila és una llista de cadenes (*strings*) i d'arbres (*trees*), que en conjunt cobreixen una porció del text. Per exemple, en analitzar la frase "the dog saw the man" amb una gramàtica típica, el ShiftReduceParser produirà la pila següent, que inclou "the dog saw":

```
[(NP: (Det: 'the') (N: 'dog')), (V: 'saw')]
```

El ShiftReduceParser intenta estendre la pila per a cobrir tot el text i combinar els elements de la pila en un únic arbre per a produir una anàlisi completa per a la frase.

Inicialment, la pila està buida. S'estén per a cobrir el text, d'esquerra a dreta, aplicant repetidament dues operacions:

- *shift* (traslladar): mou un *token* del principi del text al final de la pila.
- *reduce* (reduir): fa servir una producció de la gramàtica lliure de context per a combinar els elements de més a la dreta de la pila en un únic arbre.

Sovint es pot aplicar més d'una operació sobre una pila determinada. En aquest cas, el ShiftReduceParser fa servir l'heurística següent per a determinar quina operació ha de portar a terme:

- Només fa servir l'operació *shift* si no hi ha disponible cap operació *reduce*.
- Si hi ha disponible més d'una operació *reduce*, llavors aplica la reducció que correspon a la regla que apareix abans en la gramàtica.

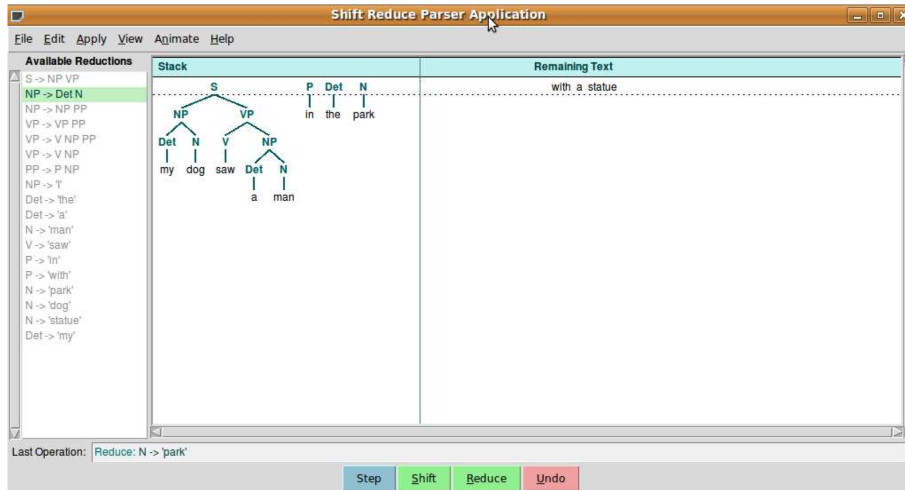
Cal tenir en compte que aquestes heurístiques no garanteixen que es triï una operació que porti a una anàlisi del text. Fins i tot, si hi ha diverses anàlisis possibles el ShiftReduceParser en retornarà només una.

També disposem d'una demostració interactiva d'aquest analitzador; la podem executar fent, en una sessió interactiva:

```
nltk.app.srparser()
```

Podem observar l'aspecte d'aquesta versió de demostració a la figura 9. Si feu clic al botó "Autostep" podreu observar tot el procés fins a arribar a una anàlisi possible.

Figura 9. *Demo* de l'analitzador *shift-reduce*



A les figures següents podem observar l'anàlisi de l'oració "el gos menja" amb una gramàtica adient amb l'analitzador *shift-reduce*. Podeu executar el programa `srparser_app-cat.py` per a observar el mateix procés. Observem que en aquest cas l'analitzador arriba a una anàlisi vàlida.

1. Shift: 'el'

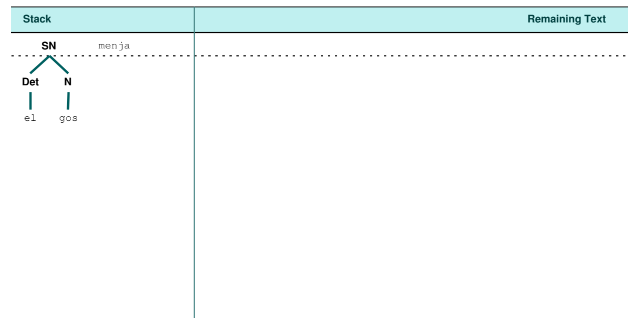
2. Reduce: Det -> 'el'

3. Shift: 'gos'

4. Reduce N -> 'gos'

5. Reduce: SN -> Det N

6. Shift: 'menja'



7. Reduce: V -> 'menja'

8. Reduce: SV -> V

9. Reduce O -> SN SV

Podem observar que el procés d'anàlisi s'ha fet en menys passos que en el cas de l'analitzador descendent recursiu. Tot i això, cal tenir en compte que un analitzador com aquest no sempre troba l'anàlisi tot i que n'hi hagi. Si executeu el programa `srparser_app-cat2.py` veurem el procés d'anàlisi de la frase "el gos menja carn" amb la mateixa gramàtica i veurem que no assoleix cap anàlisi vàlida.

1. Shift: 'el'

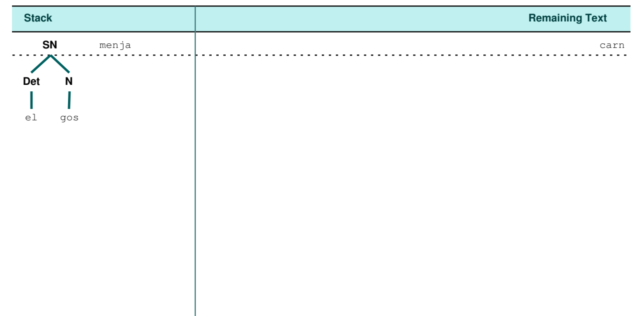
2. Reduce: Det -> 'el'

3. Shift: 'gos'

4. Reduce: N -> 'gos'

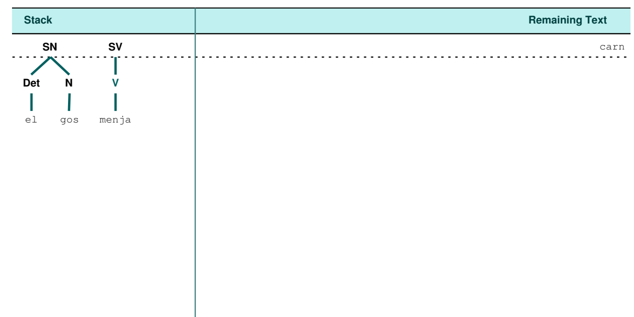
5. Reduce: SN -> Det N

6. Shift: 'menja'

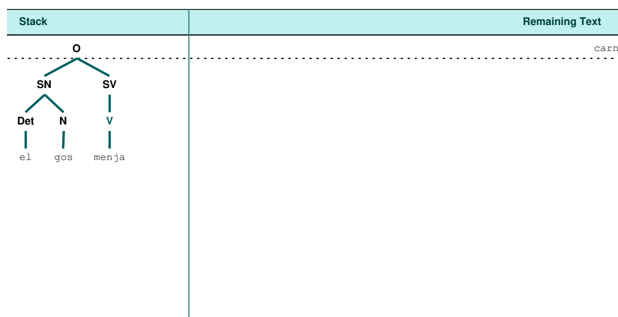


7. Reduce: V -> 'menja'

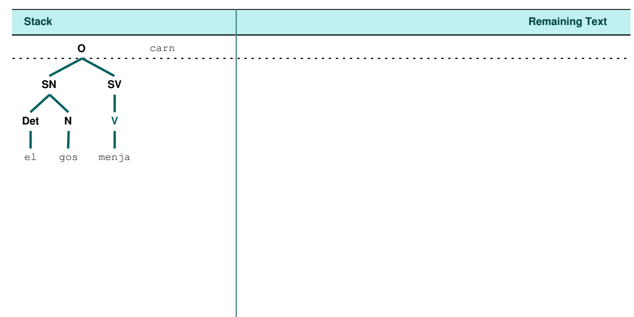
8. Reduce: SV -> V



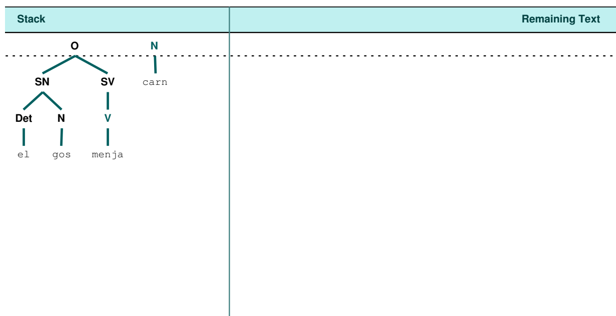
9. Reduce: -> SN SV



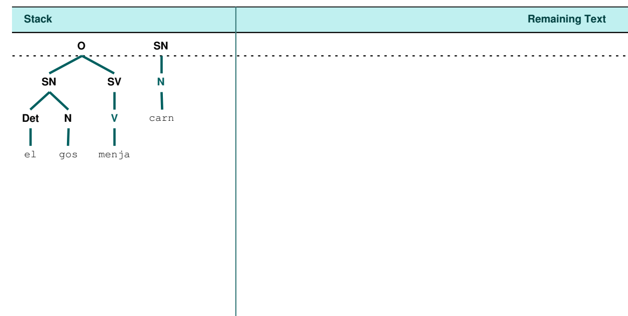
10. Shift: carn



11. Reduce: N -> 'carn'



12. Reduce: SN -> N (i *failure*)



2.3. L'analitzador *left-corner*

Un analitzador *left-corner* és un híbrid entre les estratègies descendents i ascendents; és de fet un analitzador descendent, però que fa servir un filtratge ascendent. Amb aquesta aproximació s'evita que l'analitzador arribi a bucles infinits quan es troba amb una producció recursiva per l'esquerra. El primer que fa un analitzador *left-corner* és processar la gramàtica i construir una taula en què cada filera conté dues cel·les, la primera conté un símbol no terminal, i la segona conté totes els possibles extrems esquerres per a aquest no terminal. Si considerem la gramàtica següent:

```
O -> SN SV
SP -> Prep N
SN -> Det N | Det N SP | PronPer
SV -> V | V SN | V SP | V SN SP
PronPer -> 'ell' | 'ella'
Prep -> 'de' | 'a'
Det -> 'el' | 'un'
N -> 'nen' | 'formatge' | 'casa' | 'entrepà'
V -> 'canta' | 'menja'
```

La taula inicial que construeix l'analitzador la podem trobar a la taula 1.

Taula 1. Extrems esquerres de la gramàtica d'exemple

Categoria	Extrem esquerre (preterminals)
O	SN
SP	Prep
SN	Det, PronPer
SV	V

Cada cop que l'analitzador considera una determinada producció es verifica que la paraula d'entrada següent sigui compatible amb almenys una de les categories preterminals de la taula.

2.4. Analitzadors amb taules auxiliars (*Chart parsers*)

Una estratègia per a optimitzar el rendiment dels analitzadors és l'ús de taules auxiliars on s'emmagatzemen els resultats parcials que es van obtenint en el procés d'anàlisi. Aquests resultats parcials es poden reutilitzar més endavant si convé, sense la necessitat de tornar-los a obtenir. Aquestes taules són llistes de resultats parcials d'anàlisi, que poden ser utilitzats en qualsevol moment: indiquen a quins segments de l'entrada es corresponen, quina és la seva categoria final i quina és l'anàlisi que proposen per al segment.

NLTK implementa analitzadors amb taules auxiliars en la classe *chart*. Si fem `help(nltk.parse.chart)`, obtenim informació sobre aquesta classe, que resumim aquí:

> Aquesta classe proporciona implementacions de classes de dades i analitzadors per a analitzar amb taules auxiliars (*char parsers*) que fan servir tècniques de programació dinàmica per a analitzar un text de manera eficient. Un *chartparser* deriva arbres d'anàlisi per a un text afegint de manera interactiva hipòtesis d'anàlisi a la taula. La taula és com una pissarra on es poden compondre i combinar aquestes hipòtesis.

Quan un analitzador amb taules auxiliars comença a analitzar un text crea una nova taula que està buida i que abasta tot el text. Després afegeix noves hipòtesis a la taula d'una manera incremental. Un conjunt de *chart rules* especifica les condicions que s'han de donar perquè s'afegeixi una nova hipòtesi a la taula. L'anàlisi s'ha completat un cop la taula assoleixi un estat en què cap de les *chart rules* afegeixi cap nova hipòtesi.

NLTK també proporciona una versió de demostració d'aquest analitzador; podem accedir a aquesta versió de demostració des d'una sessió interactiva de Python. En la primera versió de demostració veurem com funciona amb una estratègia descendent:

```
>>> import nltk
>>> nltk.parse.chart.demo(1, should_print_times=False, should_print_grammar=True,
trace=1, sent="John saw a dog", numparses=1)
* Grammar
Grammar with 18 productions (start state = S)
    S -> NP VP
    PP -> 'with' NP
    NP -> NP PP
    VP -> VP PP
    VP -> Verb NP
    VP -> Verb
    NP -> Det Noun
    NP -> 'John'
    NP -> 'I'
    Det -> 'the'
    Det -> 'my'
```

```
    Det -> 'a'
    Noun -> 'dog'
    Noun -> 'cookie'
    Verb -> 'ate'
    Verb -> 'saw'
    Prep -> 'with'
    Prep -> 'under'
* Sentence:
John saw a dog
['John', 'saw', 'a', 'dog']

* Strategy: Top-down

|.   John  .   saw   .    a    .   dog   .|
|[---------]         .         .         .| [0:1] 'John'
|.         [---------]         .         .| [1:2] 'saw'
|.         .         [---------]         .| [2:3] 'a'
|.         .         .         [---------]| [3:4] 'dog'
|>         .         .         .         .| [0:0] S  -> * NP VP
|>         .         .         .         .| [0:0] NP -> * NP PP
|>         .         .         .         .| [0:0] NP -> * Det Noun
|>         .         .         .         .| [0:0] NP -> * 'John'
|[---------]         .         .         .| [0:1] NP -> 'John' *
|[--------->         .         .         .| [0:1] S  -> NP * VP
|[--------->         .         .         .| [0:1] NP -> NP * PP
|.         >         .         .         .| [1:1] VP -> * VP PP
|.         >         .         .         .| [1:1] VP -> * Verb NP
|.         >         .         .         .| [1:1] VP -> * Verb
|.         >         .         .         .| [1:1] Verb -> * 'saw'
|.         [---------]         .         .| [1:2] Verb -> 'saw' *
|.         [--------->         .         .| [1:2] VP -> Verb * NP
|.         [---------]         .         .| [1:2] VP -> Verb *
|[-------------------]         .         .| [0:2] S  -> NP VP *
|.         [--------->         .         .| [1:2] VP -> VP * PP
|.         .         >         .         .| [2:2] NP -> * NP PP
|.         .         >         .         .| [2:2] NP -> * Det Noun
|.         .         >         .         .| [2:2] Det -> * 'a'
|.         .         [---------]         .| [2:3] Det -> 'a' *
|.         .         [--------->         .| [2:3] NP -> Det * Noun
|.         .         .         >         .| [3:3] Noun -> * 'dog'
|.         .         .         [---------]| [3:4] Noun -> 'dog' *
|.         .         [-------------------]| [2:4] NP -> Det Noun *
|.         [-----------------------------]| [1:4] VP -> Verb NP *
|.         .         [------------------->| [2:4] NP -> NP * PP
|[=======================================]| [0:4] S  -> NP VP *
|.         [----------------------------->| [1:4] VP -> VP * PP
(S (NP John) (VP (Verb saw) (NP (Det a) (Noun dog))))
```

El primer que ens mostra la versió de demostració és la gramàtica que fa servir, l'oració per analitzar i l'estratègia que farà servir. Les regles de la gramàtica que s'han pogut fer servir per a analitzar un fragment de la frase es mostren de la manera següent:

```
[---------]
```

i la seva amplada correspondrà al nombre de *tokens* que pot analitzar (que també estaran determinats per les xifres que s'expressen entre claudàtors), així, si ens fixem:

```
|.         [-----------------------------]| [1:4] VP -> Verb NP *
```

indica que la regla especificada pot analitzar els *tokens* del 2 (*saw*) al 4 (*dog*).

També podem mostrar el resultat de la versió de demostració fent servir una estratègia ascendent (ara ja no farem que escrigui la gramàtica que fa servir):

```
>>> nltk.parse.chart.demo(2, should_print_times=False, should_print_grammar=False,
trace=1, sent="John saw a dog", numparses=1)
* Sentence:
John saw a dog
['John', 'saw', 'a', 'dog']

* Strategy: Bottom-up

|.   John  .   saw   .    a    .   dog   .|
|[---------]         .         .         .| [0:1] 'John'
|.         [---------]         .         .| [1:2] 'saw'
|.         .         [---------]         .| [2:3] 'a'
|.         .         .         [---------]| [3:4] 'dog'
|>         .         .         .         .| [0:0] NP -> * 'John'
|[---------]         .         .         .| [0:1] NP -> 'John' *
|>         .         .         .         .| [0:0] S  -> * NP VP
|>         .         .         .         .| [0:0] NP -> * NP PP
|[--------->         .         .         .| [0:1] S  -> NP * VP
|[--------->         .         .         .| [0:1] NP -> NP * PP
|.         >         .         .         .| [1:1] Verb -> * 'saw'
|.         [---------]         .         .| [1:2] Verb -> 'saw' *
|.         >         .         .         .| [1:1] VP -> * Verb NP
|.         >         .         .         .| [1:1] VP -> * Verb
|.         [--------->         .         .| [1:2] VP -> Verb * NP
|.         [---------]         .         .| [1:2] VP -> Verb *
|.         >         .         .         .| [1:1] VP -> * VP PP
```

```
|[-------------------]         .         .| [0:2] S  -> NP VP *
|.         [--------->         .         .| [1:2] VP -> VP * PP
|.         .         >         .         .| [2:2] Det -> * 'a'
|.         .         [---------]         .| [2:3] Det -> 'a' *
|.         .         >         .         .| [2:2] NP -> * Det Noun
|.         .         [--------->         .| [2:3] NP -> Det * Noun
|.         .         .         >         .| [3:3] Noun -> * 'dog'
|.         .         .         [---------]| [3:4] Noun -> 'dog' *
|.         .         [-------------------]| [2:4] NP -> Det Noun *
|.         .         >         .         .| [2:2] S  -> * NP VP
|.         .         >         .         .| [2:2] NP -> * NP PP
|.         [-----------------------------]| [1:4] VP -> Verb NP *
|.         .         [------------------->| [2:4] S  -> NP * VP
|.         .         [------------------->| [2:4] NP -> NP * PP
|[=======================================]| [0:4] S  -> NP VP *
|.         [----------------------------->| [1:4] VP -> VP * PP
(S (NP John) (VP (Verb saw) (NP (Det a) (Noun dog))))
```

I també amb una estratègia ascendent *left-corner*:

```
>>> nltk.parse.chart.demo(3, should_print_times=False, should_print_grammar=False,
trace=1, sent="John saw a dog", numparses=1)
* Sentence:
John saw a dog
['John', 'saw', 'a', 'dog']

* Strategy: Bottom-up left-corner

|.   John  .   saw   .    a    .   dog   .|
|[---------]         .         .         .| [0:1] 'John'
|.         [---------]         .         .| [1:2] 'saw'
|.         .         [---------]         .| [2:3] 'a'
|.         .         .         [---------]| [3:4] 'dog'
|[---------]         .         .         .| [0:1] NP -> 'John' *
|[--------->         .         .         .| [0:1] S  -> NP * VP
|[--------->         .         .         .| [0:1] NP -> NP * PP
|.         [---------]         .         .| [1:2] Verb -> 'saw' *
|.         [--------->         .         .| [1:2] VP -> Verb * NP
|.         [---------]         .         .| [1:2] VP -> Verb *
|.         [--------->         .         .| [1:2] VP -> VP * PP
|[-------------------]         .         .| [0:2] S  -> NP VP *
|.         .         [---------]         .| [2:3] Det -> 'a' *
|.         .         [--------->         .| [2:3] NP -> Det * Noun
|.         .         .         [---------]| [3:4] Noun -> 'dog' *
|.         .         [-------------------]| [2:4] NP -> Det Noun *
|.         .         [------------------->| [2:4] S  -> NP * VP
```

```
|.         .         [-------------------->| [2:4] NP -> NP * PP
|.         [------------------------------]| [1:4] VP -> Verb NP *
|.         [------------------------------>| [1:4] VP -> VP * PP
|[=======================================]| [0:4] S  -> NP VP *
(S (NP John) (VP (Verb saw) (NP (Det a) (Noun dog))))
```

3. Gramàtica de trets

Les gramàtiques lliures de context que hem presentat a l'apartat 1 descriuen els constituents sintàctics amb l'ajut d'un conjunt limitat d'etiquetes de categories. Aquestes etiquetes poden ser adequades per a descriure l'estructura més general d'una oració, però no són pràctiques si volem fer distincions gramaticals més fines. Si prenem com a exemple la gramàtica `gramatica4.cfg`, en què volem descriure la concordança determinant-nom o sintagma nominal-verb veiem que el formalisme es complica.

Les gramàtiques de trets* permeten expressar trets específics per a les diferents categories. La mateixa gramàtica `gramatica4.cfg` es pot expressar d'una manera molt més eficient fent servir trets (`gramatica5.fcfg`).

*En anglès, *feature-based grammars.*

```
% start O
##############################
# Regles gramatica
##############################
# Regles d'expansió d'O
O -> SN[NUM=?n] SV[NUM=?n]
# Regles d'expansió de SN
SN[NUM=?n] -> N[NUM=?n]
SN[NUM=?n] -> Det[NUM=?n, GEN=?g] N[NUM=?n, GEN=?g]
# Regles d'expansió de SV
SV[TENSE=?t, NUM=?n] -> VI[TENSE=?t, NUM=?n]
SV[TENSE=?t, NUM=?n] -> VI[TENSE=?t, NUM=?n] SN
SV[TENSE=?t, NUM=?n] -> VT[TENSE=?t, NUM=?n] SN
##############################
# Regles lèxiques
##############################
Det[NUM=sg, GEN=m] -> 'el' | 'un'
Det[NUM=sg, GEN=f] -> 'la' | 'una'
Det[NUM=pl, GEN=m] -> 'els' | 'uns'
Det[NUM=pl, GEN=f] -> 'les' | 'uns'
N[NUM=sg, GEN=m] -> 'noi' | 'entrepà' | 'professor'
N[NUM=sg, GEN=f] -> 'noia' | 'professora' | 'nota'
N[NUM=pl, GEN=m] -> 'nois' | 'entrepans' | 'professors'
N[NUM=pl, GEN=f] -> 'noies' | 'professores' | 'notes'
VI[TENSE=pres,  NUM=sg] -> 'menja'
VT[TENSE=pres, NUM=sg] -> 'diu'
VI[TENSE=pres,  NUM=pl] -> 'mengen'
VT[TENSE=pres, NUM=pl] -> 'diuen'
```

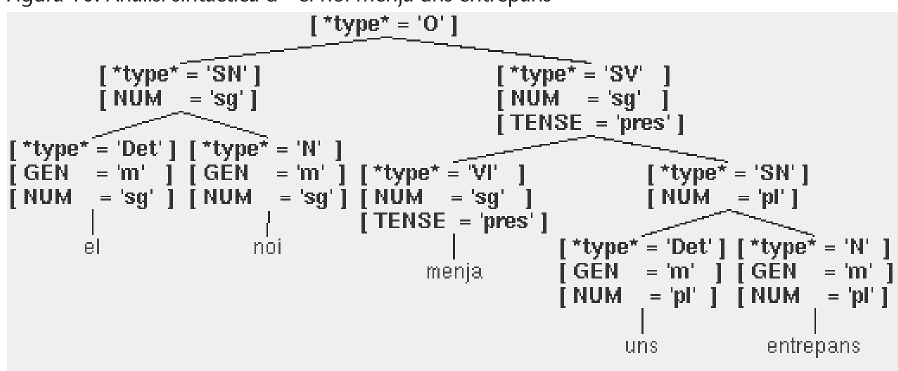
El programa que utilitzarem és molt semblant a l'anterior i demanarà una gramàtica i una frase per analitzar (`programa-8-4.py`).

```
# -*- coding: utf-8 -*-
import nltk
from nltk.parse import load_parser
print " Indica el nom de la gramàtica: ",
fgramatica=raw_input()

print "Escriu la frase per analitzar: ",
frase=raw_input()

tokens=frase.split()

gramatica=load_parser('file:%s' % fgramatica)

arbres=gramatica.nbest_parse(tokens)

for arbre in arbres:
    print arbre
    arbre.draw()
```

Si demanem que analitzi l'oració "el noi menja uns entrepans" ens oferirà una anàlisi en mode text i com a arbre (figura 10):

```
(O[]
  (SN[NUM='sg']
    (Det[GEN='m', NUM='sg'] el)
    (N[GEN='m', NUM='sg'] noi))
  (SV[NUM='sg', TENSE='pres']
    (VI[NUM='sg', TENSE='pres'] menja)
    (SN[NUM='pl']
      (Det[GEN='m', NUM='pl'] uns)
      (N[GEN='m', NUM='pl'] entrepans))))
```

Figura 10. Anàlisi sintàctica d' "el noi menja uns entrepans"



3.1. Processament d'estructures de trets

En aquest subapartat veurem com es poden processar amb NLTK les estructures de trets. També presentarem l'operació fonamental amb estructures de trets, que s'anomena *unificació*. Les gramàtiques de trets treballen fonamentalment amb aquestes estructures i amb aquesta operació fonamental.

En NLTK les estructures de trets es declaren amb el constructor *FeatStruct()*. Aquesta estructura és semblant a un diccionari i podem accedir a cada un dels valors i assignar-los de la manera habitual. En una sessió interactiva de Python proveu el següent (no oblideu de fer "import nltk"):

```
>>> et1=nltk.FeatStruct(CAT='Det', NUM='sg')
>>> et1['GEN']='m'
>>> print et1['CAT']
Det
```

També es poden definir estructures de trets amb valors complexos, és a dir, que un dels seus valors sigui també una estructura de trets:

```
>>> et1=nltk.FeatStruct("[POS='N', CONC=[PER=3, NUM='pl', GEN='f']]")
>>> print et1
[        [ GEN = 'f'  ] ]
[ CONC = [ NUM = 'pl' ] ]
[        [ PER = 3    ] ]
[                       ]
[ POS  = 'N'            ]
```

Aquesta estructura de trets la podríem haver declarat també de la manera següent:

```
>>> et0=nltk.FeatStruct(PER='3', NUM='pl', GEN='f')
>>> et1=nltk.FeatStruct(POS='N', CONC=et0)
>>> print et1
[        [ GEN = 'f'  ] ]
[ CONC = [ NUM = 'pl' ] ]
[        [ PER = '3'  ] ]
[                       ]
[ POS  = 'N'
```

Les estructures de trets no es fan servir únicament en lingüística; aquestes estructures poden emmagatzemar qualsevol tipus d'informació. En l'exemple següent podem observar una estructura de trets que emmagatzema informació sobre una persona:

```
>>> print nltk.FeatStruct(nom='Maria', edat='25', tel='932198765')
[ edat = '25'        ]
[ nom  = 'Maria'     ]
[ tel  = '932198765' ]
```

Normalment les estructures de trets donen una informació parcial sobre algun objecte. En aquest sentit, podem ordenar les estructures de trets segons siguin més generals o més específiques. Per exemple, de les estructures de trets següents, la *a* és més general que la *b* i aquesta, a la seva vegada, és més general que la *c*.

```
>>> a=nltk.FeatStruct(NUM=125)
>>> b=nltk.FeatStruct(NUM=125,CARRER='Balmes')
>>> c=nltk.FeatStruct(NUM=125,CARRER='Balmes',CIUTAT='Barcelona')
>>> print a
[ NUM = 125 ]
>>> print b
[ CARRER = 'Balmes' ]
[ NUM    = 125      ]
>>> print c
[ CARRER = 'Balmes'    ]
[ CIUTAT = 'Barcelona' ]
[ NUM    = 125         ]
```

Aquest tipus d'ordenació s'anomena *subsumció*: una estructura més general *subsumeix* una estructura menys general.

La fusió d'informació de dues estructures de trets s'anomena *unificació*. Amb NLTK es pot fer aquesta operació amb el mètode *unify()*. Vegem l'exemple següent:

```
>>> et1=nltk.FeatStruct(NUM=125, CARRER='Balmes')
>>> et2=nltk.FeatStruct(CIUTAT='Barcelona')
>>> print et1.unify(et2)
[ CARRER = 'Balmes'    ]
[ CIUTAT = 'Barcelona' ]
[ NUM    = 125         ]
```

Ara bé, no sempre es pot portar a terme aquesta operació. Fixeu-vos en el següent exemple:

```
>>> et1=nltk.FeatStruct(NUM=125, CARRER='Balmes')
>>> et2=nltk.FeatStruct(CARRER='Muntaner')
>>> print et1.unify(et2)
None
```

Les gramàtiques de trets que presentem en aquest apartat funcionen bàsicament amb l'operació d'*unificació*. Fixem-nos en l'exemple següent, en què els trets del determinant i el nom unifiquen i poden constituir un sintagma nominal:

```
>>> det=nltk.FeatStruct(GEN='m',NUM='s')
>>> nom=nltk.FeatStruct(GEN='m',NUM='s')
>>> sn=det.unify(nom)
>>> print sn
[ GEN = 'm' ]
[ NUM = 's' ]
```

En canvi, en l'exemple següent, els trets del determinant i el nom no unifiquen:

```
>>> det=nltk.FeatStruct(GEN='m',NUM='s')
>>> nom=nltk.FeatStruct(GEN='f',NUM='s')
>>> sn=det.unify(nom)
>>> print sn
None
```

Exercici

Amplieu la `gramatica5.fcfg` (i anomeneu-la `gramatica6.fcfg`) perquè pugui analitzar les oracions següents:

- els nois i les noies mengen entrepans de formatge
- els nois i les noies mengen entrepans de formatge i pastissos de xocolata
- el noi llegeix un llibre a casa

4. Les gramàtiques lliures de context probabilístiques

En certes situacions, una gramàtica lliure de context pot donar més d'una possible anàlisi. Fixem-nos en l'oració següent:

jo miro la noia amb el telescopi

L'oració és ambigua des del punt de vista sintàctic, ja que "amb el telescopi" pot complementar "noia" o "miro". Si tenim la gramàtica següent lliure de context (`gramatica7.cfg`) (podeu executar el programa `programa-8-3.py`):

```
O -> SN SV
SN -> Det N
SN -> Det N SP
SN -> PP
SV -> V
SV -> V SN
SV -> V SN SP
SP -> Prep SN
Det -> 'el'|'la'
PP -> 'jo'
N -> 'noia'|'telescopi'
V -> 'miro'
Prep -> 'amb'
```

Ens ofereix les dues anàlisis següents (que també podem observar a les figures 11 i 12):

```
(O
  (SN (PP jo))
  (SV
    (V miro)
    (SN (Det la) (N noia))
    (SP (Prep amb) (SN (Det el) (N telescopi)))))
```

```
(O
  (SN (PP jo))
```

```
(SV
  (V miro)
  (SN (Det la) (N noia))
  (SP (Prep amb) (SN (Det el) (N telescopi)))))
```

Figura 11. Anàlisi sintàctica de "jo miro la noia amb el telescopi"

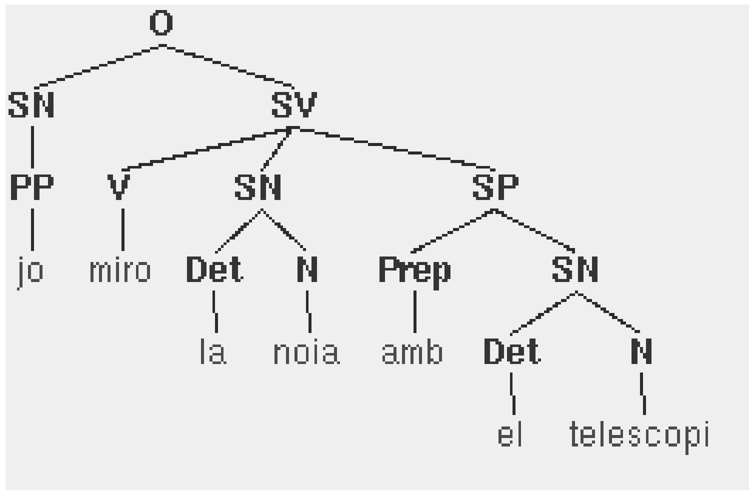
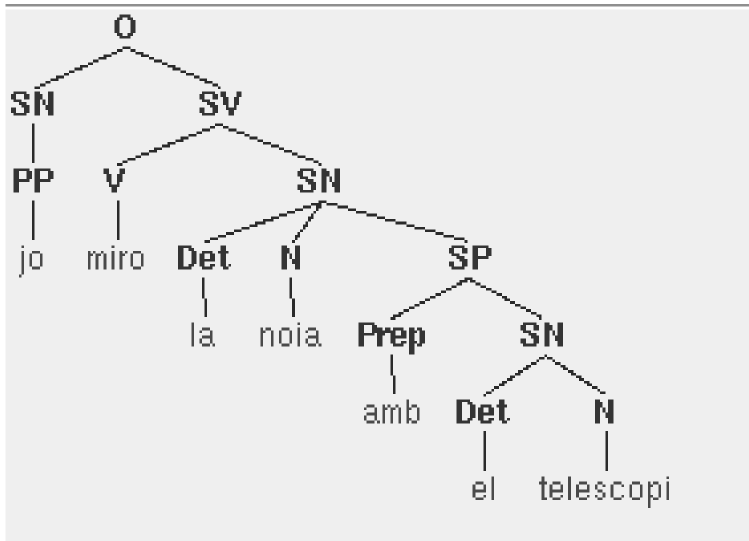


Figura 12. Anàlisi sintàctica de "jo miro la noia amb el telescopi"



A un parlant de la llengua el primer que li ve al cap és que jo miro la noia fent servir un telescopi i no que la noia porta un telescopi. Però en l'oració següent, amb la mateixa estructura sintàctica, "jo miro la noia amb el barret", queda molt clar que la noia porta un barret.

Una gramàtica lliure de context probabilística* és una gramàtica lliure de context amb informació de la probabilitat de cada una de les produccions. Per tal d'assegurar que totes les anàlisis generades per la gramàtica formin una distribució de probabilitats, les PCFG imposen com a restricció que totes les produccions amb una part esquerra donada tinguin una suma de probabilitats igual a 1.

*En anglès, *Probabilistic Context Free Grammar* (PCFG).

A continuació (`programa-8-5.py`) podem observar un exemple de gramàtica lliure de context probabilística:

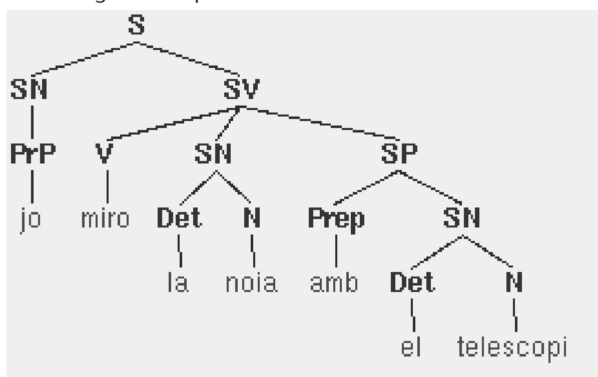
```
import nltk
grammar = nltk.parse_pcfg("""
S  -> SN SV  [1.0]
SN -> PrP  [0.3]
SN -> Det N      [0.5]
SN -> SN SP  [0.2]
SP -> Prep SN    [1.0]
SV -> V    [0.2]
SV -> V SN  [0.4]
SV -> V SN SP  [0.4]
PrP -> 'jo'      [1.0]
Det -> 'la'      [0.5]
Det -> 'el'      [0.5]
        N -> 'noia'      [0.5]
        N -> 'telescopi' [0.5]
        Prep -> 'amb'    [1.0]
        V -> 'miro'      [1.0]
    """)
print grammar
viterbi_parser = nltk.ViterbiParser(grammar)
frase="jo miro la noia amb el telescopi"
tokens = frase.split()
analisi= viterbi_parser.parse(tokens)
print analisi
analisi.draw()
```

Si executem aquest programa obtindrem la sortida següent i també l'anàlisi en forma gràfica (figura 13). Fixeu-vos que també dóna una xifra de probabilitat total ($p = 0{,}001875$).

```
Grammar with 15 productions (start state = S)
    S -> SN SV [1.0]
    SN -> PrP [0.3]
    SN -> Det N [0.5]
    SN -> SN SP [0.2]
    SP -> Prep SN [1.0]
    SV -> V [0.2]
    SV -> V SN [0.4]
    SV -> V SN SP [0.4]
    PrP -> 'jo' [1.0]
    Det -> 'la' [0.5]
    Det -> 'el' [0.5]
```

```
    N -> 'noia' [0.5]
    N -> 'telescopi' [0.5]
    Prep -> 'amb' [1.0]
    V -> 'miro' [1.0]
(S
  (SN (PrP jo))
  (SV
    (V miro)
    (SN (Det la) (N noia))
    (SP (Prep amb) (SN (Det el) (N telescopi))))) (p=0.001875)
```

Figura 13. Anàlisi sintàctica de "jo miro la noia amb el telescopi" amb una gramàtica probabilística



4.1. Inducció de gramàtiques

Com hem vist, les PCFG són com les CFG però amb probabilitats. En l'exemple anterior el que hem fet és simplement especificar aquestes probabilitats en la gramàtica (en realitat ens les hem inventades). En un cas real, d'on podem obtenir aquests valors de probabilitat? El més habitual és estimar aquestes probabilitats a partir de dades d'entrenament, concretament a partir d'arbres s'anàlisi (*parse trees*) o *treebanks*. Un *treebank* és un corpus en què cada oració s'ha anotat amb la seva estructura sintàctica. NLTK ofereix diversos *treebanks*:

- alpino: Alpino Treebank (Dutch)
- cess_cat: CESS-CAT Treebank (Catalan)
- cess_esp: CESS-ESP Treebank (Spanish)
- floresta: Floresta Treebank (Portuguese)
- treebank: Penn Treebank Corpus Sample (English)
- sinica: Sinica Treebank Corpus Sample (Chinese)

En una sessió interactiva podem observar les primeres oracions analitzades, per exemple, del Penn Treebank:

```
>>> print nltk.corpus.treebank.parsed_sents()
[Tree('S', [Tree('NP-SBJ', [Tree('NP', [Tree('NNP', ['Pierre']),
```

```
Tree('NNP', ['Vinken'])]), Tree(',', [',']), Tree('ADJP', [Tree('NP',
[Tree('CD', ['61']), Tree('NNS', ['years'])]),
Tree('JJ', ['old'])]), Tree(',', [','])]), Tree('VP', [Tree('MD', ['will']),
Tree('VP', [Tree('VB', ['join']), Tree('NP', [Tree('DT', ['the']),
Tree('NN', ['board'])]), Tree('PP-CLR', [Tree('IN', ['as']), Tree('NP',
[Tree('DT', ['a']), Tree('JJ', ['nonexecutive']),
Tree('NN', ['director'])])]), Tree('NP-TMP', [Tree('NNP', ['Nov.']),
Tree('CD', ['29'])])])])]), Tree('.', ['.'])]), Tree('S', [Tree('NP-SBJ',
[Tree('NNP', ['Mr.']), Tree('NNP', ['Vinken'])]), Tree('VP',
[Tree('VBZ', ['is']), Tree('NP-PRD', [Tree('NP', [Tree('NN', ['chairman'])]),
Tree('PP', [Tree('IN', ['of']), Tree('NP', [Tree('NP', [Tree('NNP', ['Elsevier']),
Tree('NNP', ['N.V.'])]), Tree(',', [',']), Tree('NP', [Tree('DT', ['the']),
Tree('NNP', ['Dutch']), Tree('VBG', ['publishing']),
Tree('NN', ['group'])])])])])])]), Tree('.', ['.'])]), ...]
```

A partir d'aquest *treebank* podem fer un programa (`programa-8-6.py`) que indueixi una gramàtica i analitzi l'oració "the man saw the girl in the park":

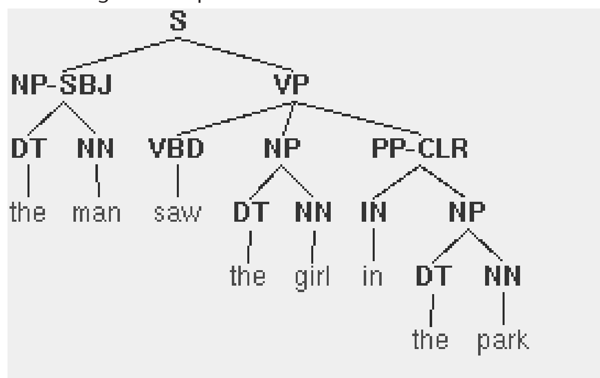
```
import nltk
from itertools import islice
productions = []
S = nltk.Nonterminal('S')
for tree in nltk.corpus.treebank.parsed_sents():
      productions += tree.productions()
grammar = nltk.induce_pcfg(S, productions)
print grammar
viterbi_parser = nltk.ViterbiParser(grammar)
frase="the man saw the girl in the park"
tokens = frase.split()
analisi= viterbi_parser.parse(tokens)
print analisi
analisi.draw()
```

Si executem aquest programa obtindrem la sortida següent (la gramàtica induïda és molt més llarga, però reproduïm només un fragment; si no voleu visualitzar la gramàtica elimineu o comenteu la línia *print grammar*) i l'anàlisi de la figura 14. Tingueu en compte que aquest programa pot trigar una bona estona a executar-se.

```
...
NNP -> 'Solaia' [0.000106269925611]
JJ -> 'proof' [0.000171408981831]
VB -> 'swallow' [0.000391542678152]
VB -> 'parallel' [0.000391542678152]
NNP -> 'NRDC' [0.000106269925611]
```

```
    CD -> '5.92' [0.000282007896221]
    NNP -> 'Frenzy' [0.000425079702444]
    NNS -> 'attacks' [0.000496113775426]
    SBAR -> WHNP-258 S [0.000424628450106]
    JJR -> 'fewer' [0.0157480314961]
    RB -> 'but' [0.000354358610914]
    VP -> VBD PP-LOC-PRD S-ADV [6.89179875948e-05]
    VBG -> 'dominating' [0.000684931506849]
    VBG -> 'failing' [0.0027397260274]
    VP -> VBD ADJP-PRD ADVP-TMP [0.000206753962784]
    VBN -> 'contacted' [0.00140581068416]
    NP-SBJ -> DT JJ NNP NNP [0.000392567390735]
(S
  (NP-SBJ (DT the) (NN man))
  (VP
    (VBD saw)
    (NP (DT the) (NN girl))
    (PP-CLR (IN in) (NP (DT the) (NN park))))) (p=6.93813037949e-22)
```

Figura 14. Anàlisi sintàctica de "the man saw the girl in the park" amb una gramàtica probabilística induïda



NLTK també incorpora un *treebank* per al català (el *cess_cat*). Podem veure algunes oracions analitzades d'aquest *treebank* en una sessió interactiva:

```
>>> print nltk.corpus.cess_cat.parsed_sents()
[Tree('S', [Tree('sno-SUJ', [Tree('espec.ms', [Tree('da0ms0', ['El'])]),
Tree('grup.nom.ms', [Tree('np0000o', ['Tribunal_Suprem']), Tree('sno',
[Tree('grup.nom.ms', [Tree('Fpa', ['-Fpa-']), Tree('np0000o', ['TS']),
Tree('Fpt', ['-Fpt-'])])])])])]), Tree('grup.verb', [Tree('vaip3s0', ['ha']),
Tree('vmp00sm', ['confirmat'])]), Tree('sn-CD', [Tree('espec.fs',
[Tree('da0fs0', ['la'])]), Tree('grup.nom.fs', [Tree('ncfs000', ['condemna']),
Tree('sp', [Tree('prep', [Tree('sps00', ['a'])]),
Tree('sn.co', [Tree('sn', [Tree('espec.mp', [Tree('dn0cp0', ['quatre'])]),
Tree('grup.nom.mp', [Tree('ncmp000', ['anys']), Tree('sp', [Tree('prep',
[Tree('sps00', ["d'"])]), Tree('sn', [Tree('grup.nom.fs',
[Tree('ncfs000', ['inhabilitaci\xf3']), Tree('s.a.fs', [Tree('grup.a.fs',
```

```
[Tree('aq0cs0', ['especial'])])])])])])])])]), Tree('coord', [Tree('cc', ['i'])]),
Tree('sn', [Tree('espec.fs', [Tree('di0fs0', ['una'])]), Tree('grup.nom.fs',
[Tree('ncfs000', ['multa']), Tree('sp', [Tree('prep', [Tree('sps00', ['de'])]),
Tree('snn', [Tree('espec.mp', [Tree('Z', ['3,6'])]),
Tree('grup.nom.mp', [Tree('ncmp000', ['milions']), Tree('sp',
[Tree('prep', [Tree('sps00', ['de'])]), Tree('sn', [Tree('grup.nom.fp',
[Tree('Zm...
```

Amb aquest *treebank* podem induir una gramàtica per al català (`programa-8-7.py`).
En aquest exemple fem servir les primeres 1.000 oracions del treebank.

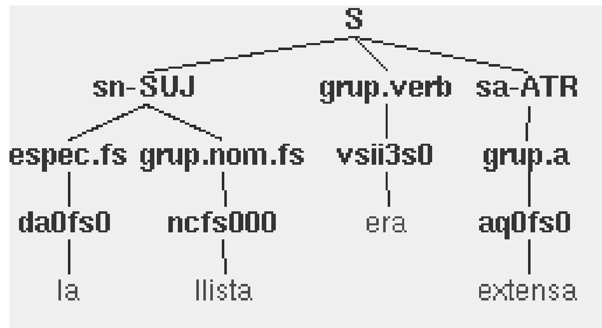
```
import nltk
from itertools import islice
productions = []
S = nltk.Nonterminal('S')
sents=nltk.corpus.cess_cat.parsed_sents();

for tree in sents[0:1000]:
      productions += tree.productions()
grammar = nltk.induce_pcfg(S, productions)
print grammar
viterbi_parser = nltk.ViterbiParser(grammar)
frase="la llista era extensa"
tokens = frase.split()
analisi= viterbi_parser.parse(tokens)
print analisi
analisi.draw()
```

Si executem el programa obtindrem la sortida següent i l'anàlisi de la figura 15:

```
    ....
    vmp00sm -> 'autoinculpat' [0.002331002331]
    grup.nom.fp.1n -> Z [0.363636363636]
    di0cs0 -> 'cada' [0.555555555556]
    S.F.R -> relatiu-SUJ morfema.verbal-PASS grup.verb sadv-CC [0.006]
    W -> '1985' [0.003663003663]
    np0000o -> Çol\xb7legi_d'_Aparelladors_i_Arquitectes_T\xe8cnics_de_Barcelona"
              [0.00133689839572]
    S.F.A-CC -> conj.subord morfema.verbal-IMPERS grup.verb sp-CREG [0.0161290322581]
    S.NF.C -> Fe infinitiu sadv-CC sp-CC Fe [0.00302114803625]
    ncms000 -> 'text' [0.00207576543851]
    Z -> '62.200' [0.00299401197605]
(S
  (sn-SUJ
    (espec.fs (da0fs0 la))
```

```
    (grup.nom.fs (ncfs000 llista)))
  (grup.verb (vsii3s0 era))
  (sa-ATR (grup.a (aq0fs0 extensa)))) (p=2.10954249253e-13)
```

Figura 15. Anàlisi sintàctica de "la llista era extensa" amb una gramàtica probabilística induïda



4.2. Analitzadors per a gramàtiques probabilístiques: l'algorisme de Viterbi

En l'apartat 2 vam veure un seguit d'analitzadors (*parsers*) que permeten fer anàlisis sintàctiques amb gramàtiques lliures de context. En les gramàtiques lliures de context que hem estudiat en aquest apartat hem afegit una informació addicional a cada una de les produccions de la gramàtica: la seva probabilitat. Una determinada oració pot tenir moltes possibles anàlisis segons una gramàtica donada. Si aquesta gramàtica és probabilística, la probabilitat de cada una d'aquestes anàlisis és el producte de les probabilitats de cadascuna de les regles utilitzades per a derivar l'anàlisi.

El càlcul de l'anàlisi més probable pot ser massa llarg si el que es fa és calcular totes les possibles anàlisis i per a cada anàlisi calcula el producte de les seves probabilitats.

Si ens fixem en els programes que hem fet servir en aquest apartat, tots fan servir el ViterbiParser que proporciona l'NLTK. Aquest analitzador fa servir l'algorisme de Viterbi.

Lectures recomanades

Per saber-ne més sobre l'algorisme de Viterbi podeu consultar les obres de Viterbi (1967), Viterbi i Omura (1979) i Forney (1973).

L'algorisme de Viterbi permet trobar les seqüències d'estats més probables en un model ocult de Markov (HMM, *hidden Markov model*) a partir d'una observació, és a dir, obté la seqüència òptima que explica millor la seqüència d'observacions.

Un **model ocult de Markov** és un model estadístic en el qual s'assumeix que el sistema per modelar és un procés de Markov de paràmetres desconeguts. Un **procés** o **cadena de Markov** és una sèrie d'esdeveniments en els quals la probabilitat que ocorri un esdeveniment depèn de l'esdeveniment immediatament anterior.

No entrarem en gaires detalls sobre aquest algorisme i només direm que en comptes de calcular les probabilitats de tots els possibles camins, el que fa és descartar aquells que no tenen possibilitats de ser considerats de màxima versemblança (*maximum likelihood*). Se'n pot trobar una descripció detallada del funcionament a Kenn i Samuelsson (1997).

El ViterbiParser de NLTK és un analitzador ascendent per a gramàtiques lliures de context probabilístiques que fa servir tècniques de programació dinàmica per a trobar l'anàlisi més probable d'un text. L'anàlisi la porta a terme omplint una taula de constituents més probables. Aquesta taula registra la representació en arbre d'anàlisi més probable per a qualsevol fragment donat i valor de node. Concretament conté una entrada per a cada valor d'índex inicial, índex final i valor de node, i registra el subarbre més probable que abraça des de l'índex inicial fins a l'índex final, i que té el valor de node donat.

Aquest analitzador emplena aquesta taula de manera incremental. Comença omplint totes les entrades per constituents que abracen un element de text (és a dir, les entrades per a les quals l'índex final és un més que l'índex inicial). Un cop ha omplert totes les entrades de la taula corresponents als constituents que abracen un element del text, emplena les entrades per als constituents que abracen dos elements del text. Després continua omplint les entrades corresponents a constituents que abracen porcions com més va més grans del text, fins que omple tota la taula.

Per a trobar el constituent més probable per a un fragment i valor de node, l'analitzador té en compte totes les produccions que poden produir aquest valor de node. Per a cada producció, troba tots els fills que de manera col·lectiva cobreixen el fragment de text i tenen els valors de node especificats per la part dreta de la producció. Si la probabilitat de l'arbre format per l'aplicació de la producció al fill és més gran que la probabilitat de l'entrada actual de la taula, llavors la taula s'actualitza amb aquest nou arbre.

Podem executar una demostració d'aquest analitzador fent:

```
>>> nltk.parse.viterbi.demo()

  1: I saw the man with my telescope
     <Grammar with 17 productions>

  2: the boy saw Jack with Bob under the table with a telescope
     <Grammar with 23 productions>

Which demo (1-2)? 1

sent: I saw the man with my telescope
parser: <ViterbiParser for <Grammar with 17 productions>>
```

```
grammar: Grammar with 17 productions (start state = S)
    S -> NP VP [1.0]
    NP -> Det N [0.5]
    NP -> NP PP [0.25]
    NP -> 'John' [0.1]
    NP -> 'I' [0.15]
    Det -> 'the' [0.8]
    Det -> 'my' [0.2]
    N -> 'man' [0.5]
    N -> 'telescope' [0.5]
    VP -> VP PP [0.1]
    VP -> V NP [0.7]
    VP -> V [0.2]
    V -> 'ate' [0.35]
    V -> 'saw' [0.65]
    PP -> P NP [1.0]
    P -> 'with' [0.61]
    P -> 'under' [0.39]
Inserting tokens into the most likely constituents table...
   Insert: |=......| I
   Insert: |.=.....| saw
   Insert: |..=....| the
   Insert: |...=...| man
   Insert: |....=..| with
   Insert: |.....=.| my
   Insert: |......=| telescope
Finding the most likely constituents spanning 1 text elements...
   Insert: |=......| NP -> 'I' [0.15]              0.1500000000
   Insert: |.=.....| V -> 'saw' [0.65]             0.6500000000
   Insert: |.=.....| VP -> V [0.2]                 0.1300000000
   Insert: |..=....| Det -> 'the' [0.8]            0.8000000000
   Insert: |...=...| N -> 'man' [0.5]              0.5000000000
   Insert: |....=..| P -> 'with' [0.61]            0.6100000000
   Insert: |.....=.| Det -> 'my' [0.2]             0.2000000000
   Insert: |......=| N -> 'telescope' [0.5]        0.5000000000
Finding the most likely constituents spanning 2 text elements...
   Insert: |==.....| S -> NP VP [1.0]              0.0195000000
   Insert: |..==...| NP -> Det N [0.5]             0.2000000000
   Insert: |.....==| NP -> Det N [0.5]             0.0500000000
Finding the most likely constituents spanning 3 text elements...
   Insert: |.===...| VP -> V NP [0.7]              0.0910000000
   Insert: |....===| PP -> P NP [1.0]              0.0305000000
Finding the most likely constituents spanning 4 text elements...
   Insert: |====...| S -> NP VP [1.0]              0.0136500000
Finding the most likely constituents spanning 5 text elements...
   Insert: |..=====| NP -> NP PP [0.25]            0.0015250000
Finding the most likely constituents spanning 6 text elements...
```

```
   Insert: |.======| VP -> VP PP [0.1]              0.0002775500
   Insert: |.======| VP -> V NP [0.7]               0.0006938750
  Discard: |.======| VP -> VP PP [0.1]              0.0002775500
Finding the most likely constituents spanning 7 text elements...
   Insert: |=======| S -> NP VP [1.0]               0.0001040812

Time (secs)   # Parses   Average P(parse)
-----------------------------------------
     0.0123          1   0.00010408125000
------------------------------------------
        n/a          1   0.00010408125000

Draw parses (y/n)? n

Print parses (y/n)? y
 (S
  (NP I)
  (VP
    (V saw)
    (NP
      (NP (Det the) (N man))
      (PP (P with) (NP (Det my) (N telescope)))))) [0.00010408125]
```

En primer lloc la versió de demostració ens demana que triem entre una de les dues oracions que proposa. Un cop triada ens mostra la gramàtica lliure de context probabilística amb la qual treballarà. A partir d'aquí comença a omplir la taula de constituents més probables. En primer lloc omple tots els *tokens*, cada un a la seva posició. Posteriorment va omplint aquesta taula amb constituents que abracen 1, 2, 3, 4, 5, 6 i 7 elements del text d'entrada. Un cop fet això mostra els temps, les anàlisis que ha trobat i les seves probabilitats i ens permet dibuixar i imprimir l'anàlisi resultant.

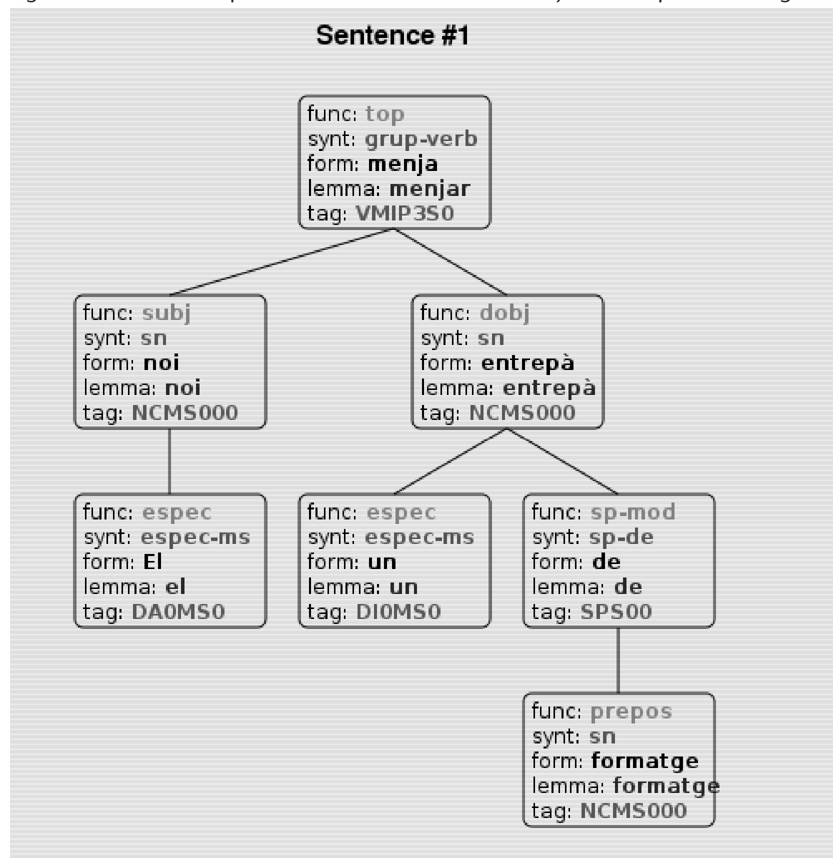
5. Gramàtiques de dependències

Les gramàtiques que hem introduït fins ara analitzen les oracions tenint en compte com les paraules i els grups de paraules es combinen per a formar constituents i com es relacionen aquests constituents. Hi ha una orientació diferent i complementària, les gramàtiques de dependències, que se centren a establir com les paraules es relacionen unes amb les altres.

La **dependència** és una relació asimètrica binària que s'estableix entre un nucli (*head*) i els seus dependents. Normalment es pren com a nucli el verb de l'oració.

A la figura 16 podem observar una anàlisi de dependències de l'oració "El noi menja un entrepà de formatge" feta amb la versió de demostració en línia de l'analitzador Freeling (http://garraf.epsevg.upc.es/freeling/demo.php).

Figura 16. Anàlisi de dependències de l'oració "El noi menja un entrepà de formatge"



5.1. *Treebanks* de dependències de l'NLTK

L'NLTK ofereix alguns *treebanks* anotats per dependències:

- Dependency Treebanks from CoNLL 2007 (Spanish and Basque Subset)
- Dependency Parsed Treebank

En una sessió interactiva podem observar la primera oració del Dependency Parsed Treebank en diversos formats.

En primer lloc podem veure simplement l'oració:

```
>>> print nltk.corpus.dependency_treebank.sents()[0]
['Pierre', 'Vinken', ',', '61', 'years', 'old', ',', 'will', 'join', 'the', 'board',
'as', 'a', 'nonexecutive', 'director', 'Nov.', '29', '.']
>>>
```

També podem observar tota la informació que conté el *treebank* de la primera oració:

```
>>> print nltk.corpus.dependency_treebank.parsed_sents()[0]
[{'address': 0, 'deps': [8], 'rel': 'TOP', 'tag': 'TOP', 'word': None},
 {'address': 1,
  'deps': [],
  'head': 2,
  'rel': '',
  'tag': 'NNP',
  'word': 'Pierre'},
 {'address': 2,
  'deps': [1, 3, 6, 7],
  'head': 8,
  'rel': '',
  'tag': 'NNP',
  'word': 'Vinken'},
 {'address': 3, 'deps': [], 'head': 2, 'rel': '', 'tag': ',', 'word': ','},
 {'address': 4, 'deps': [], 'head': 5, 'rel': '', 'tag': 'CD', 'word': '61'},
 {'address': 5,
  'deps': [4],
  'head': 6,
  'rel': '',
  'tag': 'NNS',
  'word': 'years'},
 {'address': 6, 'deps': [5], 'head': 2, 'rel': '', 'tag': 'JJ', 'word': 'old'},
 {'address': 7, 'deps': [], 'head': 2, 'rel': '', 'tag': ',', 'word': ','},
 {'address': 8,
```

```
 'deps': [2, 9, 18],
 'head': 0,
 'rel': '',
 'tag': 'MD',
 'word': 'will'},
{'address': 9,
 'deps': [11, 12, 16],
 'head': 8,
 'rel': '',
 'tag': 'VB',
 'word': 'join'},
{'address': 10,
 'deps': [],
 'head': 11,
 'rel': '',
 'tag': 'DT',
 'word': 'the'},
{'address': 11,
 'deps': [10],
 'head': 9,
 'rel': '',
 'tag': 'NN',
 'word': 'board'},
{'address': 12,
 'deps': [15],
 'head': 9,
 'rel': '',
 'tag': 'IN',
 'word': 'as'},
{'address': 13, 'deps': [], 'head': 15, 'rel': '', 'tag': 'DT', 'word': 'a'},
{'address': 14,
 'deps': [],
 'head': 15,
 'rel': '',
 'tag': 'JJ',
 'word': 'nonexecutive'},
{'address': 15,
 'deps': [13, 14],
 'head': 12,
 'rel': '',
 'tag': 'NN',
 'word': 'director'},
{'address': 16,
 'deps': [17],
 'head': 9,
 'rel': '',
 'tag': 'NNP',
```

```
  'word': 'Nov.'},
 {'address': 17, 'deps': [], 'head': 16, 'rel': '', 'tag': 'CD', 'word': '29'},
 {'address': 18, 'deps': [], 'head': 8, 'rel': '', 'tag': '.', 'word': '.'}]
```

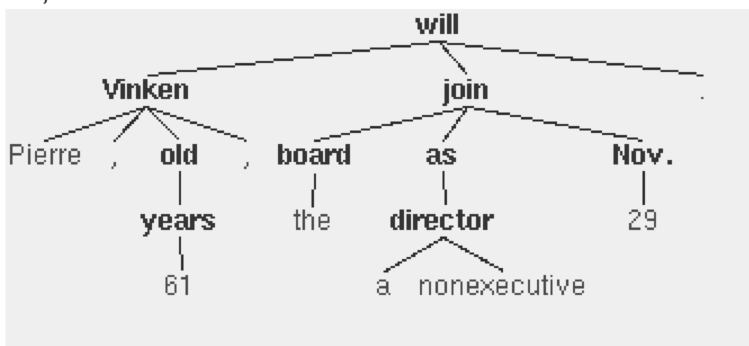
O bé una anàlisi exclusivament de dependències:

```
>>> print nltk.corpus.dependency_treebank.parsed_sents()[0].tree()
(will
  (Vinken Pierre , (old (years 61)) ,)
  (join (board the) (as (director a nonexecutive)) (Nov. 29))
  .)
```

També la podem veure en forma de gràfic a la figura 17.

```
>>> print nltk.corpus.dependency_treebank.parsed_sents()[0].tree().draw()
```

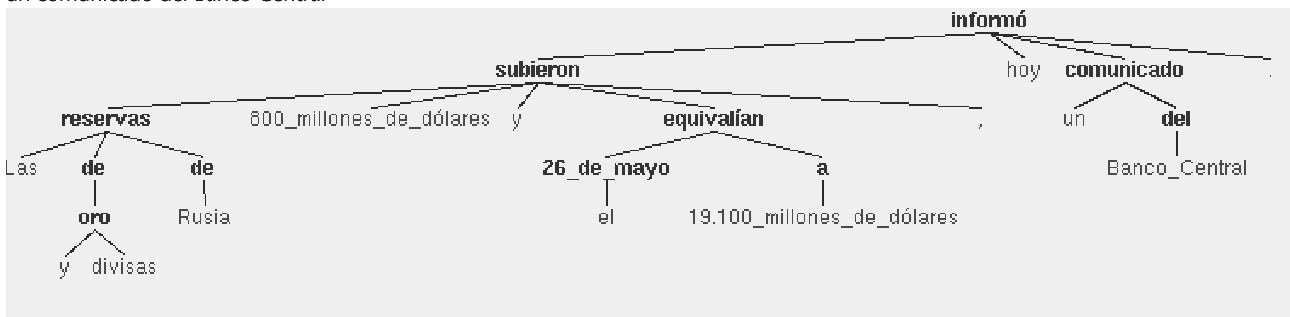
Figura 17. Anàlisi de dependències de l'oració "Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29."



També podem veure (figura 18) una anàlisi de dependències per al castellà de l'oració "Las reservas de oro y divisas de Rusia subieron 80 millones de dólares, informó hoy un comunicado del Banco Central":

```
>>> print nltk.corpus.conll2007.parsed_sents()[0].tree().draw()
```

Figura 18. Anàlisi de dependències de l'oració "Las reservas de oro y divisas de Rusia subieron 80 millones de dólares, informó hoy un comunicado del Banco Central"



Resum

En aquest mòdul hem après a desenvolupar gramàtiques lliures de context amb NLTK i hem estudiat alguns algorismes d'anàlisi (o *parsers*) que es poden fer servir amb aquest tipus de gramàtiques. Hem vist també com es poden induir gramàtiques probabilístiques a partir de corpus analitzats sintàcticament (*treebanks*) i l'algorisme de Viterbi, que ens permet desenvolupar analitzadors per a gramàtiques lliures de context probabilístiques. Finalment, hem vist un altre enfocament d'anàlisi, les gramàtiques de dependències, que se centren a establir com les paraules es relacionen unes amb les altres.

Bibliografia

Carroll, J. (2003). *Parsing*, capítol 12, (pàgs. 233–248). Oxford University Press.

Forney, G. (1973). «The Viterbi Algorithm». A: *Proc. IEEE*, volum 61(3): pàgs. 268–278.

Jurafsky, D. i Martin, J. H. (2000). *Parsing with Context-Free Grammars*, capítol 10, (pàgs. 357–394). Prentice-Hall.

Kenn, B. i Samuelsson, C. (1997). «The Linguist Guide to Statistics: Don't Panic».

Slocum, J. (1981). «A practical comparision of parsing strategies». A: «Proceedings of the 19th annual meeting on Association for Computational Linguistics», (pàgs. 1–6). Morristown, NJ, USA: Association for Computational Linguistics.

Viterbi, A. (1967). «Error Bounds for Convolutional Codes and and Asymptotically Optimum Decoding Algorithm». A: *IEE Transactions on Information Theory*.

Viterbi, A. i Omura, J. (1979). *Principles of Digital Communication and Coding*. New York: McGraw-Hill Book Company.