Implementació de l'algorisme CYK

Àlex Miquel Casanovas Cordero, 23843979V
Víctor Molina Díez, 23927471L

PAA - Maig 2023

1 Introducció

L'objectiu d'aquest treball és implementar en Python l'algoritme de CKY. Aquest és un algoritme de parsing per Gramàtiques Lliures de Context (Context-free Grammars). Donada una gramàtica i una paraula, l'algoritme ha de retornar un booleà corresponent a si la paraula forma part de la gramàtica (True) o no (False). Aquest algoritme es pot implementar amb *Programació Dinàmica*, d'aquesta forma aplicarem els conceptes apresos a PAA i ens ajudarà a consolidar alguns conceptes de l'assignatura PLH. A part de la implementació del CKY, també implementarem la versió probabilística i una funció capaç de transformar una CFG a CNF. A més a més, i com a treball addicional, hem volgut implementar un codi que mostri l'arbre sintàctic que genera una determinada gramàtica per a una certa paraula acceptada per la mateixa.

1.1 Fitxers entregats

Aquesta carpeta conté diversos fitxers de codi i de prova:

- Codi:
  - **main.py:** Aquest codi serveix per realitzar les proves
  - **functions.py:** Conté la funció per llegir la gramàtica, la classe CKY i la classe CKYProb
  - **FNC.py:** Conté la classe FNCTransform per passar una CFG a FNC.
- Jocs de Prova:
  - **CYK_0.txt**
  - **CYK_1.txt**
  - **CYK_2.txt**
  - **CYK_3.txt**
  - **CYK_4.txt**
  - **CYK_5.txt**
  - **CYK_6.txt**
  - **CYK_7.txt**
  - **CYK_0_Prob.txt**
  - **CYK_1_Prob.txt**
  - **CYK_2_Prob.txt**

2 CKY

Abans d'implementar l'algorisme de CKY, necessitem entendre com funciona aquest algorisme. El CKY retorna si una paraula forma part o no d'una CFG (Gramàtica lliure de Context). Primer, definirem a que ens referim amb una CFG:

Una CFG és una tupla G = ⟨N, Σ, R, S⟩. On:

- N: Conjunt de símbols no terminals.
- Σ: Conjunt de símbols terminals.
- R: Conjunt de regles de la forma $X \rightarrow Y_1,Y_2,...,Y_s$. Amb: $X \in$ N; $Y_s \in$ (N ∪ Σ), $\forall s$, $s > 0$. Anomenem *RHS(s)* = X i *LHS(s)* = $Y_1,Y_2,...,Y_s$.
- S: Símbol d'inici. S ∈ N.

Veient això, ja tenim una idea del format en que li hem de passar la gramàtica al nostre algoritme. No obstant, aquesta gramàtica ha d'estar en Forma Normal de Chomsky (FNC), com veurem en profunditat en l'*Apartat 4*. Els motius d'aquesta assumpció són els següents:

- **Evitar ambigüitats:** La CNF pot evitar algunes ambigüitats d'una CFG. No obstant, una CFG pot ser intrinsicament ambigua i generar diversos *parsing trees*.
- **Posar un límit en la complexitat:** El CKY té una complexitat de $O(n^3|G|)$ gràcies a que en estar en FNC, la cerca de les regles és més eficient.
- **L'arbre sintàctic serà binari:** Com una FNC només pot tenir dos no terminals per regla, l'arbre resultant sempre serà binari. Cosa que fa més fàcil recorre'l en cas que es necessiti.

2.1 Llegir la gramàtica

El primer en que ens centrarem és en com passar-li la gramàtica al CKY. El primer que necessitem és llegir la gramàtica del nostre arxiu de text. Per això hem creat una funció anomenada `read_grammar()`. Aquesta funció rep un únic argument, el nom del fitxer .txt que conté la nostra gramàtica. La gramàtica del fitxer ha de seguir el següent format:

```
S → NP VP
NP → D N | N N | time | flies | arrow
VP → V NP | V ADVP
ADVP → ADV NP
N → time | flies | arrow
D → an
ADV → like
V → flies | like
```

De la forma en que està escrita la gramàtica en el fitxer, som capaços d'extreure tota la informació que necessitem per passar-li a l'algoritme:

- Les paraules minúscules representen símbols no terminals.
- Les paraules majúscules representen símbols terminals.

- La RHS(1) sempre és el símbol d'inici.
- Els "|" separen les diferents regles que genera un símbol no terminal.
- Els diferents símbols en una regla es separen amb espais (Per escriure símbols de més d'una lletra)
- **Extra:**
  - Si es vol simbolitzar la paraula buida, s'utilitza el "#" i es considerarà terminal.
  - En el cas de tenir una PCFG la probabilitat s'escriura com un símbol més al final de cada regla. Si la probabilitat és 1, s'ha d'escriure com 1.0.

Sabent el format de l'arxiu de text i la informació que ens dona, la funció `read_grammar()` ens retorna la gramàtica en forma de tupla (N, Σ, R, S) tal que:

- N és un **set()**
- Σ és un **set()**
- R és un **dict(list(list()))**
- S és una **string**

2.2 L'algoritme

Un cop tenim la nostra gramàtica com una variable de `Python`, podem començar a crear l'algoritme. Per implementar l'algoritme hem decidit utilitzar les classes de *Python* ja que, en el nostre cas, des de que ens van introduir la *Programació Orientada a Objectes* a PAA no les hem utilitzat gaire. Després de realitzar aquesta pràctica hem comprovat com de còmode i ordenat és treballar amb *POO* i ho començarem a utilitzar bastant més seguit.

Per implementar l'algoritme hem creat una classe `CKY` la qual rep un argument, la gramàtica. La inicialització de la classe comprova que la gramàtica no sigui probabilística, després descompon la gramàtica en els seus atributs:

```
self.noterminals = Grammar[0]
self.terminals = Grammar[1]
self.rules = Grammar[2]
self.start = Grammar[3]
```

Un cop creada la instància de la classe amb la nostra gramàtica disposem del mètode `check_word(word)` que comprova si la paraula forma part de la gramàtica. A part de la paraula, aquest mètode també consta de l'argument *phrase* que per defecte es False. Tal com hem implementat l'algoritme, podem rebre tant frases (*time flies like an arrow*) que s'han d'analitzar paraula per paraula; com paraules que s'han d'analitzar caràcter per caràcter (*aabbab*). Al passar aquest argument, distingim aquests dos casos, ja que si no sorgiria la possibilitat de passar una frase d'una sola paraula (*flies*) i que l'analitzes lletra per lletra.

Per comprovar si la paraula forma part de la gramàtica, aquest mètode crida internament a un mètode privat anomenat `_create_chart()`. Aquest mètode és l'encarregat de crear la taula fent ús de *Programació Dinàmica*. Un cop generada tota la taula, comprova si el símbol d'inici es troba a l'última casella de la taula. Si hi es retorna True, si no, retorna False.

```
if self.start not in self.chart[0,len(word)-1]:
        return False
    else:
        return True
```

Un cop vist com comprovem si una paraula es troba a la gramàtica, anem a endinsar-nos en el mètode `_create_chart()`, l'encarregat de crear la taula i on s'aplica l'algoritme *CKY* amb *Programació Dinàmica*.

2.3 Creació de la Taula amb Programació Dinàmica

El primer que fem quan cridem al mètode `_create_chart()` és inicialitzar la taula. Per fer-ho hem creat un altre mètode privat `_init_chart()` encarregat d'inicialitzar les caselles (i,i) amb les regles pertinents a cada paraula i de la frase. Per comoditat, hem utilitzat un diccionari per emmagatzemar la taula.

Per trobar les regles associades a un símbol, també hem creat un mètode, `_search_rules()`, que donat un conjunt de símbols busca i retorna un diccionari amb les regles que retornen cada símbol. En cas que una regla surti més d'un cop, se li afegeix un número per distingir-la.

```
if key in result:
    result[f"{key}{count[key]}"] = (symbol)
    count[key] += 1
else:
    result[key] = (symbol)
    count[key] = 1
```

Un cop inicialitzada la taula, es comença a executar l'algoritme. Aquest, anirà recorrent la taula nivell a nivell i columna per columna. Per omplir una cel·la haurem de fer totes les possibles combinacions que portin a aquella cel·la seguint l'algoritme de CKY. El nombre de cel·les per fer la combinació augmenta en funció del nivell on es trobi la cel·la que volem omplir. La fórmula per saber quines cel·les mirar és:

$$\forall k \in \{1, \ldots, nivell\} \quad \text{Cel·la1} = \text{taula}[i, i+k], \quad \text{Cel·la2} = \text{taula}[i+1+k, j]^1$$

Per calcular totes les regles que surten de la combinació de dues cel·les hem creat un altre mètode anomenat `_combine_cells()` que rep com a argument les posicions de les cel·les que ha de combinar, sent i,j la posició de la cel·la 1 i k,l la de la cel·la 2.

Un cop creades totes les possibles combinacions, cridem al mètode `_search_rules()` que com hem explicat, buscarà totes les possibles regles que satisfacin les diferents combinacions. A més al trobar una possible regla guardarà, no només el RHS, sino també el LHS, codificant cada símbol amb les caselles d'on prové. D'aquesta manera, aconseguim evitar ambigüitats i a més utilitzarem aquesta informació per fer la traça de l'arbre, com comentarem en l'*Apartat 5*.

Aquesta codificació és realitza en el mètode de `_combine_cells()` i es coficica de la següent manera:

$$X\_i, j$$

On X representa la regla que estem referenciant, i i, j la posició de la taula.

[1] On i representa les columnes de la matriu i $j = nivell + i$

A més cal recalcar en els separadors. Ja que hem cregut convenient afegir-los per evitar ambigüitats. La "," l'afegim per separar les coordenades, ja les dimensions de la taula fossin de més d'un dígit podria haver ambigüitats (111 és 1,11 o 11,1?). Pel que fa el "_" l'hem utilitzat per un motiu similar. Ja que una casella pot guardar més d'una regla amb el mateix símbol no terminal. I per distingir-les afegim un número al darrere, com hem vist abans. Si no existís el "_" aquest número podria confondre's amb les coordenades de la regla (X11,1 és X_11,1 o X1_1,1?).
Un cop omplerta la taula, el mètode `check_word` ja pot comprovar si la paraula es troba o no a la gramàtica mirant l'última posició de la taula, que seria `taula[0,len(word)-1]`

2.4 Jocs de prova: Exemple CKY

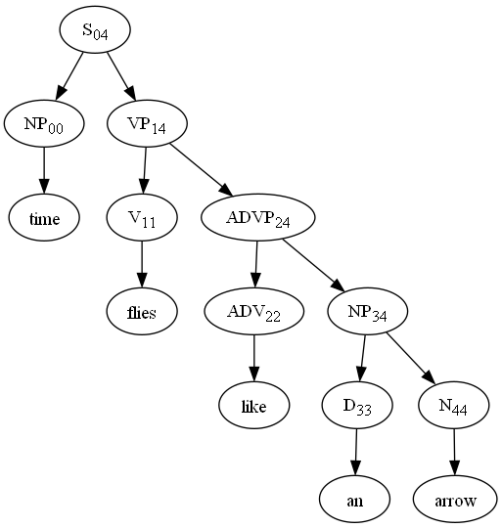
Per acabar amb aquest apartat, posarem un exemple dels jocs de prova on es veu el funcionament d'aquest algoritme. La gramàtica que utilitzarem és la següent:

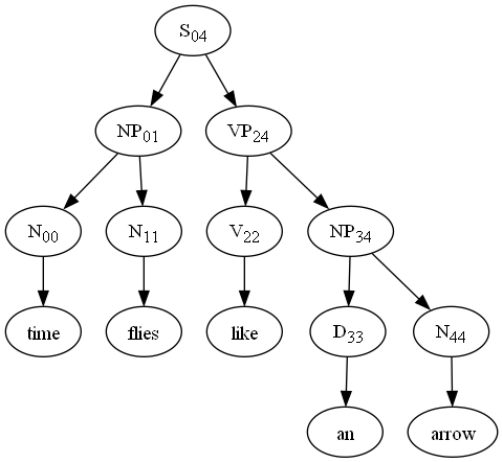
```
S → NP VP
NP → D N | N N | time | flies | arrow
VP → V NP | V ADVP
ADVP → ADV NP
N → time | flies | arrow
D → an
ADV → like
V → flies | like
```

Si provem la següent frase *"time flies like an arrow"* veurem que l'algoritme retorna que la frase es troba a la gramàtica, no obstant, si mirem la taula generada podem veure un dels punts febles d'aquest algoritme:

```
{(0, 0): {'NP': 'time', 'N': 'time'}, (1, 1): {'NP': 'flies', 'N':
    'flies', 'V': 'flies'}, (2, 2): {'ADV': 'like', 'V': 'like'}, (3,
    3): {'D': 'an'}, (4, 4): {'NP': 'arrow', 'N': 'arrow'}, (0, 1):
    {'NP': 'N_0,0 N_1,1'}, (1, 2): {}, (2, 3): {}, (3, 4): {'NP':
    'D_3,3 N_4,4'}, (0, 2): {}, (1, 3): {}, (2, 4): {'VP': 'V_2,2
    NP_3,4', 'ADVP': 'ADV_2,2 NP_3,4'}, (0, 3): {}, (1, 4): {'S':
    'NP_1,1 VP_2,4', 'VP': 'V_1,1 ADVP_2,4'}, (0, 4): {'S': 'NP_0,0
    VP_1,4', 'S1': 'NP_0,1 VP_2,4'}}
```

Com podem veure el símbol inicial es troba en la casella (0,4) de la nostra taula. No obstant si ens fixem en les regles d'aquesta casella, hi ha dues regles que tenen el símbol inicial. Això és degut a que aquesta gramàtica pot donar lloc a ambigüitats, i generar més d'un parsing tree correcte:





Com podem observar, hi ha dos arbres de parsing diferents. Això ens fa preguntar, quin dels dos arbres és més correcte?

En aquest cas és fàcil de dir a simple vista. Té molt més sentit dir *"El temps vola com una fletxa"* que dir *"A les mosques temporals els agrada una fletxa"*. No obstant en casos més complexos pot no ser tan evident.

És per això que es van decidir introduir probabilitats a les regles, creant el CKY probabilístic. A continuació farem la seva implementació i comentarem els resultats amb aquest mateix exemple per veure el seu funcionament.

3 CKY Probabilístic

Un cop implementat el CKY, implementar el probabilístic no va ser gaire complicat. De fet, compartien algunes de les funcions, per això hem estat capaços de crear la classe `CKYProb(CKY)` com una subclasse de CKY. D'aquesta manera, aprofitem un dels potencials de la *Programació Orientada a Objectes* que és l'herència d'atributs. Per això, només comentarem aquelles funcions que hem hagut de canviar per fer funcionar l'algoritme probabilístic.

El CKY probabilístic és una variant de l'algorisme en que es té en compte la probabilitat de que es doni una certa regla. Això evita les ambigüitats ja que dels possibles arbres de parsing resultants podem considerar com a correcte el més probable. La condició per posar les probabilitats, és que la suma de probabilitats de totes les regles formades per el mateix símbol no terminal han de sumar 1.

Abans de començar, cal fer un incís sobre com hem implementat l'emmagatzematge de les probabilitats de les regles. I és que hem decidit no complicar-nos gaire. Per això, l'hem guardat al final de la llista de RHS de la regla. Per tant, sempre que vulguem consultar la probabilitat mirarem la posició [-1] d'aquesta llista.

L'algorisme en si funciona de la mateixa manera. La diferència és que quan omplim una nova cel·la li hem d'assignar una probabilitat. Aquesta probabilitat serà la probabilitat acumulada de totes les regles anteriors:

Donada una regla $r \in R$ la probabilitat d'aquesta regla en una casella i, j és:

$$P(r[i,j]) = P(r) \cdot \prod_{k \in RHS(r)} P(k) \tag{1}$$

On $P(k)$ és la probabilitat que té k en la casella d'on prové.

El primer que hem hagut de modificar és el `__init__` ja que ha de comprovar que la gramàtica sigui probabilística. Això ho fem tenint en compte com hem emmagatzemat les probabilitats. Per tant, per comprovar-ho, hem de mirar que l'últim element de la llista de RHS d'alguna de les regles contingui un ".", que denota que és una probabilitat.

La següent funció aa modificar ha estat `check_word`. La modificació d'aquesta funció és molt subtil. El que hem canviat és el que retorna la funció. Ja que tenint el CKY probabilístic no té gaire sentit retorna Cert o Fals. Per tant, en cas que la paraula formi part de la gramàtica, el que farem és retornar la regla de la casella `taula[0,len(word)-1]` que contingui el símbol d'inici i tingui la probabilitat més alta. A més guardarem aquesta informació en un nou atribut anomenat `self.best_sol`. Això ens serà útil per dibuixar sempre l'arbre de parsing més probable, com veurem en l'*Apartat 5*.

A part d'aquests dos mètodes hem hagut de canviar un altre, que és l'encarregat d'anar guardant les probabilitats de cada regla. El mètode que hem hagut de modificar és `_search_rules()`. Ja que aquest cop no ens serveix amb guardar la regla, sinó que hem de mirar les probabilitats dels RHS de la regla. Per això hem hagut de descodificar els símbols, per extreure de quines caselles provenien i quina regla era exactament. Un cop descodificat podem calcular la nova probabilitat multiplicant les tres probabilitats com hem comentat abans.

```
if len(n) != 1:
    son1 = n[0].split("_")
    son2 = n[1].split("_")
    i,j=son1[1].split(',')
    k,l=son2[1].split(',')
    prob1 = self.chart[int(i),int(j)][son1[0]][1]
    prob2 = self.chart[int(k),int(l)][son2[0]][1]
    prob_total = float(value[-1])*prob1*prob2
else:
    prob_total = float(value[-1])
if key in result:
    result[f"{key}{count[key]}"] = (symbol,prob_total)
    count[key] += 1
else:
    result[key] = (symbol,prob_total)
    count[key] = 1
```

Com podem veure, la probabilitat es guarda junt amb el símbol com una tupla.

3.1 Jocs de prova: Exemple CKY Probabilístic

Un cop explicada la implementació del CKY probabilístic, tornarem a provar el joc de prova anterior per veure com afecta aquest algorisme a la solució. La gramàtica amb probabilitats que

utilitzarem és la següent:

```
S → NP VP 1.0
NP → D N 0.4 | N N 0.25 | time 0.14 | flies 0.07 | arrow 0.14
VP → V NP 0.6 | V ADVP 0.4
ADVP → ADV NP 1.0
N → time 0.4 | flies 0.2 | arrow 0.4
D → an 1.0
ADV → like 1.0
V → flies 0.5 | like 0.5
```

Si provem la frase *"time flies like an arrow"* veurem que l'algoritme no només retorna que la frase es troba a la gramàtica sinó que a més ens donarà el parsing tree més probable:

```
{(0, 0): {'NP': ('time', 0.14), 'N': ('time', 0.4)}, (1, 1): {'NP':
    ('flies', 0.07), 'N': ('flies', 0.2), 'V': ('flies', 0.5)}, (2, 2):
    {'ADV': ('like', 1.0), 'V': ('like', 0.5)}, (3, 3): {'D': ('an',
    1.0)}, (4, 4): {'NP': ('arrow', 0.14), 'N': ('arrow', 0.4)}, (0,
    1): {'NP': ('N_0,0 N_1,1', 0.020000000000000004)}, (1, 2): {}, (2,
    3): {}, (3, 4): {'NP': ('D_3,3 N_4,4', 0.16000000000000003)}, (0,
    2): {}, (1, 3): {}, (2, 4): {'VP': ('V_2,2 NP_3,4',
    0.04800000000000001), 'ADVP': ('ADV_2,2 NP_3,4',
    0.16000000000000003)}, (0, 3): {}, (1, 4): {'S': ('NP_1,1 VP_2,4',
    0.003360000000000001), 'VP': ('V_1,1 ADVP_2,4',
    0.03200000000000001)}, (0, 4): {'S': ('NP_0,0 VP_1,4',
    0.004480000000000001), 'S1': ('NP_0,1 VP_2,4',
    0.0009600000000000003)}}
```

Com podem veure les probabilitats van evolucionant a mida que avança la taula fins a arribar a la casella (0,4) on surten els dos mateixos arbres que en el joc de proves anterior però un amb més probabilitat que l'altre. Si mirem l'arbre més probable, veiem el següent:

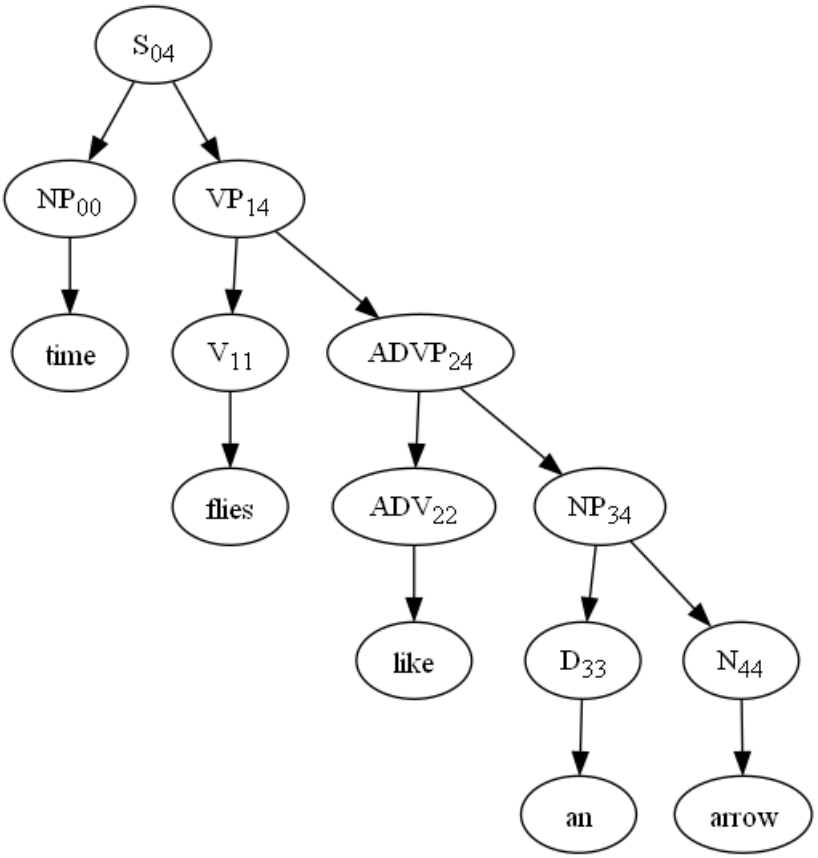


Figure 1: Arbre sintàctic més probable

Com podem observar, l'arbre més probable ha estat el de *"El temps vola com una fletxa"* que és el que tenia més sentit dels dos. D'aquesta manera demostrem que l'algorisme funciona i a més millora molt el parsing de paraules ja que evita les ambigüitats.

4 Transformació d'una gramàtica a FNC

En aquesta secció exposarem de quina manera hem implementat el codi que realitza la conversió de qualsevol CFG/PCFG a una CFG/PCFG en Forma Normal de Chomsky (FNC a partir d'ara). Aquest és i ha estat un exercici molt interessant a dur a terme ja que complementa l'aprenentatge i la consolidació dels coneixements necessaris per a aplicar el CYK alhora que completa la immersió en el món de les gramàtiques no contextuals tot requerint un nivell de programació elevat. Sent conscients de que la part opcional proposada en la pràctica és únicament la d'implementar un transformador per a gramàtiques no probabilístiques, hem decidit realitzar un transformador també per a gramàtiques probabilístiques donat que ens ha semblat força interessant.

Cal fer un incís en el fet que la implementació d'aquest programa amb el llenguatge de programació *Python* ha estat realitzat seguint el paradigma de *Programació Orientada a Objectes* ja que, com veurem més endavant, suposa un increment substancial en l'eficiència del programa.

Forma Normal de Chomsky

Abans d'entrar en detalls sobre la implementació realitzada d'aquest transformador gramatical explicarem de manera breu què significa tenir una CFG en Forma Normal de Chomsky, quines assumpcions realitzem i quines conseqüències tenen aquestes.

Definició 1 *Una CFG és una tupla* $G = \langle N, \Sigma, R, S \rangle$. *On:*

- N*: Conjunt de símbols no terminals.*
- Σ*: Conjunt de símbols terminals.*
- R*: Conjunt de regles de la forma* $\mathrm{X} \rightarrow Y_1,Y_2,...,Y_s$. *Amb:* $\mathrm{X} \in N$; $Y_s \in (N \cup \Sigma)$, $\forall s$, $s > 0$. *Anomenem* $\mathrm{RHS(s)} = \mathrm{X}$ *i* $\mathrm{LHS(s)} = Y_1,Y_2,...,Y_s$.
- S*: Símbol d'inici.* $S \in N$.

Definició 2 *Una PCFG és una tupla* $G_P = \langle G,p \rangle$. *On:*

- G*: Tupla equivalent a la d'una CFG.*
- p*: Conjunt de valors probabilístics associats a cadascuna de les regles* $r \in R$*, de manera que:*

$$\sum_{r \in R \mid LHS(r)=X} p(r) = 1 \qquad \forall X \in N \tag{2}$$

Definició 3 *Una CFG o PCFG està expressada en Forma Normal de Chomsky (FNC) si, i només si, representa una tupla* G_{FNC} *amb els elements descrits anteriorment i ademés:*

- R*: Les regles que determinen el llenguatge de la gramàtica són del tipus:*
  - $\mathrm{X} \rightarrow Y_s,Y_t \quad p(r_1) \qquad \mathrm{X},Y_s,Y_t \in N$; $p(r_1) \in p$; $Y_s,Y_t \neq S$.
  - $\mathrm{X} \rightarrow a \quad p(r_2) \qquad a \in \Sigma$; $p(r_2) \in p$; $a \neq \epsilon$.

Teorema 1 *Qualsevol CFG o PCFG que no generi la seqüència o paraula buida pot ser expressada en FNC.*

Teorema 2 *El llengautge generat per les regles i símbols d'una determinada gramàtica és igual al llenguatge generat per aquella mateixa gramàtica en FNC.*

Així doncs, qualsevol regla de la gramàtica donada que no compleixi les condicions demandades per una FNC, tal i com descriu la Definició 3, haurà de ser transformada d'alguna manera per tal que les acabi complint i que el resultat sigui equivalent a la gramàtica original.

És per això que hi ha establides una sèrie de transformacions per a tots els casos diferents en els quals alguna de les premises necessàries per a què una CFG/PCFG estigui en FNC es viola. Totes les transformacions s'apliquen en els mateixos casos i de la mateixa manera tant per a CFG com per a PCFG, afegint-hi en aquest últim una lògica de transformació de les probabilitats associades a cadascuna de les regles que en breus exposarem. A continuació, descriurem tots aquests casos i, conjuntament, explicarem de quina manera hem implementat cadascuna d'aquestes transformacions.

Transformació START

Aquest tipus de transformacions afecten a aquelles regles r de la gramàtica tals que $S \in RHS(r)$, ja que violen el primer punt de la Definició 3. En aquest cas, la transformació consisteix en afegir un nou símbol d'inici S_o i una nova regla de la manera:

- $S_o \rightarrow S \quad p(r_1) = 1$
- $S_o \in N$

D'aquesta manera, el nou símbol d'inici no queda a la dreta de cap regla, tal i com es requereix en una gramàtica en FNC. Cal parar atenció al fet que, si ens trobéssim en el cas d'estar transformant una PCFG, l'única acció addicional que hauriem de fer seria afegir una probabilitat de 1 a la nova regla creada ja que al ser afegida de manera externa sempre es complirà.
Pel que fa al nostre codi, tal i com es pot observar en l'annex A, el que realitzem és una simple cerca lineal a través de les diferents regles de la gramàtica emmagatzemades en la variable de classe **self.r**. Si es dona el cas de que el símbol d'inici **self.s** està en la part dreta d'alguna regla, s'afegeix al nou conjunt de regles **n_r** la nova regla i afegim el nou símbol a la variable que emmagatzema els no terminals **self.n_t**. Veiem que hi ha un **if** que diferencia els casos en els que **self._k** és 1 (PCFG) o és 0 (CFG). En el primer cas, a la llista s'afegeix un **1.0** en la última posició indicant aquesta probabilitat.

Transformació DEL

La transformació que explicarem a continuació és probablement la que més ens ha costat d'implementar de totes les que presentarem. Aquesta s'aplica sobre aquelles regles que són nul·lables, és a dir que són de l'estil:

- $X \rightarrow \epsilon \quad p(r) \qquad X, \epsilon \in N; X \neq S; \epsilon = paraula\ buida;$

El que es fa en aquesta situació és eliminar la regla nul·lable en qüestió i, per cada regla que contingui X a la seva *RHS* afegir-ne/derivar-ne tantes regles noves com possibles combinacions diferents de RHS traient alguna aparició de X en la regla. Per exemple, si inicialment la nostra CFG/PCFG té les següents regles:

- $X \rightarrow \epsilon \quad p(r_1) \mid D \quad p(r_2)$

- $A \rightarrow B, X, C, X \quad p(r_3)$

Aleshores, si apliquem la transformació DEL, passem a tenir-ne les següents:

- $X \rightarrow D \quad p(r_2)$
- $A \rightarrow B, X, C, X \quad p(r_3) \mid B, C, X \;\; p(r_3) * p(r_1) \mid B, X, C \;\; p(r_3) * p(r_1) \mid B, C \;\; p(r_3) * p(r_1)$

La dificultat en aquesta transformació recau en el fet que en el procés d'addició de noves regles per a transformar una nul·lable en pot tornar a resultar alguna altra nul·lable que requerirà transformació. En aquest cas el que hem implementat és un contenidor que contingui totes les regles que quedin per nul·lar i anar afegint sempre que sigui necessari. La nova regla nul·lable derivada serà tractada sempre i quan no hagi estat tractada anteriorment, per això necessitem un altre contenidor que contingui les ja tractades. Podem veure que si ens trobem en el cas d'una PCFG aleshores les probabilitats associades a les regles derivades de treure algun símbol nul·lable es corresponen amb el producte de la probabilitat de la regla nul·lable $p(r_1)$ per la regla que conté a LHS(r) el símbol nul·lable $p(r_3)$. El codi que podem veure en l'annex C és força extens ja que, com acabem d'explicar, aquesta transformació requereix moltes operacions i té en compte un elevat nombre de condicions. De la implementació volem fer un cert incís en la manera en la què hem creat totes les combinacions de regles traient-ne algun dels seus símbols nul·lables. Tal i com es pot observar, hem creat una llista **l** que conté tantes tuples com combinacions possibles de 1s i 0s de longitud igual a la longitud de la regla original. Hem fet una mena de producte cartesià. Dites tuples representen totes les possibles combinacions, mantenint l'ordre, traient una certa quantitat de símbols (aquells que es corresponguin posicionalment en el lloc en el qual la tupla conté un 1). Per cada tupla, si una certa posició "x" conté un 1, aleshores el símbol que roman a la posició "x" de la regla original és suprimit sempre que aquest símbol es correspongui amb el símbol nul·lable. Afegim tantes noves regles com seqüències diferents hagin sortit d'aquest procediment.

Transformació TERM

Aquestes transformacions afecten aquelles regles r de la gramàtica que presenten símbols terminals a la seva part dreta, és a dir, que es presenten de la forma:

- $X \rightarrow Y_1, Y_2, ..., a_1, ..., Y_{s-1}, Y_s \quad p(r_1) \qquad X, Y_1, Y_2, Y_{s-1}, Y_s \in (N \cup \Sigma); a_1 \in \Sigma$

Quan això succeeix, la transformació consta en afegir un nou símbol no terminal N_a per cadascun dels símbols terminals a_1 que provoquen una d'aquestes transformacions i una nova regla que els relacioni. En el cas que estiguem davant d'una PCFG la probabilitat associada a la nova regla és 1, ja que és de creació pròpia. A més a més, la regla original és modificada tot substituïnt les aparicions del símbol terminal pel nou símbol no terminal creat, per tal que aquesta infracció en la definició de FNC no segueixi vigent. Ho fem de la següent manera:

- $N_a \rightarrow a_1 \quad p(r_2) = 1$
- $X \rightarrow Y_1, Y_2, ..., N_a, ..., Y_{s-1}, Y_s \quad p(r_1) \qquad X, Y_1, Y_2, Y_{s-1}, Y_s \in (N \cup \Sigma); N_a \in N$

Pel que fa al mètode de la classe implementada per dur a terme la transformació que es correspon amb TERM, tal i com podem veure en l'apèndix B, el que realitzem és una nova cerca lineal per totes les regles de la gramàtica buscant si hi ha alguna que necessiti d'aquesta transformació. La creació del diccionari *new_symbols* és necessària per evitar la generació excessiva de símbols no terminals sempre que ens trobem que un mateix símbol terminal provoca una transformació TERM en diferents regles de la gramàtica. Si no tinguéssim aquest contenidor auxiliar, succeïria el següent:

- $X \to Y_1, Y_2, ..., a_1, ..., Y_{s-1}, Y_s \qquad X, Y_1, Y_2, Y_{s-1}, Y_s \in (N \cup \Sigma); a_1 \in \Sigma$
- $W \to Y_1, Y_2, ..., a_1, ..., Y_{s-1}, Y_s \qquad W, Y_1, Y_2, Y_{s-1}, Y_s \in (N \cup \Sigma); a_1 \in \Sigma$
- $N_a \to a_1$
- $X \to Y_1, Y_2, ..., N_a, ..., Y_{s-1}, Y_s \qquad X, Y_1, Y_2, Y_{s-1}, Y_s \in (N \cup \Sigma); N_a \in N$
- $L_a \to a_1$
- $W \to Y_1, Y_2, ..., L_a, ..., Y_{s-1}, Y_s \qquad W, Y_1, Y_2, Y_{s-1}, Y_s \in (N \cup \Sigma); L_a \in N$

Crearíem dues regles i dos nous símbols no terminals que farien la mateixa funció. En canvi, si tenim el coneixement de si algun símbol terminal ja ha estat associat a un nou no terminal i sabem quin és (com ens permet saber el contenidor) aleshores amb crear únicament una regla per cada terminal que forcés TERM ja ens serviria. Al codi de l'annex B, *ant* fa referència a *RHS* de cada regla de la gramàtica. Si trobem un *ant* de longitud major que 1 i conté algun terminal, aleshores procedim amb la transformació. Fem un bucle *while* per seleccionar un símbol de l'abecedari no usat i modifiquem la regla original.

Transformació BIN

Un cop arribat a aquest punt podem afirmar que totes aquelles regles de la gramàtica, en haver passat per les anteriors transformacions de manera seqüencial i en l'ordre que es presenten, i que contenen més d'un símbol a RHS(r) són tots no terminals. Les transformacions del tipus BIN, afecten a aquelles regles que tenen $n \geq 3$ símbols no terminals a RHS(r) i que violen la definició 3 abans proposada. Són de l'estil:

- $X \to Y_1, Y_2, ..., Y_{s-1}, Y_s \qquad p(r_1) \qquad X, Y_1, Y_2, Y_{s-1}, Y_s \in N; s \geq 3$

En aquest cas la transformació consta en crear un nou símbol i una nova regla per cada parell de símbols a RHS(r) de la regla r a transformar i anar substituint en la regla original els nous símbols creats, on cada 1 nou símbol substitueix a 2 originals. Iterem fins que la regla original tingui $s = 2$. Si tenim un PCFG aleshores les noves regles creades se'ls associa probabilitat 1 i la regla original manté la seva probabilitat (ja que es multiplica per aquest 1). Per exemple:

- $X \to Y_1, A_1 \qquad p(r_1)$
- $A_1 \to Y_2, A_2 \qquad p(r_2) = 1$
- $A_2 \to Y_3, A_3 \qquad p(r_3) = 1$
- $A_{s-2} \to Y_{s-1}, Y_s \qquad p(r_s) = 1$

Pel que fa al codi que podem veure en l'annex D és simplement una cerca lineal d'aquelles regles que tenen RHS(r) de longitud major que 2 i, en trobar-les, es realitza un bucle **while** realitzant la substitució explicada anteriorment fins que l'RHS(r) té exactament longitud 2.

Transformació UNIT

Arribat al punt en el qual tenim el símbol d'inici només a LHS(r), totes les regles nul·lables s'han suprimit, els símbols terminals només apareixen en RHS(r) de lognitud 1 i els símbols no terminals que apareixien en aquest mateix costat i tenien longitud major que 2 s'han binaritzat, només queda transformar aquelles regles que contenen símbols terminals a RHS(r) que són solitaris, és a dir de longitud igual a 1. Són de l'estil:

- $X \to Y \qquad p(r_1) \qquad X, Y \in N.$

El que hem de fer és substituir en la regla que provoca aquesta transformació l'aparició del símbol no terminal solitari Y per totes aquelles regles r que tenen LHS(r) = Y, substituint d'aquesta manera la regla original per:

- $Y \to Z_1, ..., Z_s \qquad p(r_2) \qquad Y, Z_1, Z_s \in (N \cup \Sigma)$
- $X \to Z_1, ..., Z_s \qquad p(r_1) * p(r_2) \qquad X, Z_1, Z_s \in (N \cup \Sigma)$

Pot succeir que en la derivació d'aquestes noves regles se'n derivi una altra que necessiti transformació UNIT, de manera que com podem veure en el codi de l'annex E hem de tenir una contenidor per guardar aquelles regles a transformar i anar iterant i afegint sobre aquest sempre que sigui necessari. Afegirem una nova regla derivada a transformar sempre i quan la nova regla a derivar no hagi estat o s'està transformant actualment ja que podriem caure en un bucle infinit. Així doncs, com reflexa el codi, també necessitem un altre contenidor que guardi aquelles regles ja transformades. És important comentar que la probabilitat de les regles derivades és el producte de la regla transformada i de la regla que conté a LHS(r) el símbol no terminal solitari que s'està tractant.

Jocs de Prova: Exemples FNC

A continuació exposarem els resultats que hem obtingut en aplicar la implementacó realitzada per a transformar qualsevol CFG/PCFG a FNC. Els següents jocs de prova han estat creats de manera específica per a testejar les diferents característiques i dificultats de la implementació així com per exemplificar en profunditat els mecanismes que es duen a terme en la transformació. Abans de seguir però, cal fer un incís sobre l'ordre d'aplicació de les transformacions. L'ordre en el què s'han presentat en aquest document és l'ordre d'aplicació que es duu a terme en la nostra implementació ja que preserva el resultat de les anteriors transformacions. Hi ha certs ordres que són destructius respecte d'anteriors canvis.

Joc de Prova 1

El següent PCFG no està en FNC i requereix les transformacions: TERM, BIN, UNIT.

```
S → NP VP 1.0
NP → DET N 0.6 | N 0.4
VP → VT NP PP 0.7 | VI 0.3
PP → with NP 1.0
DET → the 0.6 | a 0.4
N → cat 0.3 | fish 0.5 | knife 0.2
VT → eats 1.0
VI → eats 1.0
```

En aplicar la primera transformació TERM els canvis que obtenim reflecteixen a la perfecció l'explicació feta anteriorment:

```
PP → X NP 1.0
X → with 1.0
```

Després apliquem BIN i els canvis són:

```
B → NP PP 1.0
VP → VT B 0.7 | VI 0.3
```

Per últim, en aplicar UNIT, veiem que el producte de probabilitats es duu a terme de manera correcta i també es fan les substitucions pertinents. El diccionari de llistes resultant (que és el que es mostra en la nostra implementació) és el següent:

```
{'S': [['NP', 'VP', '1.0']], 'NP': [['DET', 'N', '0.6'], ['cat', '0.12'],
['fish', '0.2'], ['knife', '0.08']], 'VP': [['VT', 'B', '0.7'], ['eats',
'0.3']], 'PP': [['X', 'NP', '1.0']], 'DET': [['the', '0.6'], ['a', '0.4']],
'N': [['cat', '0.3'], ['fish', '0.5'], ['knife', '0.2']], 'VT': [['eats',
'1.0']], 'VI': [['eats', '1.0']], 'X': [['with', '1.0']], 'B': [['NP', 'PP',
'1.0']]}
```

Podem observar que el producte entre la regla original i la regla que té a LHS(r) el símbol a transformar en unit s'ha produït correctament, així com les addicions i supressions de símbols diferents.

Joc de Prova 2

A continuació presentem un nou joc de prova en el què podem veure de quina manera es comporta la nostra implementació davant de situacions de transformació DEL i START, i de quina manera gestiona el fet de derivar noves regles que causen una nova transformació. El joc de proves del CFG és el següent:

```
S → A S
A → # | B
B → B | C
C → c
```

En l'anterior gramàtica veiem que una transformació aplicada de manera correcta hauria d'afegir un nou símbol d'inici, hauria d'eliminar la regla nul·lable de manera correcta i, alhora, afegir-ne les regles pertinents. Com podem veure també, aquesta gramàtica ha estat creada específicament per a què hi hagi la possibilitat de ciclar infinitament en aplicar UNIT a la regla $A \rightarrow B$ ja que existeix la regla $B \rightarrow B$ que encara no ha estat tractada. El resultat de la gramàtica en FNC és el següent:

```
{'S': [['A', 'S']], 'A': [['c']], 'B': [['c']], 'C': [['c']],
'So': [['A', 'S']]}
```

Veiem que la gramàtica resultant està en FNC. La implementació ha sabut tractar les noves regles que requerien UNIT, així com aquelles transformacions UNIT que derivaven en transformacions que ja havíem tractat, com per exemple $B \rightarrow B$, que ha estat suprimida correctament. Veiem que en el càlcul de totes les possibles regles derivables en aplicar DEL a $A \rightarrow \#$, de manera intermitja s'ha creat la regla $S \rightarrow S$ però ha estat eliminada al ser UNIT i al ser ja tractada.

La resta de jocs de prova s'adjunten a l'entrega d'aquest treball per a què el lector/avaluador pugui testejar de manera local la nostra implementació per a diferents casos, alguns d'ells força extensos.

5 Extra: Creació de l'arbre sintàctic

En aquesta secció exposarem una part extraordinària i afegida que hem decidit implementar per a complementar la feina feta entre la part obligatòria i la opcional proposada. Hem implementat la traça dels algorismes CYK i PCYK, fets amb programació dinàmica, que representa l'arbre sintàctic que es forma en una determinada gramàtica per a generar una certa paraula que és acceptada per la pròpia gramàtica. És l'arbre que adopta la gramàtica per a una certa paraula *ACCEPTADA*. Aquesta implementació s'inclou en les pròpies classes *CYK* i *CYKProb* explicades anteriorment, en les quals s'inclou com a un mètode més. Consta de dues parts, la generació de la traça que realitza l'algorisme en el seu "camí" cap a una solució, si aquesta existeix i la generació i visualització de l'arbre binari a partir d'aquesta donada traça.

Traça

Per generar la traça hem creat un mètode privat anomenar `_build_tree`. Aquest mètode genera l'arbre recursivament. Per fer-ho fa ús de la codificació que hem utilitzat per guardar de quines caselles provenia la RHS d'una regla en una casella concreta. La descodificació que fem de fet és la mateixa que realitzem en el mètode `_search_rules()` de la classe `CKYProb`.

El mètode `_build_tree` rep 3 arguments. Les posicions x, y de la casella i el LHS de la regla que hem de buscar en aquella casella. Com hem dit és un mètode recursiu, per tant, la primera crida serà tindrà els paràmetres per obtenir l'arrel del parsing tree. Aquests paràmetres són:

- $x = 0$
- y = len(word) -1
- $term$ = self.best_sol

La casella que mirem amb aquestes coordenades és l'última de totes, la que conté la solució. El terme que li passem és `self.best_sol` perquè en el cas que hi hagi diversos parsing trees volem que ens doni el millor. Això només és útil en el cas probabilístic, ja que en l'altre no sabem quin arbre és millor, per tant `self.best_sol` = `self.start`. Un cop arribat aquí es descompon la informació i es crida recursivament al fill dret i al fill esquerre de l'arbre que serán els dos no terminals en la part RHS de la regla (ja que la gramàtica està en FNC). El cas base és quan el subarbre només té un fill, ja que ha de ser un símbol terminal.

Aquest mètode l'hem hagut de repetir per la classe `CKY` i `CKYProb` ja que en el segon cas la informació està guardada com (symbol, prob) i havíem de quedar-nos només amb el primer element de la tupla que és el símbol i no amb la probabilitat. A part d'això el codi és el mateix.

Per guardar l'arbre utilitzem diccionaris, on la clau és l'arrel i el valor una tupla amb el fill dret i el fill esquerre. D'aquesta manera cada subarbre es representa com un diccionari. Aquest és el format en que rebrà l'arbre el mètode que genera el dibuix.

Generació de l'arbre

Un cop determinada l'estructura amb la qual representem la traça de l'execució de l'algorisme que accepta una certa paraula donada una gramàtica, la funció que genera l'arbre l'hem de contextualitzar al voltant d'aquesta estructura.

El que hem realitzat ha estat la creació d'un graf dirigit amb el mòdul de *Python, Graphviz* que ens ha permès la creació del graf i la posterior renderització a imatge.

La implementació d'aquesta funcionalitat l'hem realitzat a partir del mètode **self.show_tree(word)**. El paràmetre que hi passem a la funció és la paraula de la que volem que es generi l'arbre associat a la gramàtica que l'accepta, de manera que la paraula ha d'estar prèviament acceptada pel mètode **self.check_word(word)**. Aquest mètode crida a la funció que crea la traça de l'algorisme feta per a aquesta paraula, i hi passa el diccionari de tuples (traça) a la funció **build_graph(tree)** que és la que genera el graf i el renderitza.

La generació del graf de *Graphviz* és força senzilla però alhora divertida ja que la funció que hem implementat per fer-ho és recursiva, aprofitant-nos així de l'estructura d'arbre binari que tenen tots els arbres sintàctics que provenen de gramàtiques en FNC.

El cas base d'aquesta funció comprova si els fills de l'arrel de l'arbre actual són una tupla. En cas que no, és a dir si l'arrel no té **fesq** i **fdre** aleshores el node actual deriva en un node terminal i es pot considerar una fulla. Aquí s'afegeix l'aresta entre el node fulla no terminal i el node terminal al graf i no es fa cap crida recursiva. En el cas recursiu s'agafa l'arrel de l'arbre i es fan les següents crides recursives:

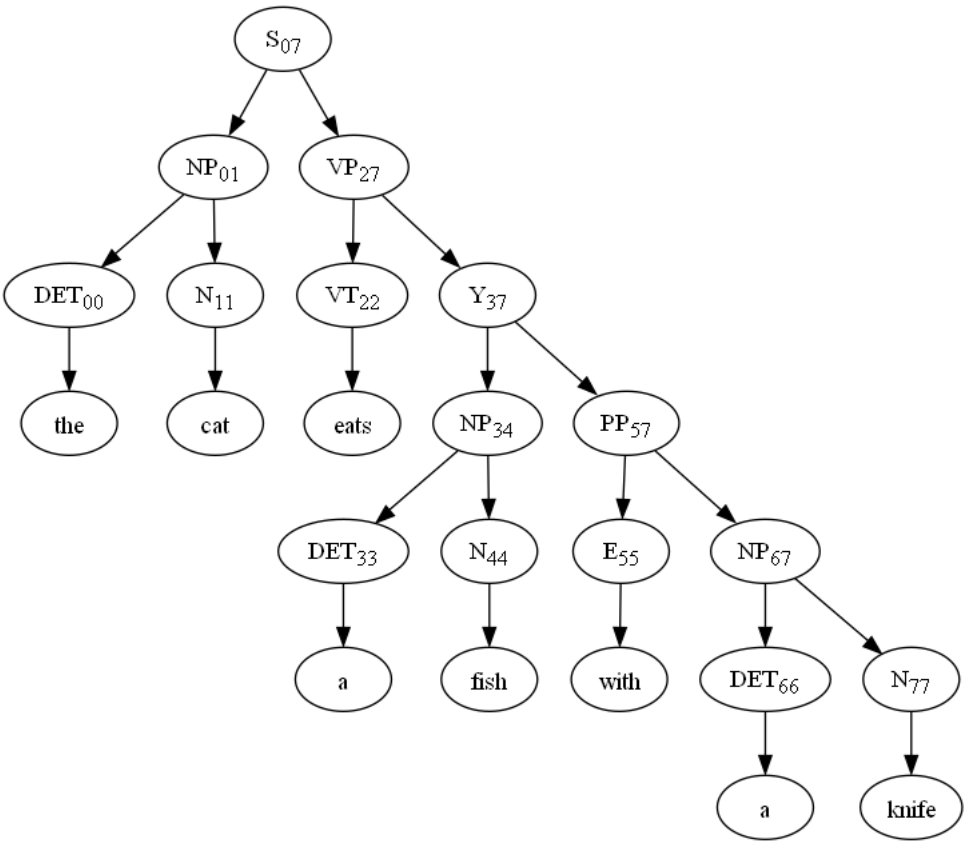
```
else:
    fesq = build_graph(tree[root][0])
    fdre = build_graph(tree[root][1])
    h.edge(root,fesq);h.edge(root,fdre)

    return root
```

Com veiem, el fet que la funció retorni l'arrel de l'arbre actual fa possible que les arestes es construeixin de manera recursiva ja que el **fesq** i el **fdre** d'un cert arbre són arrels d'un altre arbre i així successivament. Cal incidir en el fet que la instància del Digraf de *Graphviz* es crea a fora de la funció i a dins del mètode.

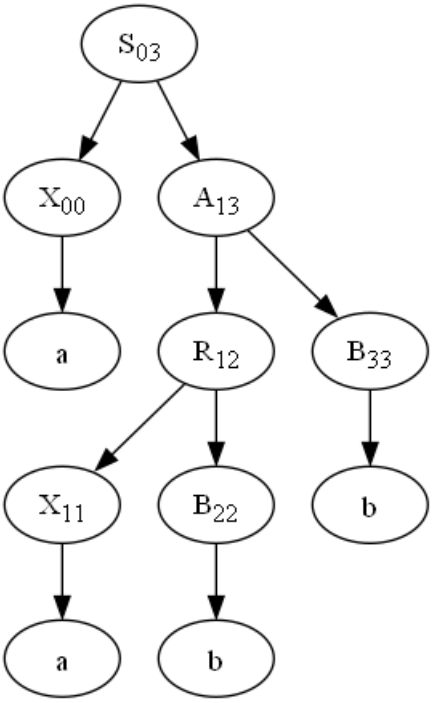
Per últim, cal comentar que els nodes de l'arbre estan personalitzats no només pel símbol no terminal o terminal si no que també hi apareix informació sobre la casella de la graella generada pel CYK/PCYK de manera que podem observar perfectament el recorregut i resultat de l'algorisme corresponent. A continuació podem veure dues imatges corresponents a l'arbre generat per dues gramàtiques diferents i per a paraules o seqüències de paraules acceptades per aquestes.

Figure 2: Arbre sintàctic de la gramàtica del Joc de Prova 1 per a una certa frase



En la Figura 2 podem veure l'arbre sintàctic creat per la gramàtica del Joc de Prova 1 de l'anterior secció per a la seqüència de paraules *"the cat eats a fish with a knife"*. En aquesta figura veiem els subíndexos indicants les posicions de les caselles de la taula de programació dinàmica, així com la seqüència de nodes que provoca l'acceptació d'aquesta seqüència tot formant l'arbre sintàctic (amb noms, sintagmes i verbs) associat.

Figure 3: Arbre sintàctic d'una de les gramàtiques incloses als jocs de prova per a una certa paraula.



En la Figura 3 observem un altre arbre sintàctic d'una de les gramàtiques que es proposen com a jocs de prova per a la seqüència *"aabb"* en el què veiem quines associacions de regles es formen per donar lloc a la seqüència acceptada.

6 Conclusions

Després de realitzar aquesta pràctica hem aconseguit consolidar diversos conceptes, no només apressos a PAA sinó a altres assignatures com PLH i PA2. Per una banda, hem implementat el programa utilitzant la *Programació Orientada a Objectes* que vam aprendre a PA2 i que no havíem utilitzat gaire. Després d'utilitzar-la per aquesta pràctica ens hem donat compte del potencial que té i de per què és tan utilitzada.

Per altra banda, hem implementat l'algoritme eficientment utilitzant Programació Dinàmica i aplicant-la a un cas no tan comú com els realitzats a classe, ja que la regla per omplir la taula era bastant més complexa. A més, hem implementat el parsing tree, que vindria a ser la traça de la taula, que també ha tingut la seva gràcia. També amb això, hem consolidat conceptes de parsing de gramàtiques i anàlisi de constituents, apresos tant a l'assignatura de PAA com sobretot a PLH, relacionant així dues assignatures de la carrera.

A més després de realitzar aquest treball hem vist l'utilitat d'aquest algoritme i l'importància del parsing de gramàtiques. Ja que, tot i que les gramàtiques que fem servir són de "joguina", amb una gramàtica probabilística ben feta i amb sentit, pot ser una eina molt potent.

Appendices

A Classe FNC: mètode START

```
def _START(self):
    n_r = self.r.copy()
    for key,value in self.r.items():
        for c in range(len(value)):
            ant = value[c]
            if (self.s in ant):
                if self._k:
                    n_r["So"] = [[self.s,"1.0"]]
                else:
                    n_r["So"] = [[self.s]]
                self.n_t.add("So")
                self.s = "So"
    self.r = n_r.copy()
```

B Classe FNC: mètode TERM

```
def _TERM(self):
    n_r = self.r.copy()
    new_symbols = {}
    for key,value in self.r.items():
        for c in range(len(value)):
            ant = n_r[key][c]
            if (len(ant)>(1+self._k)):
                for i in range(len(ant)-self._k):
                    if (ant[i] in self.t):
                        if ant[i] not in new_symbols:
                            n = chr(random.randint(ord('A'), ord('Z')))
                            while n in self.n_t:
                                n = chr(random.randint(ord('A'), ord('Z')))
                            new_symbols[ant[i]] = n
                        else:
                            n = new_symbols[ant[i]]
                        if self._k:
                                n_r[n] = [[ant[i],"1.0"]]
                        else:
                            n_r[n] = [[ant[i]]]
                        ant[i] = n; self.n_t.add(n)
    self.r = n_r.copy()
```

C Classe FNC: mètode DEL

```
def _DEL(self):
    n_r = self.r.copy()
    eps_rules = set();removed=set()
    for key,value in self.r.items():
        for c in range(len(value)):
            ant = n_r[key][c]
            if len(ant)==(1+self._k) and (ant[0] == "#"):
                if self._k:
                    eps_rules.add((key,ant[-1]))
                else:
                    eps_rules.add((key))
                del n_r[key][c]
                removed.add(key)

    while len(eps_rules)>0:
        if self._k:
            key,prob = eps_rules.pop()
        else:
            key = eps_rules.pop()

        n_r_2 = n_r.copy()

        for clau,valor in n_r_2.items():
            for c in range(len(valor)):
                if self._k:
                    ant = n_r[clau][c][:-1]
                    prob_ant = n_r[clau][c][-1]
                else:
                    ant = n_r[clau][c]
                if key in ant:
                    l =  list(product(range(2),repeat=len(ant)))[1:]
                    for cmb in l:
                        new_rule = []
                        for x in range(len(cmb)):
                            if cmb[x] == 0:
                                new_rule.append(ant[x])
                            elif (cmb[x] == 1) and (ant[x] != key):
                                new_rule.append(ant[x])

                        if (new_rule == []):
                            new_rule.append("#")

                        if self._k:
                            new_rule.append(str(round(float(prob[-1])*
                            float(prob_ant[-1]),3)))

                        if (new_rule[0] == "#"):
                            if clau not in removed:
```

```
                                    if self._k:
                                        eps_rules.add((clau,new_rule[-1]))
                                    else:
                                        eps_rules.add(clau)
                                    removed.add(clau)

                                elif new_rule not in n_r[clau]:
                                    n_r[clau].append(new_rule)

        self.r = n_r.copy()
```

D Classe FNC: mètode BIN

```
    def _BIN(self):
        n_r = self.r.copy()
        for key,value in self.r.items():
            for c in range(len(value)):
                ant = n_r[key][c]
                if (len(ant)>(1+self._k)):
                    while not len(ant)==(2+self._k):
                        n = chr(random.randint(ord('A'), ord('Z')))
                        while n in self.n_t:
                            n = chr(random.randint(ord('A'), ord('Z')))
                        if self._k:
                            n_r[n],ant[-3:-1] = [ant[-3:-1]+["1.0"]],n
                        else:
                            n_r[n],ant[-2:] = [ant[-2:]],n
                        self.n_t.add(n)
        self.r = n_r.copy()
```

E Classe FNC: mètode UNIT

```
    def _UNIT(self):
        n_r = self.r.copy()
        unit = []
        removed = []
        for key,value in n_r.items():
            for c in range(len(value)):
                ant = n_r[key][c]
                if (len(ant) == (1+self._k)) and (ant[0] in self.n_t):
                    unit.append((key,ant))

        while len(unit)>0:
            key,rhs = unit.pop(0)
            removed.append((key,rhs))
            n_r[key].remove(rhs)
```

```
    if rhs[0] != key:
        for a in n_r[rhs[0]]:
            if (key,a) not in removed:
                a_new = a.copy()
                if self._k:
                    a_new[-1] = str(round(float(a_new[-1])*
                    float(rhs[-1]),3))
                n_r[key].append(a_new)
                if (len(a) == (1+self._k)) and (a[0] in self.n_t):
                    unit.append((key,a_new))
self.r = n_r.copy()
```