# Informe de la Pràctica

Lluc Furriols Llimargas, Pau Prat Moreno

31 de maig de 2024



Universitat Politècnica de Catalunya
Grau en Intel·ligència Artificial
Programació i Algorísmia Avançada

**Resum**

Aquest document correspon a l'informe de la pràctica de l'assignatura Programació i Algorísmia Avançada del Grau en Intel·ligència Artificial de la Universitat Politècnica de Catalunya (UPC).

En aquest informe es detallen els passos que hem seguit per tal d'implementar el programa que implementa l'algorisme de CKY (Cocke–Kasami–Younger) que se'ns proposava en codi Python, així com els diferents fitxers necessaris per aquesta pràctica.

Per tal de provar el funcionament del programa, es poden modificar els fitxers de input per tal de provar les gramàtiques i les paraules que l'usuari vulgui.

# Índex

# 1 Descripció dels fitxers

En aquest apartat es presenten els diferents fitxers que s'han utilitzat per dur a terme aquesta pràctica. Més endavant s'explica de manera més detallada la implementació de codi que hem realitzat en cadascun d'aquests fitxers. Per tant, a continuació es resumeix què fa cada arxiu.

## 1.1 main.py

En primer lloc, el fitxer main és el punt d'entrada del projecte que, mitjançant la lectura de gramàtiques i paraules des d'un fitxer d'entrada, comprova si aquestes paraules formen part de la gramàtica utilitzant l'algorisme CKY. Escriu els resultats detallats en un fitxer de sortida, proporcionant una visió completa de les comprovacions realitzades.

Hem optat per implementar també l'extensió 1 (convertir qualsevol CFG en CNF). Així doncs, el fitxer main, també admet gramàtiques de qualsevol tipus, i genera la corresponent gramàtica en CNF.

Per tal de tenir una fàcil interacció amb el programa, a l'inici de l'execució es pregunta a l'usuari quin tipus de gramàtica vol provar d'executar (CFG o Probabilístic CFG), i també es pregunta si l'usuari vol veure la taula generada per l'algorisme o no. A partir de les preferències de l'usuari, el programa processa un fitxer amb CFG's o un fitxer amb PCFG's i les seves respectives paraules.

## 1.2 CFG.py

La classe CFG està dissenyada per facilitar el treball amb gramàtiques lliures de context, incloent la conversió automàtica a CNF si és necessari i la implementació de l'algorisme CKY per verificar si una paraula pertany a la gramàtica. Això permet que el codi pugui processar tant gramàtiques en format CFG general com en CNF directament. Aquesta classe utilitza el codi Transform_into_cnf per convertir a CNF quan és necessari.

## 1.3 Transform_into_cnf.py

Aquesta classe permet transformar qualsevol CFG en una gramàtica equivalent en CNF, assegurant així que es pot utilitzar amb l'algorisme CKY

## 1.4 input.txt

El fitxer d'entrada està dissenyat per incloure diverses gramàtiques lliures de context (CFG) i llistes de paraules que es volen verificar amb aquestes gramàtiques. Cada gramàtica i les paraules associades estan separades per una línia en blanc. El format específic del fitxer és el següent:

- **Regles de la Gramàtica**:
  - Les regles de producció es defineixen utilitzant la notació `LHS -> RHS` (*left-hand side* i *right-hand side*).
  - Els símbols no-terminals es representen amb lletres majúscules.
  - Els símbols terminals es representen amb lletres minúscules.
  - Les múltiples alternatives per a una producció es separen amb el símbol '`|`'.
- **Llista de Paraules**:

  - Les paraules que es volen verificar es llisten una per línia després de les regles de producció.

- **Separació entre Gramàtiques**:
  - Una línia en blanc separa les diferents gramàtiques i les seves respectives llistes de paraules.

És a dir, un exemple d'arxiu d'entrada podria ser:

```
S -> a | XA | AX | b
A -> RB
B -> AX | b | a
X -> a
R -> XB

a
b
aa
ab
ba
abaa
```

Figura 1: Exemple d'arxiu input.txt

## 1.5 output.txt

Aquest és el fitxer que és generat després d'executar el main.py. En aquest fitxer es poden veure quines paraules poden ser generades i quines no per una gramàtica donada. És a dir, per la gramàtica donada, l'arxiu d'output mostra `True` o `False` per cada paraula, indicant si pot ser o no generada per la gramàtica en qüestió. En el cas de l'exemple anterior, s'imprimirien els següents resultats en aquest arxiu:

S -> a | XA | AX | b
A -> RB
B -> AX | b | a
X -> a
R -> XB

Grammar is in CNF
Results:
a: True
b: True
aa: False
ab: False

ba: False
abaa: True

Com podem veure, com que en aquest cas la gramàtica proporcionada està en Forma Normal de Chomsky (*CNF* en anglès), en aquest arxiu s'especifica que està en CNF, i per tant, la gramàtica no ha patit cap modificació.
En cas que la gramàtica donada no estigués en CNF, també es mostra la conversió feta per passar de qualsevol gramàtica CFG fins a una CNF.
A més, com hem vist anteriorment, l'usuari pot decidir que s'imprimeixi la taula dinàmica de l'algorisme CKY per tal que es pugui visualitzar què ha fet l'algorisme.
A continuació es mostra un exemple d'entrada i de sortida per il·lustrar el contingut del fitxer generat.

### 1.5.1 Entrada

Recordem que el fitxer d'entrada pot contenir múltiples gramàtiques amb les seves paraules que es volen comprovar. Per l'exemple farem només una gramàtica: Introduïm una gramàtica molt senzilla que NO està en CNF que genera paraules amb el mateix nombre de **a**'s que de **b**'s.

```
Gramàtica:
S -> aSb |

Paraules a verificar:
aaabbb
abaaa
```

### 1.5.2 Sortida

```
Original Grammar:
S -> aSb |

Grammar is not in CNF

Converted into CNF Grammar:
S -> AC | AB
A -> a
B -> b
C -> SB
```

```
RESULTS:
```

```
aaabbb: True

Table generated by the CKY algorithm:
+---+---+---+---+---+---+
| A |   |   |   |   | S |
+---+---+---+---+---+---+
|   | A |   |   | S | C |
+---+---+---+---+---+---+
|   |   | A | S | C |   |
+---+---+---+---+---+---+
|   |   |   | B |   |   |
+---+---+---+---+---+---+
|   |   |   |   | B |   |
+---+---+---+---+---+---+
|   |   |   |   |   | B |
+---+---+---+---+---+---+


abaaa: False

Table generated by the CKY algorithm:
+---+---+---+---+---+
| A | S |   |   |   |
+---+---+---+---+---+
|   | B |   |   |   |
+---+---+---+---+---+
|   |   | A |   |   |
+---+---+---+---+---+
|   |   |   | A |   |
+---+---+---+---+---+
|   |   |   |   | A |
+---+---+---+---+---+
```

## 1.6 Probabilistic CKY

Per implementar l'algorisme CKY probabilístic hem creat una nova classe (PCKY), la qual és molt semblant a la classe CKY però està dissenyada per suportar probabilitats en les regles de producció.

### 1.6.1 Inputs per el probabilistic CKY

Finalment, ens trobem amb el input_PCKY.txt. Aquest fitxer conté una gramàtica lliure de context probabilística (PCFG) i una llista de paraules per verificar amb aquesta gramàtica. Cada producció inclou una probabilitat associada. La gramàtica i les paraules associades estan separades per una línia en blanc.

**Exemple del Fitxer d'Entrada**:

```
S -> AB | CD | CB | SS 0.6
A -> BC | a 0.1
B -> SC | b 1.0
C -> DD | b 0.2
D -> BA 0.2

ab
ba
aabb
abab
baba
```

# 2 Implementació

En aquest apartat es detalla el codi que hem desenvolupat per aquest projecte, justificant cada decisió que hem pres i explicant detalladament com ho hem implementat en codi Python.

## 2.1 Classe CFG

Tot i que en un inici no teníem pensat realitzar el projecte d'aquesta manera, durant els primers dies de treball vam decidir que la manera més còmode seria implementar les gramàtiques CFG amb classes, ja que d'aquesta manera podríem mantenir una estructura clara i ben organitzada del codi. A més, l'ús de classes permet afegir noves funcionalitats de manera més senzilla i estructurada, la qual cosa ens seria molt útil posteriorment en voler transformar una gramàtica a CNF. Així doncs, també hem fet ús de classes en l'extensió 1, per tal de poder convertir qualsevol CFG a Forma Normal de Chomsky, que expliquem més en detall posteriorment.

Per tant, la classe **CFG** s'encarrega de gestionar les gramàtiques lliures de context per tal d'aplicar l'algoritme CKY i així determinar si una paraula pertany al llenguatge generat per la mateixa gramàtica.

En primer lloc, el mètode `__innit__` inicialitza un objecte d'aquesta classe, en què es defineixen les següents propietats:

- **self.grammar** emmagatzema la gramàtica original.
- **self.rules** conté les regles de la gramàtica parsejades mitjançant el mètode parse_grammar.
- **self.is_cnf** és un booleà que indica si la gramàtica està en CNF, comprovat pel mètode check_if_cnf.
- **self.cnf_grammar** conté la gramàtica en CNF. Si la gramàtica no està en CNF, es transforma utilitzant la classe CNF.

#### parse_grammar

Aquest mètode és crucial ja que, a partir de les regles de la gramàtica donada, retorna un diccionari on les claus són les parts esquerres de les regles (lhs), i els valors són llistes amb les parts dretes de les regles (rhs).
Per exemple, si tenim la següent gramàtica:
S -> AB | BC
A -> a
B -> b | a
C -> c
El mètode `parse_grammar` retorna el dicccionari: {'S': ['AB', 'BC'], 'A': ['a'], 'B': ['b', 'a'], 'C': ['c']}
Finalment, aquest diccionari s'emmagatzema a la variable d'instància `self.rules`.

#### cky_algorithm

La funció cky_algorithm és l'encarregada d'implementar l'algoritme. Hem utilitzat els conceptes de programació dinàmica adquirits durant aquest quadrimestre, així com els coneixements sobre aquest algoritme que s'ens ha explicat a l'assignatura de Processament del Llenguatge per tal de

crear la taula del cky. Aquesta s'encarrega de verificar, per una paraula donada, si pot ser generada per la gramàtica en qüestió.
La taula és una matriu bidimensional de mida $n \times n$, on 'n' és la longitud de la paraula. Cada cel·la de la taula és un conjunt que conté els no terminals que poden generar la subcadena corresponent de la paraula. El procés d'omplir la taula es fa en quatre passos principals:

- **Inicialització de la Taula**: Es crea la taula, en què inicialment totes les caselles contenen el conjunt buit.

```
table = [[set() for _ in range(n)] for _ in range(n)]
```

- **Omplir la diagonal**: Aquesta part de la taula representa subcadenes de longitud 1. Per cada símbol terminal en la paraula, es comprova si existeix una regla de la gramàtica on el terminal és a la dreta de la regla. Si existeix, el no terminal a l'esquerra de la regla s'afegeix a la cel·la corresponent de la taula.

```
for j in range(n):
        for lhs, rhs_list in rules.items():
            for rhs in rhs_list:
                if rhs == word[j]:
                    table[j][j].add(lhs)
```

- **Omplir la resta de la taula**: Es procedeix a omplir la resta de la taula, considerant totes les possibles subdivisions de la paraula. Per a cada subdivisió, es busquen produccions de la gramàtica que concordin amb les combinacions de subcadenes i s'afegeixen els símbols no-terminals corresponents al conjunt de la casella.

```
# Fill the rest of the table
for span in range(2, n + 1):
    for i in range(n - span + 1):
        j = i + span - 1
        for k in range(i, j):
            for lhs, rhs_list in rules.items():
                for rhs in rhs_list:
                    if len(rhs) == 2:
                        B, C = rhs
                        if B in table[i][k] and C in table[k + 1][j]:
                            table[i][j].add(lhs)
```

- **Verificació de Generació**: Finalment, comprova si la paraula pot ser generada per la gramàtica comprovant si l'símbol inicial de la gramàtica ('S') es troba a la cel·la superior dreta de la taula.

```
return ['S' in table[0][n-1], table]
```

### 2.1.1 Exemple Visual

A continuació veiem un exemple visual per la gramàtica:

```
S -> AB | BA
A -> a
B -> b
```

i la paraula **aba**:

```
+---+---+---+
| A | S |   |
+---+---+---+
|   | B | S |
+---+---+---+
|   |   | A |
+---+---+---+
```

En aquesta taula, cada cel·la (i, j) conté el conjunt de símbols de la gramàtica que poden generar la subcadena de la paraula des de la posició i fins a la posició j. Per exemple, la cel·la (0, 1) conté 'S' perquè la subcadena "ab"pot ser generada per 'S' segons la gramàtica.

No obstant, però, veiem que aquesta paraula no pot ser generada per la gramàtica, ja que la casella de dalt a la dreta, no hi ha una 'S', i això indica que des del símbol inicial de la gramàtica, no es pot arribar a la paraula **aba**.

# 3 Extensió 1 - Convertir a CNF

En la primera extensió del treball se'ns demanava poder convertir qualsevol gramàtica CFG a Forma Normal de Chomsky (CNF). Per tal de fer-ho de la manera que creiem més adequada, vam crear la classe **CNF**, que es troba a l'arxiu `transform_into_cnf`.
El constructor de la classe inicialitza l'objecte amb les regles de la gramàtica donada i crida al mètode `transform_to_CNF` per començar la transformació. A més, s'emmagatzema el conjunt de símbols no terminals existents per tal d'assegurar que els nous símbols que es crearan siguin únics i no hi hagi lletres repetides.

## 3.1 Generació de No-Terminals Únics

El mètode `get_unique_nonterminal` genera un símbol no terminal únic que no existeix en el conjunt actual de símbols no terminals. Hem assumit que no hi haurà cap gramàtica que necessitarà més de 26 símbols no terminal (que són totes les possibles lletres del abecedari), ja que normalment una gramàtica no té tantes produccions que requereixin l'ús de tants símbols no terminals addicionals. Per tant, el mètode recorre l'alfabet i verifica si cada lletra està present en el conjunt de no terminals existents. Quan troba una lletra que no hi és, l'afegeix al conjunt i la retorna com el nou símbol no terminal. Si s'acaben les lletres de l'alfabet sense trobar un símbol únic, apareix una excepció indicant que s'han esgotat els símbols únics disponibles.

## 3.2 Eliminació de Produccions Nul·les

El mètode `remove_null_productions` elimina les produccions nul·les (produccions que generen la paraula buida) de la gramàtica. Aquest procés es realitza en dos passos:

1. Inicialment s'identifiquen els símbols que poden generar la paraula buida directament. Després, es fa servir un bucle iteratiu per ampliar aquest conjunt, afegint símbols que poden generar la paraula buida indirectament a través de produccions que contenen altres símbols *nullable* (que poden generar la paraula buida).

2. Es creen noves produccions que consideren totes les possibles combinacions de símbols no terminals que poden contenir aquests símbols *nullable*, assegurant-se que la gramàtica resultant no generi la paraula buida de manera innecessària.

## 3.3 Eliminació de Produccions Unitàries

El mètode `unitary_rule` elimina les produccions unitàries (produccions del tipus $A \to B$, on $A$ i $B$ són no terminals) de la gramàtica. Substitueix aquestes produccions per les produccions corresponents de la variable no terminal a la qual es referencien. És a dir, imaginem que tenim la gramàtica:
$A \to B$
$B \to C$
$C \to c$
Un cop eliminem les produccions unitàries, aquesta gramàtica quedaria de la següent manera:
$A \to c$
Per tant, la implementació que hem realitzat per aquesta regla comença identificant totes les produccions unitàries i emmagatzemant-les en una llista. Després, utilitza un bucle per eliminar aquestes

produccions unitàries i les substitueix amb les produccions corresponents de les variables no terminals referenciades. Si una producció unitària referència una altra producció unitària, també es processa de manera recursiva fins a eliminar totes les produccions unitàries.

### 3.4 Eliminació de Terminals en Produccions Mixtes

El mètode `hybrid_rule`, que pren aquest nom ja que que coneixíem aquesta regla amb el nom de 'regla híbrida' (ensenyada a l'assignatura PLH), consisteix en eliminar els símbols terminals de la part dreta de les produccions quan estan barrejats amb altres no terminals o terminals. Així doncs, es substitueixen aquests terminals per nous no terminals que generen únicament aquests terminals.

### 3.5 Eliminació de Produccions No Binàries

El mètode `non_binary_rule` transforma les produccions que tenen més de dos no terminals a la part dreta en produccions binàries. Això es realitza creant nous no terminals per representar combinacions de dos no terminals.

### 3.6 Comprovació de Produccions Binàries

El mètode `is_binary` comprova si totes les produccions tenen com a màxim dos símbols no terminals a la part dreta. Ho fa recorrent totes les produccions i comptant els símbols no terminals en cada producció. Si troba alguna producció amb més de dos no terminals, retorna False; en cas contrari, retorna True.

### 3.7 Transformació a CNF

El mètode `transform_to_CNF` és el mètode principal que coordina els altres mètodes per transformar la gramàtica original a la Forma Normal de Chomsky. Aquest mètode segueix els passos següents:

1. Eliminar produccions nul·les utilitzant `remove_null_productions`.
2. Eliminar produccions unitàries amb `unitary_rule`.
3. Eliminar terminals en produccions mixtes amb `hybrid_rule`.
4. Eliminar produccions no binàries utilitzant `non_binary_rule` fins que totes les produccions siguin binàries (`is_binary`).
5. Finalment, assegura que el símbol inicial 'S' estigui primer en l'ordre de les regles de la gramàtica.

### 3.8 Obtenció de la Gramàtica en Forma Normal de Chomsky

Finalment, el mètode `get_cnf_grammar` simplement retorna la gramàtica que ha estat transformada a la Forma Normal de Chomsky (CNF). Quan es crea una instància de la classe `CNF`, el constructor `__init__` immediatament crida al mètode `transform_to_CNF`, que s'encarrega de transformar la gramàtica original a CNF. La gramàtica resultant es guarda a l'atribut `cnf_grammar` de l'objecte.

És per això que en l'arxiu `cfg.py`, en el cas que la gramàtica no estigui en Formal Normal Chomsky (`not self.is_cnf`), es crida a `CNF(self.rules).get_cnf_grammar()`. Així doncs, aquesta crida

transforma la gramàtica original a CNF utilitzant els mètodes de la classe `CNF`, i després guarda la gramàtica resultant a l'atribut `self.cnf_grammar`.

# 4 Extensió 2 - Probabilistic CKY

La classe **PCFG** és força semblant a la classe **CFG**, però s'han fet algunes modificacions per tal de poder tractar amb gramàtiques probabilístiques. Això ens permet calcular la probabilitat de generació d'una paraula específica per una gramàtica probabilistica donada. Aquesta classe, però, espera que la gramàtica donada estigui en CNF, i per tant no són necessaris els mètodes per comprobar que la gramàtica estigui en CNF ni els de transformar-la en CNF.

En primer lloc, el mètode `__init__` inicialitza un objecte d'aquesta classe, en què es defineixen les següents propietats:

- **self.grammar** emmagatzema la gramàtica original.
- **self.rules** conté les regles de la gramàtica parsejades mitjançant el mètode `parse_probabilistic_grammar`.

**parse_probabilistic_grammar**

Aquest mètode a partir de les regles de la gramàtica donada, retorna un diccionari on les claus són les parts esquerres de les regles (lhs), i els valors són tuples amb les parts dretes de les regles (rhs) i les seves probabilitats associades. Aquest metode és important ja que serveix per fer un "preprocessament"de les regles de gramàtica. A diferència del parse_grammar de la classe CFG, aquest mètode contempla la probabilitat de les gramàtiques, però té una estructura practicament igual. A continuació veiem la conversió que fa:

"S -> AB | BC 0.6"-----> 'S': (['AB', 'BC'], 0.6)

**cky_algorithm**

La funció `cky_algorithm` implementa l'algoritme CKY per a gramàtiques probabilístiques. A diferència de la versió no probabilística, cada cel·la de la taula CKY emmagatzema un diccionari amb els no terminals i les seves probabilitats associades. El funcionament és idèntic al de la classe CFG, però a més, aquest mètode emmagatzema les probabilitats de cada cel·la. Com que el codi és pràcticament igual no cal tornar-lo a ensenyar.

### 4.0.1 Exemple del funcionament de la funció cky_algorithm

Considerem la gramàtica probabilística següent:

```
S -> AC | AB 1.0
A -> a 0.5
B -> b 0.5
C -> SB 1.0
```

i la paraula `aabb`.
1. **Inicialització de la taula**:

| {} | | | |
|---|---|---|---|
| | {} | | |
| | | {} | |
| | | | {} |

2. **Omplir la diagonal**:

| A: 0.5 | | | |
|---|---|---|---|
| | A: 0.5 | | |
| | | B: 0.5 | |
| | | | B: 0.5 |

3. **Omplir la resta de la taula**: - Considerant la subcadena `aa` (`span = 2`):

| {A} | | | |
|---|---|---|---|
| | {A} | | |
| | | {B} | |
| | | | {B} |

No hi ha cap regla que permeti crear la subcadena 'aa'

- Considerant la subcadena `ab` (`span = 2`):

| A: 0.5 | | | |
|---|---|---|---|
| | A: 0.5 | S: 0.25 | |
| | | B: 0.5 | |
| | | | B: 0.5 |

- `table[1][2]` conté {`S: 0.25`}. La probabilitat de 0.25 prové de multiplicar la casella A per la B per la probabilitat de S (0.5*0.5*1)

- Considerant la subcadena `abb` (`span = 3`):

| | | | |
|---|---|---|---|
| A: 0.5 | | | |
| | A: 0.5 | S: 0.25 | C: 0.125 |
| | | B: 0.5 | |
| | | | B: 0.5 |

-Veiem que es pot generar la paraula `abb` a partir de la regla C. La regla C produeix SB amb una probabilitat de 1. Per tant, en la taula escrivim una C: 0.125, on la probabilitat prové de multiplicar: 0.25*0.5*1.

L'algorisme anirà iterant per totes les subcadenes i omplint quan sigui possible generar aquella subcadena amb un símbol no-terminal. Al final de totes les iteracions la taula quedarà així:

| | | | |
|---|---|---|---|
| A: 0.5 | | | S: 0.062 |
| | A: 0.5 | S: 0.25 | C: 0.125 |
| | | B: 0.5 | |
| | | | B: 0.5 |

-On 'S': 0.062 prové de multiplicar la probabilitat de C: 0.125, A: 0.5, i S: 1.
Per tant -> 0.125*0.5*1 = 0.0625

# 5 Conclusions - Valoració de l'aprenentatge adquirit

Un cop finalitzada la pràctica, podem dir que aquest treball ens ha permès aprofundir en el camp de les gramàtiques lliures de context i la seva implementació en Python, així com la transformació a Forma Normal de Chomsky i el poder tractar amb gramàtiques probabilístiques.

Inicialment, vam decidir organitzar el nostre codi utilitzant classes, una decisió que creiem que va ser encertada. La classe CFG ens ha permès mantenir una estructura clara i ben organitzada del codi, facilitant també la gestió de les gramàtiques i la implementació de l'algoritme CKY. A més, aquesta estructura també ens va ser molt útil a l'hora d'implementar l'extensió 1, ja que des d'un principi teníem clar que, per tal de convertir una CFG a Forma Normal de Chomsky, dins de la classe CFG es cridaria a la classe CNF, i aquesta directament retornaria la gramàtica ja convertida, permetent així l'aplicació directa de l'algoritme CKY.

Quant a la implementació del algoritme CKY, l'ús de programació dinàmica va ser fonamental, i ens va ser realment útil ja que vam refrescar els conceptes que havíem treballat a classe, i de la mateixa manera ens serviria el treball fet en aquesta pràctica per la preparació de l'examen final de l'assignatura.

L'extensió 2 del projecte també ens va suposar un repte, però realment ens va ser més fàcil d'implementar que la primera extensió, ja que teníem més clar com ho havíem de fer i no ens va exigir tant de temps, doncs per l'extensió 1 vam haver de buscar a llocs webs contrastats que expliquessin pas a pas com s'havia de fer correctament la transformació de qualsevol CFG a Forma Normal de Chomsky. Per aquesta segona extensió, la classe PCFG es va construir per tal de poder tractar amb gramàtiques probabilístiques i calcular la probabilitat de generació d'una paraula específica. Aquesta classe, doncs, utilitza una estructura similar a la classe CFG, però incorpora la gestió de probabilitats associades amb cada regla de producció.
La implementació de l'algoritme CKY per a PCFG també va requerir un tractament especial de les probabilitats. Cada cel·la de la taula CKY emmagatzema els símbols no terminals, junt amb les seves probabilitats associades. Això va requerir modificar l'algoritme CKY per calcular i comparar les probabilitats de manera correcta, assegurant-nos que sempre es guardi la probabilitat més alta per a cada símbol.
Finalment, vam validar les nostres implementacions amb diversos casos de prova, incloent-hi gramàtiques simples i complexes, tant per CFG com per PCFG. Això també ens ha estat útil a l'hora de detectar certs errors que no havíem tingut en consideració.

En conclusió, aquest projecte ens ha servit per aplicar coneixements teòrics en un context pràctic i continuar guanyant experiència amb la programació. Ha sigut interessant trobar una aplicació de la programació dinàmica que havíem treballat a classe, en l'àmbit del processament del llenguatge. Així doncs, creiem que els coneixements que hem obtingut en aquesta pràctica ens seran útils de cara al futur, ja sigui en l'àmbit del processament del llenguatge o en altres camps de la IA.