Universitat Politècnica de Catalunya

Grau en Intel·ligència Artificial

Programació i Algorísmia Avançada

Implementació de l'algorisme Cocke-Kasami-Younger (CKY)

Cai Selvas Sala **Roger Baiges Trilla**

31 de maig de 2024

FIB

Resum

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

El projecte tracta sobre l'aplicació de l'algorisme de Cocke-Kasami-Younger (CKY, conegut alternativament com a CYK) en gramàtiques lliures de context (CFGs) reprsentades en Forma Normal de Chomsky (CNF). Al llarg del document s'expliquen els detalls de la implementació presentada pels autors, les decisions i assumpcions preses i les conclusions extretes.

Índex

1 Glossari i definicions

- **Gramàtica lliure de context**: En anglès *Context-Free Grammar* (CFG), és una gramàtica formal les regles de producció de la qual es poden aplicar a un símbol no terminal independentment del seu context [1]. En aquest informe es distingeix entre dos tipus de gramàtiques: *Context-Free Grammar* (abreviat a CFG) i *Probabilistic Context-Free Grammar* (abreviat a PCFG), tot i que sovint s'utilitza el concepte de gramàtica lliure de context o CFG per referir-se a tots dos tipus de forma genèrica (cosa que no passa amb el terme PCFG).

- **Forma Normal de Chomsky**: En anglès *Chomsky Normal Form* (CNF). Es diu que una gramàtica lliure de context G està en Forma Normal de Chomsky si totes les seves regles de producció són de la forma:

$$A \to BC$$

$$A \to a$$

$$S \to \epsilon$$

  on A, B i C són símbols no terminals, a és un símbol terminal (un símbol que representa un valor constant), S és el símbol inicial, i ϵ denota la cadena buida. A més, ni B ni C poden ser el símbol inicial, i la tercera regla de producció només pot aparèixer si ϵ és en $L(G)$, el llenguatge generat per G [2].

  La utilitat principal de la Forma Normal de Chomsky és que permet una millora de l'eficiència dels algorismes que operen amb gramàtiques [3].

- **Parsing**: És el procés d'analitzar una cadena de símbols, ja sigui en llenguatges naturals, llenguatges de programació o estructures de dades, conformant-se a les regles d'una gramàtica formal.

  En l'àmbit de la lingüística computacional, el terme s'utilitza per referir-se a l'anàlisi formal per part d'un ordinador d'una frase o d'una altra cadena de paraules en els seus constituents, resultant en un arbre sintàctic que mostra la seva relació sintàctica entre ells, que també pot contenir informació semàntica [4]. En aquest document s'hi refereix també com a "parseig".

2 Introducció

L'algorisme de Cocke-Kasami-Younger (CKY, o també conegut alternativament com a CYK) és un algorisme de parsing per gramàtiques lliures de context (CFGs) publicat per Itiroo Sakai l'any 1961. Aquest algoritme es basa en la programació dinàmica i requereix que la gramàtica lliure de context (CFG) estigui representada en Forma Normal de Chomsky (CNF), ja que prova totes les possibilitats de dividir les produccions en dues subproduccions [5].

L'objectiu d'aquest projecte és aconseguir representar en Python un gramàtica lliure de context i implementar un algoritme CKY per tal de determinar correctament si un conjunt de paraules pot ser generat per la gramàtica.

Addicionalment, es volen afegir altres funcionalitats que permetin tractar de manera més còmoda, flexible i útil amb les gramàtiques lliures de context i amb l'algorisme de CKY.

En aquest document s'explica tota la implementació que s'ha realitzat, els detalls tècnics, les assumpcions realitzades i les decisions preses al llarg del projecte.

3 Estructura d'arxius i directoris

L'entrega d'aquest projecte conté diferents directoris i arxius, on cada un d'ells emmagatzema una part diferent del projecte per tal de que tot estigui ben organitzat i sigui comprensible. L'estructura d'arxius i directoris que s'hi pot trobar és la següent:

- **Directori *`/src`***: Conté tots els arxius del codi intern necessari pel funcionament correcte de la implementació de l'algorisme de CKY realitzada i per la declaració i tractament de gramàtiques lliures de context. Concretament, s'hi poden trobar els següents arxius:
  - **Arxiu *`cfg.py`***: Conté la definició de la classe `CFG` en Python, que permet representar les regles d'una gramàtica lliure de context mitjançant un diccionari. A més, proporciona múltiples mètodes per declarar de diferents maneres la gramàtica, obtenir-ne informació, transformar-la o visualitzar-la.
  - **Arxiu *`cky.py`***: Conté la definició de la classe `CKY` , on es troben tots els mètodes que permeten executar les diferents versions de l'algorisme de CKY (determinista o probabilístic), interactuant amb la classe `CFG` per tal d'obtenir informació sobre la gramàtica que s'utilitza en cada execució.
  - **Arxiu *`functions.py`***: Conté altres funcions que complementen els mètodes de les classes `CFG` , `CKY` i també el programa que es troba en l'arxiu *main.py*.
- **Directori *`/tests`***: Conté els arxius on hi ha els diferents jocs de prova per executar utilitzant programa definit en l'arxiu *main.py*.
- **Directori *`/results`***: Conté els arxius resultants de les execucions del programa definit en l'arxiu *main.py*.
- **Arxiu *`main.py`***: Conté el codi en Python que defineix un programa que, donat un input en el format dels jocs de prova (mencionat en la secció 6), crea la gramàtica lliure de context i aplica l'algorisme CKY a totes les paraules donades, guardant els resultats en el direcori */results*.
- **Arxiu *`examples.ipynb`***: És un arxiu de tipus *notebook* (*.ipynb*) que conté múltiples cel·les de codi que exemplifiquen de forma senzilla algunes de les funcionalitats que el la implementació realitzada permet.

Tots els detalls tècnics del codi implementat en cada arxiu es troben explicat més endavant en aquest informe.

4 Implementació del codi

En aquesta secció s'expliquen els detalls tècnics de la implementació dels diferents arxius que conformen el codi que permet el correcte funcionament de tot el projecte. És a dir, dels arxius que controlen les gramàtiques CFG, els algorismes CKY i altres funcions suplementaries.

Tot i que en aquest informe s'explica el funcionament general del codi, cada mètode o funció implementat en els arxius Python va acompanyat del seu corresponent *docstring*, on s'explica breument què fa la funció, quins paràmetres rep i què retorna. Addicionalment, també s'ha fet ús de *type hints* en tots els arxius per especificar clarament de quin tipus són les variables, paràmetres, retorns de funcions, etc. Finalment, també hi ha comentaris en certes parts del codi i es generen automàticament *warnings* (advertències) i errors amb missatges personalitzats durant les execucions per informar a l'usuari sobre possibles errors o consideracions a tenir en compte en el seu ús d'aquesta implementació. Tot això té com a objectiu facilitar la comprensió del codi a qualsevol persona que el vulgui revisar i fer que els usuaris que hi interactuïn ho facin de forma tan fàcil i còmode com sigui possible.

4.1 Arxiu *cfg.py*

En aquest arxiu es troba definida la classe `CFG`, que s'utilitza per representar gramàtiques lliures de context, tant probabilístiques com deterministes. A més, incorpora múltiples mètodes per poder generar aquestes gramàtiques des de diferents inputs, transformar-les, obtenir-ne informació, visualitzar-les, etc.

4.1.1 Assumpcions i decisions prèvies

A l'hora de realitzar la implementació de la classe `CFG` , s'han hagut de prendre múltiples decisions sobre com estructurar-la per tal de que fos efectiva, eficient, entenedora i fàcil de treballar-hi. A continuació es mencionen les diferents assumpcions o decisions prèvies a la implementació que s'han dut a terme:

- **Forma Normal de Chomsky**: Com que per l'algorisme de CKY i totes les aplicacions de la CFG que es faran en aquest projecte cal que la gramàtica es tigui en Forma Normal de Chomsky, s'ha determinat que en la classe `CFG` es guardarà la gramàtica sempre en CNF, convertint-la inicialment en cas d'haver sigut proporcionada en una altra forma. Aquesta conversió s'aplica només quan la gramàtica és determinista, però les gramàtiques probabilístiques s'han de proporcionar ja en CNF, ja que la seva transformació a CNF és bastant més complexa i no es requeria en cap apartat de l'enunciat d'aquesta pràctica.
- **Estructura interna de les regles de producció**: L'estructura escollida per les regles de producció en en l'interior de la classe `CFG` han estat els diccionaris, amb la següent estructura en funció del tipus de gramàtica:
  - **CFG**: {`A: {P_1, ..., P_n}, ..., Z: {P_1, ..., P_m}`}, on `A` i `Z` són símbols no terminals (representats com a string en Python) i cada `P_i` és una producció (també de

tipus string) lligada al símbol terminal que té com a clau en el diccionari. Cal notar que totes les produccions d'un mateix símbol terminal es troben contingudes en un conjunt (set de Python).

- **PCFG**: En aquest cas, es parteix d'un diccionari de la forma: `{A: {(P_1, Pb(P_1)), ..., (P_n, Pb(P_n))}, ..., Z: {(P_1, Pb(P_1)), ..., (P_n, Pb(P_n))}}`, on `A` i `Z` són símbols no terminals (representats com a string en Python), cada `P_i` és una producció (també de tipus string) lligada al símbol terminal que té com a clau en el diccionari i cada `Pb(P_i)` representa la probabilitat de que el símbol no terminal derivi la producció `P_i`. Cal notar que totes les produccions d'un mateix símbol no terminal es representen amb tuples `(P_i, Pb(P_i))` que es troben contingudes en un conjunt (set de Python).

  Posteriorment, aquest diccionari es divideix en un diccionari de regles (amb la mateixa estructura que per una CFG) i un diccionari de diccionaris de probabilitats amb l'estructura: `{A: {P_1: Pb(P_1), ..., P_n: Pb(P_n)}, ..., Z: {P_1: Pb(P_1), ..., P_n: Pb(P_n)}}` (utilitzant la mateixa nomenclatura ja comentada). Amb això, s'aconsegueix que l'accés a les probabilitat sigui molt més ràpid (simplement cal proporcionar les dues claus necessàries: el símbol no terminal i la producció). A més, també facilita la compatibilitat dels diferents mètodes de la classe, ja que tant per CFG com per PCFG s'utilitza un diccionari de regles de producció amb la mateixa estructura.

- **Símbols d'un únic caràcter**: En aquesta implementació, tots els símbols, tan terminals com no terminals, han d'estar representats per un sol caràcter. Concretament:
  - **Terminals**: Poden ser qualsevol lletra minúscula de l'abecedari o un nombre únic.
  - **No terminals**: Poden ser qualsevol lletra majúscula de l'abecedari.

  Aquesta decisió s'ha pres per la facilitat que proporciona Python per iterar sobre cadenes de caràcters (strings), de manera que amb aquesta implementació es pot iterar per cada caràcter d'una producció i es garanteix que en cada iteració s'està accedint a un símbol sencer (terminal o no terminal) de la gramàtica.

  Si la implementació no fos així, seria necessari emmagatzemar la producció en forma de llista/tupla de strings (on cada string seria un símbol i podria tenir qualsevol longitud). No obstant, aquesta alternativa suposaria problemes per el *hashing* de llistes (per la implementació amb sets) o per la immutabilitat de les tuples (ja que en alguns processos es modifiquen els elements de la producció). A més, l'usuari hauria de donar les produccions, o bé ja amb els símbols ja separats llista/tupla per cada producció, o bé amb els símbols separats per algun tipus de caràcter (per aplicar `split()` posteriorment), de manera que seria més laboriós i menys còmode.

  Per contrapartida, la gramàtica no permet generar paraules com a un sol element terminal (ja que contenen múltiples caràcters) i fer parsing d'una frase de paraules, de manera que es perden certes utilitats de l'algorisme CKY i les gramàtiques per àmbits relacionats amb el llenguatge natural. És a dir, aquesta implementació està limitada a que es faci parsing de paraules (no de frases), on es considera cada lletra de la paraula com un símbol terminal.

  Addicionalment, el nombre de símbols no terminals que pot utilitzar la gramàtica està limitat a les lletres de l'abecedari més algunes lletres gregues que s'han afegit [1]. Això pot portar

[1] Concretament, l'alfabet de símbols no terminals disopnibles és: SABCDEFGHIJKLMNOPQRTUVWXYZΓΔΘΛΞΠΣΦΨΩ.

problemes en gramàtiques de grans dimensions, però no hauria de resultar ser cap inconvenient per la gran majoria de les gramàtiques que es tractaran.

Finalment, cal mencionar que l'objectiu d'aquesta pràctica, segons l'enunciat proporcionat, no és en cap cas representar grans gramàtiques ni que aquestes estiguin pensades per tasques relacionades amb el llenguatge natural o altres àmbits de major complexitat, sinó que es centra en l'algorisme CKY i la seva implementació funcional. Per tant, la implementació d'aquesta gramàtica s'ha fet així per treballar més còmodament i poder tenir un codi substancialment més entenedor. En cas de que calgués tenir una gramàtica capaç de representar el que s'ha comentat, les modificacions en l'estructura del codi serien nul·les o molt poques, a part de la simple modificació sintàctica de certes línies i *type hints*.

- **Paraula buida**: Per representar la paraula buida s'ha decidit utilitzar la lletra grega ϵ, una pràctica comuna en l'àmbit de la teoria de llenguatges. Això s'aplica només a la representació interna de la gramàtica, però la classe `CFG` permet que l'usuari introdueixi la paraula buida en forma de string buida de Python.
- **Suma de probabilitats**: La suma de totes les probabilitats de les produccions de cada símbol terminal ha de ser exactament igual a 1. Aquesta assumpció és evident i és una de les condicions bàsiques de les PCFGs, però tot i així s'ha decidit especificar de forma explícita.

4.1.2 Dependències

Per tal de que la classe `CFG` funcioni correctament, cal importar les següents dependències:

- `combinations`: Funció de la llibreria `itertools` per permet obtenir totes les combinacions d'una certa mida donat un conjunt d'elements.
- `deque`: Tipus d'objecte de la llibreria `collections` que permet crear cues en Python.
- `defaultdict`: Subclasse dels diccionaris de Python que es troba en la llibreria `collections` i permet crear diccionaris amb valors per defecte.
- `dynamic_round`: Funció de l'arxiu *functions.py* que permet arrodonir probabilitats de forma dinàmica (sense una mida fixa). Serveix principalment per evitar les imperfeccions de les operacions amb nombres amb punt flotant, tal i com s'explica en l'apartat 4.4.1.
- `os`: Llibreria per poder navegar els arxius del sistema.

4.1.3 Inicialització

Com s'ha mencionat, internament les regles de producció es guarden en forma de diccionari. No obstant, l'usuari té múltiples alternatives a l'hora de proporcionar les regles a la classe `CFG` . Concretament, en la inicialització de la classe (mètode constructor `__init__()`), l'usuari pot controlar els següents paràmetres:

- **Paràmetre** `from_dict`: Permet proporcionar les regles en forma de diccionari, amb el mateix format que s'ha mencionat anteriorment en les assumpcions de la implementació. És la manera d'inicialitzar la classe més "directa", ja que la classe no haurà de convertir el format de les regles de producció, però requereix que s'especifiqui el paràmetre `probabilistic` i es recomana que també s'especifiqui `start_symbol`.

- **Paràmetre** `from_text`: Permet proporcionar les regles de producció de la gramàtica en forma de text (string). El format a seguir, a part de les condicions ja establertes prèviament, és el següent:

  - La primera línia del text ha de contenir només la paraula "CFG" o la paraula "PCFG" indicant el tipus de gramàtica (determinista o probabilística, respectivament),
  - El símbol no terminal de la primera regla de producció que es declari ha de ser el símbol inicial de la gramàtica.
  - Cada regla de producció ha d'estar en una línia diferent i seguint exactament el format següent (depenent del tipus de gramàtica):
    - **CFG**: $A -> P_1 \mid \ldots \mid P_n$, on cada P_i és una producció (string de símbols). És a dir, els caràcters " $->$" han de separar el símbol no terminal de les seves produccions i el caràcter " | " ha de separar les produccions.
    - **PCFG**: $A -> P_1[Pb(P_1)] \mid \ldots \mid P_n[Pb(P_n)]$, on cada P_i és una producció (string de símbols) i cada $Pb(P_i)$ és la probabilitat de que el símbol no terminal generi la producció. És a dir, els caràcters " $->$" han de separar el símbol no terminal de les seves produccions, el caràcter " | " ha de separar les produccions i cada producció ha d'anar acompanyada de la seva probabilitat entre els caràcters " [ ] ".

- **Paràmetre** `from_file`: Permet especificar la ruta (*path*) a un arxiu que conté la definició, seguint el format de gramàtica a partir de text (vist en el punt anterior), per tal de llegir el seu contingut i importar les regles de la gramàtica establerta. El mètode que converteix l'input de text (ja sigui proporcionat directament o des d'un arxiu) a un diccionari (la representació interna que utilitza la calasse `CFG`) és `read_rules_from_file_or_text()`.

- **Paràmetre** `from_words`: Permet generar una gramàtica determinista totalment des de zero que generi exactament les paraules proporcionades en un set de Python. El mètode que genera les regles de la gramàtica a partir de les paraules proporcionades és `generate_rules_from_words()`.

- **Paràmetre** `start_symbol`: Permet especificar el símbol no terminal inicial de la gramàtica. Només es considera si les regles de producció s'han proporcionat mitjançant el paràmetre `from_dict`.

- **Paràmetre** `probabilistic`: Permet especificar si la gramàtica és probabilística. Només es considera, i és obligatori que s'especifiqui, si les regles de producció s'han proporcionat mitjançant el paràmetre `from_dict`.

- **Paràmetre** `EMPTY`: Permet canviar el símbol per representar la paraula buida. Els símbols disponibles són ϵ, ε i λ (ϵ per defecte). Això no afecta a la correcta interpretació de la paraula buida proporcionada com a string buida (`''''`) per part de l'usuari. Tots els caràcters en minúscula excepte l'especificat `EMPTY` seran interpretats com símbols terminals comuns en la gramàtica. En cas de que la paraula buida es proporcioni com a string buida de Python (el mètode més còmode), aquest paràmetre només té efectes visuals a l'hora de representar la gramàtica.

Com a condició general i per evitar confusions, es requereix que només s'especifiqui un dels paràmetres `from_dict`, `from_text`, `from_fil` o `from_words`.

Si la gramàtica es proporciona des de qualsevol paràmetre, excepte `from_words`, els símbols terminals i no terminals es calculen automàticament en el mètode `find_symbols()` mitjançant un anàlisi de les produccions i fent múltiples comprovacions que garantitzin que la gramàtica segueix el format correcte. En el cas de `from_words` no és necessari, ja que la gramàtica s'ha creat des de zero i tots els símbols ja estan correctament classificats.

També per comprovar que s'hagin proporcionat les regles de producció en un format vàlid, els mètodes `assert_valid_format()` i `improve_format()` ajuden a detectar possibles errors i convertir la gramàtica en el format correcte per la seva representació interna. Si aquests mètodes detecten alguna anomalia irresoluble en el format, generen un error amb un missatge que informa a l'usuari sobre què l'ha provocat, permetent així que pugui ser solucionat fàcilment.

Si les regles de producció es proporcionen mitjançant `from_dict` i no s'especifica `start_symbol`, s'intentarà trobar el símbol inicial automàticament mitjançant un anàlisi de totes les produccions i símbols no terminals que es fa en el mètode `find_start_symbol()`.

Per altra banda, si es tracta d'una gramàtica probabilística, el mètode `split_probabilities()` s'encarrega de generar els dos diccionaris separats que contenen, respectivament, les regles i les probabilitats.

Finalment, una vegada ja es té la gramàtica proporcionada per l'usuari (o generada automàticament, en el cas de `from_words`), es comprova si la gramàtica està en Forma Normal de Chomsky (CNF) mitjançant el mètode `is_cnf()` i, si està en CNF, no es fa res més. En cas contrari, si es tracta d'una gramàtica probabilística s'acaba l'execució amb un error, ja que les PCFGs s'han de donar ja en CNF; però si es tracta d'una gramàtica determinista es procedeix a la seva transformació a Forma Normal de Chomsky a través del mètode `to_cnf()`.

4.1.4 Mètodes principals

La classe `CFG` conté molts mètodes per realitzar diferents tasques. No obstant, a part dels mètodes utilitzats en la inicialització, que ja s'han comentat, també hi ha un seguit de mètodes de gran importancia per la implementació realitzada:

- `to_cnf()`:: Aquest mètode serveix per convertir una gramàtica no determinista a Forma Normal de Chomsky (CNF). Aquesta conversió es fa mitjançant els següents passos [2]:

    1. **Eliminar el símbol inicial de les produccions**: Es fa mitjançant el mètode privat `__remove_start_symbol_from_rhs()` i es tracta d'introduir un nou símbol inicial S_0 amb la producció $S_0 \rightarrow S$, on S és el símbol inicial previ. D'aquesta manera, el nou símbol inicial S_0 no apareix en cap producció.
    2. **Eliminar els símbols terminals acompanyats de no terminals**: Es fa mitjançant el mètode privat `__remove_productions_with_nonsolitary_terminals()` i tracta d'eliminar les produccions on hi ha símbols terminals acompanyats de símbols no terminals, ja que els símbols terminals han de produïrse sols per tal de complir amb les regles de

la Forma Normal de Chomsky. Per fer-ho, per cada regla $A \to X_1 \dots a \dots X_n$ (on a és un símbol terminal i A, X_i són símbols no terminals), crear un nou símbol no terminal N_a tal que $N_a \to a$ i modificar la regla original a $A \to X_1 \dots N_a \dots X_n$.

3. **Eliminar les produccions amb més de dos símbols no terminals**: Una vegada ja no hi ha produccions de longitud major a 1 que continguin símbols terminals (ja que s'han eliminat en el pas anterior), cal reduir les produccions amb més de dos símbols no terminals a només dos. Es fa mitjançant `__remove_long_nonterminal_productions()` i tracta de descomposar les regles de la forma $A \to X_1 X_2 \dots X_n$ (on A, X_i són tots símbols no terminals) en regles $A \to X_1 A_1$, $A_1 \to X_2 A_2$, $\dots$, $A_{n-2} \to X_{n-1} X_n$ (on A_i són sous símbols no terminals).
4. **Eliminar regles amb ϵ**: Es fa en el mètode `__remove_epsilon_productions()` i tracta d'eliminar les regles $A \to \epsilon$ (on A no és el símbol inicial S_0). Aquest mètode és bastant complex i segueix un seguit d'altres passos:
   (a) **Símbols anul·lables**: El primer que cal fer és trobar els símbols no terminals anul·lables mitjançant el mètode `get_nullable_nonterminals()`. Un símbol no terminal A és anul·lable si existeix la regla $A \to \epsilon$ o $A \to X_1 \dots X_n$ (on tots els X_i són símbols no terminals anul·lables) [6].
   (b) **Reemplaçar per totes les combinacions**: Una vegada es tenen els símbols anul·lables, per cada símbol anul·lable A (incolent el símbol inicial S_0, en cas de ser anul·lable) amb la regla de producció $A \to X_1 \dots X_n$ es crea una gramàtica intermitja en la que aquesta mateixa regla es reemplaça per totes les possibles combinacions dels símbols $X_1 \dots X_n$ (ocultant o no diferents X_i en cada combinació).
   (c) **Eliminar les produccions de ϵ**: Finalment, s'eliminen totes les produccions del tipus $A \to \epsilon$ (on A és un símbol no terminal diferent del símbol inicial S_0. L'únic símbol no terminal que pot generar la paraula buida (ϵ) és el símbol inicial S_0, i l'ha de generar si i només sí alguna de les produccions de S_0 pot derivar ϵ.
5. **Eliminar produccions d'un únic símbol no terminal**: Finalment, mitjançant el mètode `__remove_unit_productions()` s'eliminen totes les produccions de la forma $A \to B$ (on A, B són ambdós símbols no terminals). Per fer això, es segueixen els següents passos:
   (a) **Trobar els parells unitaris**: Mitjançant el mètode `__find_unit_pairs()` es troben tots els parells (A, B) tals que existeix la regla de producció $A \to B$.
   (b) **Trobar el tancament unitari**: El tancament unitari d'un símbol no terminal A és el conjunt de símbols no terminals que es poden derivar des d'A seguint únicament regles unitàries.
   (c) **Substitució de regles unitàries**: Una vegada es té el tancament unitari de cada símbol no terminal A, per cada símbol no terminal B en aquest tancament unitari s'afegeixen a les produccions d'A totes les produccions de B (excepte les unitàries).
   (d) **Eliminar símbols no utilitzats**: Finalment, `__get_reachable_nonterminals()` es crida per obtenir tots els símbols no terminals accessibles a partir del símbol inicial S_0, i els que no siguin accessibles s'eliminen de la gramàtica, ja que mai podran ser derivats (són completament inútils).

Una vegada s'han realitzat tots aquests passos, la gramàtica es troba ja en Forma Normal de Chomsky (CNF) i pot ser utilitzada per l'algorisme CKY o altres propòsits.

- `get_or_create_nonterminal()`: A partir d'una producció o produccions retorna, si existeix, el símbol que la produeix mitjançant una crida a `get_lhs()`. Si no existeix, crea el símbol no terminal mitjançant `create_nonterminal()` i el retorna. Aquest mètode és molt utilitzat en la conversió a CNF (`to_cnf()`) o en la creació de la gramàtica a partir d'un conjunt de paraules `paràmetre from_words a __init__()`, ja que permet crar nous símbols no terminal només quan és necessari i reutilitzar els ja creats per evitar tenir-ne duplicats.

- `get_lhs()`: Aquest mètode permet, a partir d'una producció o conjunt de produccions, obtenir el símbol no terminal que la produeix. A més, amb el paràmetre `exact_match` es pot controlar si el símbol ha de produir exactament la producció donada o si aquesta es permet que sigui un subconjunt de les produccions que genera el símbol no terminal. Principalment es crida des del mètode `get_or_create_nonterminal()`.

- `generate_words()`: Aquest mètode forma part de les implementacions addicionals afegides a aquest projecte i permet obtenir totes les paraules que la gramàtica pot generar, fins a una certa mida determinada pel paràmetre `max_length`.

  A més, en les gramàtiques probabilístiques també es retornen les probabilitats de que la gramàtica generi cada una de les paraules possibles. En aquest cas, el paràmetre opcional `round_probabilities` permet arrodonir les probabilitats de forma dinàmica mitjançant la funció `dynamic_round()` (explicada en l'apartat 4.4.1), evitant les imprecisions que sovint apareixen degut a les múltiples operacions amb nombres representats en punt flotant.

- `__str__()`: Aquest mètode serveix per representar la gramàtica de forma entenedora per l'usuari en forma de text. Per fer-ho, s'ordenen els símbols no terminals en ordre entenedor i es proporcionen totes les regles i característiques que la gramàtica té. Es crida automàticament en la inicialització (per veure com s'ha interpretat inicialment la gramàtica) i després de la conversió a CNF (per veure com s'ha convertit).

4.2 Arxiu *cky.py*

En aquest arxiu es troba definida la classe `CKY`, que proporciona els mètodes necessaris per executar l'algorisme de Cocke-Kasami-Younger (CKY) en una gramàtica lliure de context (representada com una instància de la classe `CFG`).

4.2.1 Dependències

Les dependències necessàries per a la classe `CKY` són les següents:

- `CFG` : Classe definida a l'arxiu *cfg.py* que serveix per representar una gramàtica lliure de context (CFG) en Forma Normal de Chomsky (CFN), tal i com s'explica en al llarg de l'apartat 4.1.

- `defaultdict`: Subclasse dels diccionaris de Python que es troba en la llibreria `collections` i permet crear diccionaris amb valors per defecte.

- `dynamic_round`: Funció de l'arxiu *functions.py* que permet arrodonir probabilitats de forma dinàmica (sense una mida fixa). Serveix principalment per evitar les imperfeccions de les operacions amb nombres amb punt flotant, tal i com s'explica en l'apartat 4.4.1.
- `networkx`: Llibreria per crear i visualitzar gràfics i xarxes, utilitzada per representar els arbres de parseig.
- `matplotlib.pyplot`: Llibreria per visualitzar gràfics, especialment per mostrar els arbres de parseig.

4.2.2 Inicialització

Pel que fa a la inicialització d'aquesta classe simplement és rep com a argument una gramàtica; aquesta ja ha estat prèviament a CNF si ho requeria com s'ha vist en l'apartat 4.1.4.

Aquesta classe no executa directament automàticament sinó que espera a que l'usuari cridi algun dels mètodes com ara l'encarregat d'aplicar l'algorisme del `CKY` o el visualitzador.

4.2.3 Mètodes principals

En aquesta secció s'explicaran els mètodes i funcions auxiliars utilitzades per aquesta classe alhora que les decisions i justificacions realitzades.

- `set_grammar()`: Aquest mètode públic permet establir de nou la gramàtica que s'utilitzarà per fer el parseig.
- `parse()`: Aquest mètode també públic s'encarrega de realitzar el parseig d'una paraula donada utilitzant l'algorisme `CKY` . Com a paràmetres rep una paraula, la instància de `CFG` , si s'arrodoneixen les probabilitats i finalment si es volen visualitzar les solucions.

  El primer que fa la funció és realitzar un *assert* de si s'ha donat una gramàtica com a paràmetre o si, per defecte, s'utilitza la propia gramàtica que guarda la classe. Posteriorment es comprova que la gramàtica pertany a una instància de la classe `CFG` retornant un error en cas contrari.

  Per tal de diferenciar quin algorisme de parsing utilitzar, entre determinista i probabilístic, s'utilitza el mètode `is_probabilistic()` de la classe `CFG` . En el cas de ser una gramàtica probabilística es cridarà al mètode privat `__parse_probabilistic()` i, en cas contrari, es fa la crida al mètode `__parse_deterministic()`.
- `__parse_deterministic()`: Aquest mètode privat implementa el parseig `CKY` per gramàtiques determinístiques (CFG), retornant si la paraula està en el llenguatge generat per la gramàtica i els possibles arbres de parseig.

  Per fer-ho primer es guarda la llargada de la paraula donada anomenada n i s'inicialitzen les taules *table* i *backpointers*. Ambdues taules tenen una mida de $\frac{n}{2}$ degut a que tan sols s'utilitza la meitat de la matriu per tal de millorar l'eficiència.

  Pel que fa a la variable *table* aquesta es defineix incialment com una llista de *sets()* on cada posició de la *table[i][j]* representa el conjunt de símbols no terminals que poden generar la

subcadena de la paraula que comença a l'índex j i té longitud $i+1$. Així doncs, la *table[0][0]* representaria els símbols no terminals que poden generar el primer caràcter de la paraula, la *table[1][0]* els que poden formar els dos primers caràcters de la paraula i *table[1][1]* els que poden formar el segon caràcter de la paraula.

La variable *backpointers* és una matriu que s'utilitza per rastrejar com es va generar cada subcadena permetent la reconstrucció dels arbres d'anàlisi (parse trees) després que s'hagi completat la matriu *table*. En aquest cas *backpointers[i][j]* conté un diccionari on cada clau és el mateix no terminal que en *table* i els seus valors indiquen els dos altres no símbols terminals que generen i posteriorment dos índexs els quals indiquen quina part de la *string* genera cada no terminal. En el cas dels elements de la taula que es troben en la diagonal, és a dir, quan *i = 0*, enlloc de guardar els índexs simplement guardarà el símbol terminal que generen.

Per tal d'omplir les taules i començar amb el propi algorisme cal començar per els casos base que com ja es pot intuir correspon als elements de la diagonal. En el cas de *table* simplement es guarda els símbols no terminals capaços de generar cada caràcter i, en la taula *backpointers* es guarda el mateix símbol però, en aquest cas, també el caràcter que genera.

Després d'omplir la diagonal l'algorisme procedeix a omplir tota la taula. Per a cada possible longitud de subcadena *span* i per a cada possible inici de subcadena *start* es considera cada possible punt de partició de la subcadena *split*. L'algorimse verifica si un no terminal *lhs* pot generar una producció la qual consiteixi en dos no terminals que es troben en les cel·les *left_cell* i *right_cell*. Aquetes cel·les representen les posicions de *table[split - start][start]* i *table[start - 1 - (split+1)][split+1]* respectivament indicant totes les combinacions que poden crear les *substrings* a l'esquerra de la partició (*split*) i a la dreta.

Finalment un cop omplerta tota la taula si hi ha el símbol inicial en la cel·la superior *table[n-1][0]* significa que la paraula es pot generar mitjançant la gramàtica.

En el cas que es pugui generà es cridarà a la subfunció `build_trees()` la qual recursivament anirà generant de nous tots els camins que porten a la paraula a partir del símbol inicial. Posteriorment això es podrà visualitzar amb el mètode `visualize_parse_trees()`.

Considerem una paraula de longitud 3, `word = "abc"`, i una gramàtica en CNF. Es descriu el procés d'ompliment de les matrius *table* i *backpointers*.

1. **Inicialització:**
   - Es creen *table* i *backpointers* amb dimensions adequades per la paraula donada.
   - $table = [[set(), set(), set()], [set(), set()], [set()]]$
   - $backpointers = [[, ,], [,], []]$
2. **Ompliment de la Diagonal (Terminals):**
   - Es processa cada caràcter de la paraula i es troben els no terminals que poden generar aquests caràcters segons les produccions de la gramàtica.
   - Per exemple, si $A \to a$, $B \to b$, i $C \to c$ són produccions de la gramàtica:
     - *table[0][0]* = $\{A\}$
     - *table[0][1]* = $\{B\}$
     - *table[0][2]* = $\{C\}$
   - Simultàniament, es guarden els punters de retorn:
     - *backpointers[0][0]* = $\{A$: *[('a',)]*$\}$
     - *backpointers[0][1]* = $\{B$: *[('b',)]*$\}$

    - *backpointers[0][2]* = {*C: [('c',)]*}
3. **Ompliment de la Resta de la Matriu (No Terminals):**
   - Es processa cada subcadena de longitud creixent, combinant els resultats de subcadenes més curtes.
   - Per exemple, per la subcadena de longitud 2 que comença a $j = 0$ (i.e., "ab"):
     - Si $D \rightarrow AB$ és una producció de la gramàtica:
       - *left_cell = table[0][0]*
       - *right_cell = table[0][1]*
       - *table[1][0]* = {*D*}
       - *backpointers[1][0]* = {*D: [('A', 'B', 0, 1)]*}
4. **Verificació del Símbol Inicial:**
   - Després d'omplir tota la matriu, es verifica si el símbol inicial de la gramàtica es troba a la cel·la *table[n-1][0]*.
   - Si el símbol inicial es troba en aquesta cel·la, la paraula pot ser generada per la gramàtica.

- `__parse_probabilistic()`: Aquest mètode privat implementa el parseig `CKY` per gramàtiques probabilístiques (PCFG), retornant si la paraula està en el llenguatge generat per la gramàtica, la probabilitat total de la paraula i els arbres de parseig amb les seves probabilitats.

  Cal mencionar que hi ha diferents variants de l'algorisme on es pot retornar la probabilitat del camí màxima o directament, com en el nostre cas, la probabilitat total de la paraula formada per la suma de totes les probabilitats dels camins. Hem considerat que en aquest cas podria ser més interessant tornar la probabilitat total i no tan sols la del camí més probable tot i que aquesta última és la que s'acostuma a utilitzar per tasques de processament del llenguatge humà (**NLP**). En cas de voler obtenir el valor del camí màxim també es pot fer molt fàcilment mirant els arbres d'anàlisi.

  Aquest algorisme és pràcticament idèntic al determinista excepte els següents canvis:

  - **Variable table**: Aquesta variable utilitza *defaultdict(float)* enlloc de *set()* degut a que ara a banda de voler emmagatzemar els símbols no terminals que ens porten a la creació de les subcadenes també ens interessa controlar en quina probabilitat ho fan i en cas de que un element no es trobi en el diccionari retornaria el valor de 0 ajudant així a controlar possibles errors o estalviar mirar si un element es troba en les claus o no.
  - **Variable backpointers**: Aquesta variable gairebé no es veu afectada sinó que simplement en el cas inicial on es guarden la diagonal s'hi afegeix la probabilitat de cada caràcter individual.
  - **Ompliment de les diagonals**: Per tal d'omplir les diagonals es guarden les probabilitats també a banda dels símbols no terminals en ambdues taules dinàmiques.
  - **Ompliment de les taules**: A banda de verificar els símbols no terminals els quals contenen altres dos símbols no terminals capaços de generar la *substring* també es guarda la probabilitat associada a la producció corresponent que consisteix en totes les sumes de probabilitats del símbol no terminal actual multiplicat per les probabilitats de les cel·les esquerra i dreta (explicades en l'apartat anterior).

  - **Resultat**: Pel que fa al resultat, el valor de *table[n-1][0]* equivaldrà a la probabilitat de generar la paraula donada la gramàtica en CNF.
  - **Recuperació dels arbres d'anàlisi:** Es recuperen tots els arbres d'anàlisi juntament amb les seves probabilitats per separat.

- `__transform_probabilistic_to_deterministic()`: Aquest mètode privat transforma una llista d'arbres de parseig amb les seves probabilitats a una llista d'arbres d'anàlisi determinístics, eliminant les probabilitats associades a cada node.

  Primer calcula la suma de les probabilitats finals dels arbres després utilitza una funció auxiliar per eliminar les probabilitats dels nodes de cada arbre i finalment retorna la probabilitat total.

- `visualize_parse_trees()`: Aquest mètode públic visualitza els arbres d'anàlisi utilitzant les biblioteques `networkx` i `matplotlib`. Rep com a paràmetres una llista d'arbres d'anàlisi amb les seves probabilitats, la paraula que s'ha analitzat, i un indicador de si els arbres són probabilístics.

  Si els arbres són probabilístics, primer els transforma en arbres determinístics utilitzant la funció `visualize_parse_trees()` explicada anteriorment. Després, crea un gràfic utilitzant networkx i matplotlib, on cada node i aresta es representa visualment. Els nodes i arestes es disposen en un layout multipartit per mostrar la jerarquia dels arbres d'anàlisi.

  En el cas que hi hagi més d'un camí possible es representaran tants grafs no connexos en la mateixa imatge com calgui mantenint un ordre clar per facilitar la visualització.

  És important notar que el visualitzador pot no funcionar adequadament en certs casos. Per a casos senzills, es transforma en un arbre per facilitar la visualització. No obstant això, per estructures més complexes, la visualització pot resultar menys clara i més difícil d'entendre encara que probablement sigui millor que mirar directament les pròpies llistes dels arbres d'anàlisi.

  Considerem la següent gramàtica PCFG donada:

```
prules = {
    'S': {('AB', 0.9), ('BC', 0.1)},
    'A': {('BA', 0.5), ('a', 0.5)},
    'B': {('CC', 0.7), ('b', 0.3)},
    'C': {('AB', 0.6), ('a', 0.4)}
}
```

  Analitzant la paraula *"aaab"* amb aquesta gramàtica, s'ha generat el següent arbre d'anàlisi:

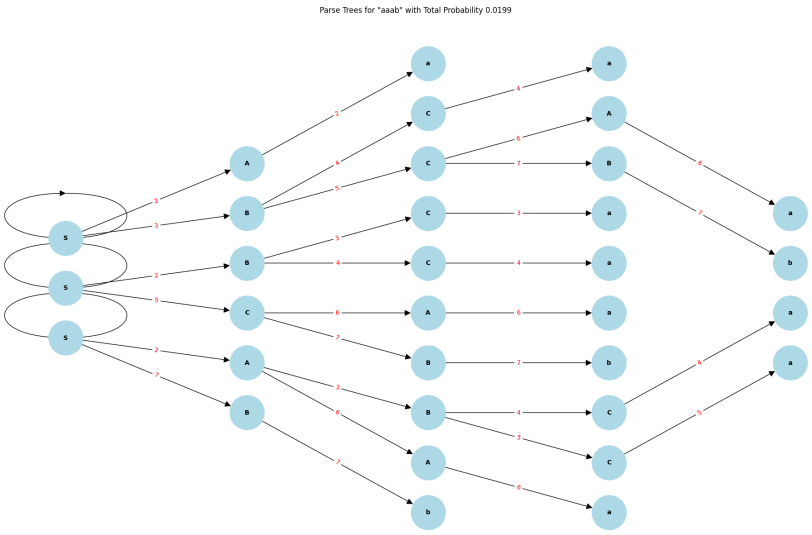

Figura 1: Imatge generada pel mètode `visualize_parse_trees()`

4.3 Mètodes especials

A banda dels mètodes mencionats anteriorment també considerarem els mètodes especials degut a la seva utilitat.

- `__init__()`: Inicialitza el parser CKY amb una gramàtica donada en forma normal de Chomsky (CNF) (mencionat prèviament en 4.2.2)
- `__call__()`: Permet que l'objecte de la classe CKY sigui invocable com una funció, passant arguments a la funció `parse`, que és la principal.

4.4 Arxiu *functions.py*

Aquesta secció correspon a les funcions auxiliars utilitzades en el projecte les quals no pertanyien a cap classe però aportaven un cer valor.

4.4.1 Funció `dynamic_round()`

Aquesta funció permet arrodonir de forma dinàmica un nombre introduït mitjançant el paràmetre `value`. En la implementació d'aquest programa s'utilitza per arrodonir les probabilitats proporcionades pel mètode `generate_words()` de la classe `CFG` o el mètode `__parse_probabilitsic()` de la classe `CKY` . Aquest arrodoniment té l'objectiu de solucionar els errors de precisió que introdueixen les operacions amb valors en punt flotant de Python.

Inicialment, el valor es converteix a text (string) per poder treballar-hi més còmodament. Seguidament, es separa la part entera de la part decimal (separats per un punt "."); i la part decimal de la part amb exponent (si n'hi ha, separada per "e"). Una vegada s'han separat, es busca en la part decimal un patró de tants caràcters "0" o "9" com indiqui el paràmetre `threshold` (per defecte 3), ja que són els valors que indiquen que el valor ja pot ser clarament arrodonit cap amunt (en cas de tractar-se de "9") o cap avall (en cas de tractar-se de "0").

Una vegada es troba el patró, s'aplica la funció de Python `round` amb paràmetre `ndigits` igual a l'índex en el que el patró de caràcters "0" o "9" consecutius comença, de manera que s'arrodoneixi tota la part que segueix a aquests nombres.

Amb aquesta implementació s'aconsegueix arrodonir satisfactòriament diferents nombres amb decimals en funció dels seus dígits, sense haver de fer-ho en una mida fixa (que pot no ser tan precís degut a que hi ha nombres amb escales molt diferents). Especialment en la implementació d'aquest projecte, és útil per comparar les probabilitats generades per `generate_words()` (de la calsse `CFG`) i `__parse_probabilistic()` (de la classe `CKY`) , ja que sinó els resultats no són exactament iguals (degut a les imprecisions de les operacions en punt flotant).

L'únic cas en el que aquesta funció pot arrodonir un nombre malament és si, per exemple, amb valor `threshold = 3`, es cridés amb `value = 0.000002459993452`, ja que els tres "9" consecutius es detectarien com el patró mencionat i el nombre retornat seria `0.00000246`, perdent precisió en el nombre. Aquest tipus d'error és més probable a mesura que valors de `theshold` són més petits, però amb valors majors o iguals a 3 es considera que és molt poc probable que hi hagi nombres amb patrons com el d'aquest exemple, de manera que pràcticament en tots els casos aquesta funció és útil.

Finalment, cal destacar que, en el mateix exemple mostrat (amb `value = 0.000002459993452`), els primers caràcters "0" consecutius de la part decimal no es tenen en compte, ja que es considera que el nombre no ha "començat" encara. Això s'ha decidit implementar així ja el càlcul de les probabilitats es fa mitjançant multiplicacions o sumes de nombres positius, de manera que si en tot moment hi ha hagut sumes o multiplicacions de "subprobabilitats" amb valor 0 no hauria d'haver derivat en cap imprecisió decimal. Per tant, per molt petit que sigui el valor, s'ha considerat que no seria correcte arrodonir-lo a 0. Cal mencionar que aquest tractament especial no es fa per caràcters "9" consecutius a l'inici de la part decimal, ja que en aquest cas si que es considera que el més probable és que es tracti sempre d'una imprecisió.

Exemples:

```
>>> dynamic_round(3.14159, 3)
3.14159

>>> dynamic_round(0.0005163749999999999, 3)
0.000516375

>>> dynamic_round(2.2680000000000003e-05, 3)
2.268e-05
```

4.4.2 Funció `split_input()`

Aquesta funció rep un dels paràmetres `file_path` o `file_text` amb el *path* a un arxiu o directament el contingut en text de l'arxiu, respectivament. En cas de proporcionar-se el *path*, s'obté el contingut de l'arxiu i es tracta com si s'hagués donat directament el text.

Aquest text s'espera que segueixi el format dels jocs de prova que es passen al programa principal (definit en l'arxiu *main.py*) i que s'han especificat en aquest document en l'apartat 5.1.

Una vegada es té el text, la funció separa línia per línia el text per tal de dividir-lo en la declaració de les regles de la gramàtica i el conjunt de paraules a parsejar.

Quan ja s'ha aconseguit separar satisfactòriament el text d'entrada, simplement es retorna el text definint la gramàtica i el conjunt de paraules a parsejar.

5 Programa principal

Com ja s'ha mencionat prèviament, el l'arxiu *main.py* es troba el codi del programa principal requerit per aquesta pràctica, que ha de ser capaç d'executar automàticament l'algorisme de CKY (tant determinista com probabilístic) pels diferents jocs de prova. És a dir, donada una gramàtica i un conjunt de paraules, aplicar l'algorisme amb la gramàtica proporcionada a totes les paraules.

5.1 Input

Els arxius que llegeix el programa han de seguir l'estructura dels jocs de prova, on s'especifica el tipus de gramàtica (determinista o probabilística), les seves regles de producció i les paraules que s'han de passar a l'algorisme de CKY.

Concretament, la primera línia ha de contenir CFG o PCFG (determinista o probabilística, respectivament).

A continuació, cada línia ha de contenir totes les produccions d'una mateix símbol no terminal separades pel caràcter " | ". A més, si es tracta d'una gramàtica probabilística, cada producció ha d'anar acompanyada de la seva probabilitat de ser produïda per aquell símbol no terminal, i s'ha d'especificar a l'interior dels caràcters " [ ] ".

Per altra banda, el símbol no terminal ha d'estar separat de les produccions mitjançant els caràcters " $->$ " (com una "fletxa"), i el primer símbol no terminal que aparegui en les regles de producció es considerarà el símbol inicial de la gramàtica.

Posteriorment, quan ja s'han declarat totes les regles, les paraules a les que se'ls ha d'aplicar l'algorisme de CKY han d'estar separades per línies, espais o tabulacions.

Finalment, cal mencionar que les línies buides o que comencin amb el caràcter "#" seran ignorades pel programa per tal de permetre comentaris en l'arxiu d'entrada i més flexibilitat en el format.

L'arxiu d'entrada serà preguntat a l'usuari pel programa i s'haurà d'especificar el nom de l'arxiu del directori */tests* per tal de que funcioni correctament.

A continuació es es poden veure exemples d'un input en format vàlid tant per una gramàtica determinista com per una probabilística.

Exemple input CFG (arxiu *racso_cfg_det10.txt*):

```
# Exercise 10: Non-ambiguous CFG for a^i b^j c^k | i = j + k
# Write a non-ambiguous CFG generating the words of the form a^i b^j c^k such
# that the number of a's coincides with the number of b's plus the number of c's.

CFG
S -> aSc | aKb | ϵ
K -> aKb | ϵ
```

```
# True (10)
ϵ
ac
ab
aacc
aabc
aaabbc
aaaabbcc
aaaccc
aaabbc
aaaabbbc

# False (10)
a
b
c
abc
abbc
aabcc
aacbb
abcc
abcbc
aaabbbccc
```

Exemple input PCFG (arxiu *prob1.txt*):

```
PCFG
S -> AB [0.9] | BC [0.1]
A -> BA [0.5] | a [0.5]
B -> CC [0.7] | b [0.3]
C -> AB [0.6] | a [0.4]

# True (8)
ab
bab
aabab
aaaaab
abbaba
bababa
aaa
bbaaa

# False(8)
ϵ
a
b
ac
```

```
bbb
abb
baa
abba
```

5.2 Implementació

La implemencaticó d'aquest programa és relativament senzilla i es dedica a fer crides a funcions definides en l'arxiu *functions.py* i a les classes `CFG` i `CKY` .

Inicialment, el programa demana a l'usuari que especifiqui el nom d'un arxiu que estigui en el directori */tests*. L'usuari pot escriure el nom de l'arxiu, escriure la paraula clau "all" per executar el programa per tots els arxius disponibles en el directori, o escriure múltiples noms d'arxiu (separats per comes) per executar el programa per aquells arxius en concret.

Per cada arxiu que es llegeixi, l'algorisme separa el contingut en la declaració de la gramàtica (en un text, string) i les paraules a analitzar (en un conjunt d'strings).

Seguidament, es crea la gramàtica mitjançant el text proporcionat amb el paràmetre `from_text` del mètode constructor de la classe `CFG` .

Posteriorment, es crea una instància de la classe `CKY` i s'executa el mètode `parse()` per totes les paraules a analitzar.

Finalment, s'escriu la representació de la gramàtica i els resultats de cada paraula analitzada en l'arxiu de sortida (en el directori */results*) seguint un format clar i entenedor i amb el mateix nom que l'arxiu d'entrada, afegint el prefix *results_*.

5.3 Output

En les primeres línies de l'arxiu de sortida es pot veure la representació de la gramàtica, que pot coincidir o no amb la gramàtica proporcionada depenent de si s'ha donat en Forma Normal de Chomsky (CNF), però en tot cas sempre és una gramàtica equivalent a la proporcionada (genera el mateix llenguatge).

Seguidament, es troben els resultats de cada paraula, on s'indica amb True o False si la paraula proporcionada pot ser generada per la gramàtica (a més de, en cas de tractar-se d'una gramàtica probabilística, donar la probabilitat de ser generada). A més, per les paraules que donen True com a resultat, es proporciona l'arbre de generació (amb les corresponents probabilitats, en cas de ser una PCFG), on es mostra l'arbre binari (amb possibles subarbres) amb el parsing de la paraula (símbols no terminals necessaris per genera-la ordenats correctament).

A continuació es poden veure exemples de l'output del programa tant per una gramàtica determinista com per una probabilística. Concretament, aquest són els outputs dels exemples presentats anteriorment (apartat 5.1).

Exemple output CFG (arxiu *results_racso_cfg_det10.txt*):

```
CFG(
        A -> EC | FD | ϵ
        B -> a
        C -> b
        D -> c
        E -> BK | a
        F -> BS | a
        K -> EC
        S -> EC | FD
)

* Start Symbol: A
* Terminal Symbols: a, b, c, ϵ
* Non-Terminal Symbols: A, B, C, D, E, F, K, S

##################################################

ϵ -> True
TREE -> [('A', 'ϵ')]

ac -> True
TREE -> [('A', ('F', 'a'), ('D', 'c'))]

ab -> True
TREE -> [('A', ('E', 'a'), ('C', 'b'))]

aacc -> True
TREE -> [('A', ('F', ('B', 'a'), ('S', ('F', 'a'), ('D', 'c'))), ('D', 'c'))]

aabc -> True
TREE -> [('A', ('F', ('B', 'a'), ('S', ('E', 'a'), ('C', 'b'))), ('D', 'c'))]

aaabbc -> True
TREE -> [('A', ('F', ('B', 'a'), ('S', ('E', ('B', 'a'), ('K', ('E', 'a'),
('C', 'b'))), ('C', 'b'))), ('D', 'c'))]

aaaabbcc -> True
TREE -> [('A', ('F', ('B', 'a'), ('S', ('F', ('B', 'a'), ('S', ('E', ('B', 'a'),
('K', ('E', 'a'), ('C', 'b'))), ('C', 'b'))), ('D', 'c'))), ('D', 'c'))]

aaaccc -> True
TREE -> [('A', ('F', ('B', 'a'), ('S', ('F', ('B', 'a'), ('S', ('F', 'a'),
('D', 'c'))), ('D', 'c'))), ('D', 'c'))]

aaabbc -> True
```

```
TREE -> [('A', ('F', ('B', 'a'), ('S', ('E', ('B', 'a'), ('K', ('E', 'a'),
('C', 'b'))), ('C', 'b'))), ('D', 'c'))]

aaaabbbc -> True
TREE -> [('A', ('F', ('B', 'a'), ('S', ('E', ('B', 'a'), ('K', ('E', ('B', 'a'),
('K', ('E', 'a'), ('C', 'b'))), ('C', 'b'))), ('C', 'b'))), ('D', 'c'))]

a -> False

b -> False

c -> False

abc -> False

abbc -> False

aabcc -> False

aacbb -> False

abcc -> False

abcbc -> False

aaabbbccc -> False
```

Exemple output PCFG (arxiu *results_prob1.txt*):

```
PCFG(
        S -> AB [0.9] | BC [0.1]
        A -> BA [0.5] | a [0.5]
        B -> CC [0.7] | b [0.3]
        C -> AB [0.6] | a [0.4]
)

* Start Symbol: S
* Terminal Symbols: a, b
* Non-Terminal Symbols: S, A, B, C

##################################################

ab -> True [0.135]
TREE -> [(('S', ('A', 'a', 0.5), ('B', 'b', 0.3)), 0.135)]

bab -> True [0.02295]
TREE -> [(('S', ('B', 'b', 0.3), (('C', ('A', 'a', 0.5),
```

```
('B', 'b', 0.3)), 0.09)), 0.0027), (('S', (('A', ('B', 'b', 0.3),
('A', 'a', 0.5)), 0.075), ('B', 'b', 0.3)), 0.02025)]

aabab -> True [0.014553]
TREE -> [(('S', ('A', 'a', 0.5), (('B', ('C', 'a', 0.4), (('C', (('A',
('B', 'b', 0.3), ('A', 'a', 0.5)), 0.075), ('B', 'b', 0.3)), 0.0135)), 0.00378)),
0.0042525), (('S', ('A', 'a', 0.5), (('B', (('C', ('A', 'a', 0.5),
('B', 'b', 0.3)), 0.09), (('C', ('A', 'a', 0.5), ('B', 'b', 0.3)), 0.09)),
0.00567)), 0.0042525), (('S', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112),
(('C', (('A', ('B', 'b', 0.3), ('A', 'a', 0.5)), 0.075), ('B', 'b', 0.3)),
0.0135)), 0.0001512), (('S', (('B', ('C', 'a', 0.4), (('C', ('A', 'a', 0.5),
('B', 'b', 0.3)), 0.09)), 0.0252), (('C', ('A', 'a', 0.5), ('B', 'b', 0.3)),
0.09)), 0.0002268), (('S', (('A', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)),
0.112), (('A', ('B', 'b', 0.3), ('A', 'a', 0.5)), 0.075)), 0.0042),
('B', 'b', 0.3)), 0.002835), (('S', (('A', (('B', ('C', 'a', 0.4),
(('C', ('A', 'a', 0.5), ('B', 'b', 0.3)), 0.09)), 0.0252),
('A', 'a', 0.5)), 0.0063), ('B', 'b', 0.3)), 0.002835)]

aaaaab -> True [0.013956768]
TREE -> [(('S', ('A', 'a', 0.5), (('B', ('C', 'a', 0.4), (('C', ('A', 'a', 0.5),
(('B', ('C', 'a', 0.4), (('C', ('A', 'a', 0.5), ('B', 'b', 0.3)), 0.09)),
0.0252)), 0.00756)), 0.003528)), 0.00254016), (('S', ('A', 'a', 0.5),
(('B', ('C', 'a', 0.4), (('C', (('A', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)),
0.112), ('A', 'a', 0.5)), 0.028), ('B', 'b', 0.3)), 0.00504)), 0.003528)),
0.00254016), (('S', ('A', 'a', 0.5), (('B', (('C', ('A', 'a', 0.5),
(('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112)), 0.0336), (('C',
('A', 'a', 0.5), ('B', 'b', 0.3)), 0.09)), 0.0021168)), 0.00254016),
(('S', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112), (('C', ('A', 'a', 0.5),
(('B', ('C', 'a', 0.4), (('C', ('A', 'a', 0.5), ('B', 'b', 0.3)), 0.09)),
0.0252)), 0.00756)), 0.00014112), (('S', (('B', ('C', 'a', 0.4),
('C', 'a', 0.4)), 0.112), (('C', (('A', (('B', ('C', 'a', 0.4),
('C', 'a', 0.4)), 0.112), ('A', 'a', 0.5)), 0.028), ('B', 'b', 0.3)), 0.00504)),
0.00014112), (('S', (('A', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112),
('A', 'a', 0.5)), 0.028), (('B', ('C', 'a', 0.4), (('C', ('A', 'a', 0.5),
('B', 'b', 0.3)), 0.09)), 0.0252)), 0.00063504), (('S', (('B', ('C', 'a', 0.4),
(('C', ('A', 'a', 0.5), (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112)),
0.0336)), 0.009408), (('C', ('A', 'a', 0.5), ('B', 'b', 0.3)), 0.09)),
0.000169344), (('S', (('B', (('C', ('A', 'a', 0.5), (('B', ('C', 'a', 0.4),
('C', 'a', 0.4)), 0.112)), 0.0336), ('C', 'a', 0.4)), 0.009408),
(('C', ('A', 'a', 0.5), ('B', 'b', 0.3)), 0.09)), 0.000169344),
(('S', (('A', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112),
(('A', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112), ('A', 'a', 0.5)),
0.028)), 0.001568), ('B', 'b', 0.3)), 0.00169344), (('S', (('A',
(('B', ('C', 'a', 0.4), (('C', ('A', 'a', 0.5), (('B', ('C', 'a', 0.4),
('C', 'a', 0.4)), 0.112)), 0.0336)), 0.009408), ('A', 'a', 0.5)), 0.004704),
('B', 'b', 0.3)), 0.00169344), (('S', (('A', (('B', (('C', ('A', 'a', 0.5),
(('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112)), 0.0336), ('C', 'a', 0.4)),
0.009408), ('A', 'a', 0.5)), 0.004704), ('B', 'b', 0.3)), 0.00169344)]
```

```
abbaba -> True [0.00036855]
TREE -> [(('S', ('A', 'a', 0.5), (('B', (('C', (('A', ('B', 'b', 0.3),
(('A', ('B', 'b', 0.3), ('A', 'a', 0.5)), 0.075)), 0.01125), ('B', 'b', 0.3)),
0.002025), ('C', 'a', 0.4)), 0.000567)), 0.00025515), (('S', (('B',
('C', 'a', 0.4), (('C', (('A', ('B', 'b', 0.3), (('A', ('B', 'b', 0.3),
('A', 'a', 0.5)), 0.075)), 0.01125), ('B', 'b', 0.3)), 0.002025)), 0.000567),
('C', 'a', 0.4)), 5.67e-05), (('S', (('B', (('C', ('A', 'a', 0.5),
('B', 'b', 0.3)), 0.09), (('C', (('A', ('B', 'b', 0.3), ('A', 'a', 0.5)), 0.075),
('B', 'b', 0.3)), 0.0135)), 0.0008505), ('C', 'a', 0.4)), 5.67e-05)]

bababa -> True [0.00032319]
TREE -> [(('S', ('B', 'b', 0.3), (('C', ('A', 'a', 0.5), (('B', (('C',
(('A', ('B', 'b', 0.3), ('A', 'a', 0.5)), 0.075), ('B', 'b', 0.3)), 0.0135),
('C', 'a', 0.4)), 0.00378)), 0.001134)), 3.402e-05), (('S', (('A',
('B', 'b', 0.3), ('A', 'a', 0.5)), 0.075), (('B', (('C', (('A', ('B', 'b', 0.3),
('A', 'a', 0.5)), 0.075), ('B', 'b', 0.3)), 0.0135), ('C', 'a', 0.4)), 0.00378)),
0.00025515), (('S', (('B', (('C', (('A', ('B', 'b', 0.3), ('A', 'a', 0.5)),
0.075), ('B', 'b', 0.3)), 0.0135), (('C', ('A', 'a', 0.5), ('B', 'b', 0.3)),
0.09)), 0.0008505), ('C', 'a', 0.4)), 3.402e-05)]

aaa -> True [0.05488]
TREE -> [(('S', ('A', 'a', 0.5), (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)),
0.112)), 0.0504), (('S', (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112),
('C', 'a', 0.4)), 0.00448)]

bbaaa -> True [0.0012852]
TREE -> [(('S', ('B', 'b', 0.3), (('C', (('A', ('B', 'b', 0.3),
('A', 'a', 0.5)), 0.075), (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)), 0.112)),
0.00504)), 0.0001512), (('S', (('A', ('B', 'b', 0.3), (('A', ('B', 'b', 0.3),
('A', 'a', 0.5)), 0.075)), 0.01125), (('B', ('C', 'a', 0.4), ('C', 'a', 0.4)),
0.112)), 0.001134)]

ϵ -> False

a -> False

b -> False

ac -> False

bbb -> False

abb -> False

baa -> False

abba -> False
```

6 Jocs de prova

Els jocs de prova són arxius amb exemples que permeten demostrar el funcionament del programa principal (definit en l'arxiu *main.py*). En aquesta entrega es troben tots en el directori */tests*. L'estructura del seu contingut és la que ja ha estat explicada prèviament en l'apartat 5.1.

Seguint el format especificat, les paraules a parsejar proporcionades en els jocs de prova preparats per aquesta entrega es divideixen en dos blocs (separats per comentaris amb el caràcter "#"), on primer hi ha paraules que haurien de donar com a resultat del CKY True (la paraula pot ser creada per la gramàtica) i en el segon bloc es troben paraules que haurien de donar False. Aquests resultats es poden comprovar realitzant les execucions i comprovant que, efectivament, en l'arxiu de resultats que es genera en el directori */results*, els resultats són els correctes. Per als jocs de prova predefinits en aquesta entrega, els resultats han estat renombrats per tal de poder tenir-los tots i també es troben en el directori */results*.

Les gramàtiques escollides per aquests jocs de prova s'han generat mitjançant la pròpia invenció dels autors del projecte, exemples en llibres o exemples en pàgines web, on s'hi pot destacar especialment la web `https://racso.cs.upc.edu`. Aquesta pàgina proporciona múltiples enunciats de gramàtiques lliures de context (CFGs) les solucions de les quals han estat trobades pels autors d'aquest projecte o, del contrari, trobades en els repositoris de GitHub especificats en [7] i [8].

7 Extensions

Com ja s'ha vist al llarg del document, en aquest projecte s'han realitzat les dues extensions proposades en l'enunciat de la pràctica. És a dir, la possibilitat d'aplicar l'algorisme de CKY en gramàtiques probabilístiques i la transformació automàtica de gramàtiques a Forma Normal de Chomsky (CNF).

Addicionalment, aquest projecte també ha inclòs altres extensions, com ara la obtenció dels arbres de parsing de l'algorisme CKY i la seva visualització utilitzant la llibreria `networkx`, la generació de totes les possibles paraules de la gramàtica (incloent-hi les probabilitats en cas de ser una PCFG), la lectura de gramàtiques des de diferents inputs (diccionaris, arxius o text), la creació de gramàtiques a partir d'un conjunt de paraules, l'arrodoniment dinàmic de les probabilitats, etc.

Totes aquestes extensions fan que la implementació presentada sigui molt més útil, divertida, còmode i comprensible per l'usuari.

8 Conclusions

Al llarg d'aquest projecte s'han pres moltes decisions que han acabat donant com a resultat una implementació de l'algorisme CKY i una forma de tractar i representar gramàtiques lliures de context que es considera que ha estat molt completa i satisfactòria.

Com s'ha mencionat en la prèvia secció 7, no només s'han assolit tots els requeriments i extensions de l'enunciat d'aquesta pràctica, sinó que també s'han afegit múltiples altres funcionalitats que fan que la interacció amb el codi Python d'aquest projecte sigui més còmode, fàcil, útil i divertida. Amb tot això, s'han pogut completar tots els objectius plantejats en la introducció (secció 2).

Finalment, els autors valoren molt positivament tot l'aprenentatge adquirit al llarg d'aquesta pràctica, ja que s'ha comprès molt millor el funcionament de les gramàtiques lliures de context (CFGs), la seva conversió a Forma Normal de Chomsky (CNF), el concepte de gramàtica probabilística (PCFG) i el seu funcionament, el concepte de parsing i el funcionament intern de l'algorisme de Cocke-Kasami-Younger (CKY).

9 Referències

[1] Wikipedia contributors. Context-free grammar — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Context-free_grammar&oldid=1221778491`, 2024. [Online; accessed 28-May-2024].

[2] Wikipedia contributors. Chomsky normal form — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Chomsky_normal_form&oldid=1175161174`, 2023. [Online; accessed 27-May-2024].

[3] Rafel Cases and Lluís. Màrquez. *Llenguatges, gramàtiques i autòmats : curs bàsic*. Aula politècnica ; 41. FIB. Edicions UPC, Barcelona, 2a ed. en aula politècnica edition, 2003.

[4] Wikipedia contributors. Parsing — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Parsing&oldid=1225407795`, 2024. [Online; accessed 28-May-2024].

[5] Wikipedia contributors. Cyk algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=CYK_algorithm&oldid=1213852912`, 2024. [Online; accessed 28-May-2024].

[6] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[7] Alexland. Racso-tc. `https://github.com/alexland7219/RACSO-TC`, 2024. Accessed: 2024-05-29.

[8] Artagok. Tc-racso. `https://github.com/Artagok/TC-RACSO`, 2024. Accessed: 2024-05-29.