clause\_generator.py

class ClauseGenerator:

* def generate\_term(self, force\_variable: bool = False, force\_constant: bool = False) -> Term: (also classes)

Այս կոդում սահմանված է՝ «Term» կլասը, որը ներկայացնում է առաջին կարգի լոգիկայի տերմին, որը կարող է լինել երկու տեսակի՝ փոփոխական կամ կոնստանտ:

**Ինչպես է աշխատում generate\_term ֆունկցիան՝**

1. **Փոփոխականը և կոնստանտը**
   * generate\_term ստուգում է՝ արդյոք պետք է ստիպված կերպով սահմանվի փոփոխական (force\_variable=True) կամ կոնստանտ (force\_constant=True):
     + Եթե երկու արժեք էլ պարտադիր դառնալով (հեռացված կոնֆլիկտ) ապա ձևավորվում է սխալ:
   * Եթե որևէ մեկը սահմանված չէ, ապա այն պատահականորեն որոշում է՝ 70% հավանականությամբ ընտրելով փոփոխական, իսկ մնացած 30%–ով՝ կոնստանտ:
2. **Տերմինի ստեղծում**
   * Եթե որոշվեց, որ պետք է ստեղծվի փոփոխական, ապա name-ը գրվում է պատահական կերպով ընտրելով սահմանված փոփոխականների ցուցակից (օրինակ՝ "x", "y", "z" և այլն):
   * Եթե ընտրվել է կոնստանտ, ապա name-ը գրվում է որոշված կոնստանտներից (օրինակ՝ "a", "b", "c", "d", "e"):
   * Հետագայում, նոր Term օբյեկտը ձևավորվում է name և is\_variable արժեքներով, որոնք հաստատում են տերմինի տեսակը:
3. **Ինչպես են ներկայացվում տերմինները**
   * Յուրաքանչյուր տերմին արտացոլվում է որպես իր անունը, ըստ \_\_str\_\_ մեթոդի, որը պարզապես վերադարձնում է name-ը:
   * Հիմնականում, այս տերմինները պարզ, այնուամենայնիվ տարբերակվում են միայն իրենց բնույթով՝ փոփոխական թե կոնստանտ:

Այս կոդը պարզապես ստեղծում է պարզ (атомային) տերմիններ, որոնք բնորոշվում են իրենց անվան ու "հավասարումը փոփոխական/կոնստանտ" դաշտով, սակայն դրանք չհամակցվում են մի տերմինի բարդ կառուցվածքով, ինչպես օրինակ՝ ֆունկցիայի կիրառմամբ:

* def generate\_literal(self, predicate: str = None, terms: List[Term] = None, negated: bool = None) -> Literal:

Այս ֆունկցիան պատահական literal (հայտարարություն) է ստեղծում, որը ներկայացնում է այն՝ ինչպես կանխատեսված՝ տերմինների վրա կիրառված predicate: Հիմնականում այն գործում է հետևյալ եղանակով.

* **Predicate-ի ընտրություն:**  
  Եթե տրված չէ հատուկ predicate, ապա random.choice(self.predicates)-ով ընտրում է նախապես սահմանված predicates ցուցակից (օրինակ՝ "P", "Q", "R" և այլն):
* **Տերմինների ստեղծում:**  
  Եթե տերմինները չեն տրամադրվել, ապա տերմինների քանակը (հետևաբար՝ predicate-ի arity) պատահական ընտրվում է 1-ից self.max\_term\_arity-ի միջակայքում: Հետևաբար, օգտագործվում է self.generate\_term() ֆունկցիան՝ ստեղծելու ցանկավարժ տերմիններ, որոնք կարող են լինել փոփոխականներ կամ կոնստանտներ:
* **Նեգացիայի որոշում:**  
  Եթե negated արգումենտը չի տրամադրվել, ապա պատահական ընտրություն է կատարվում՝ մոտավորապես 50% հավանականությամբ True (հայտարարությունը նեղացված է) կամ False (ներմուծված առանց նեղացման):

Հետևաբար, ֆունկցիան վերադարձնում է Literal օբյեկտ, որը կազմված է ընտրված predicate-ի, ստեղծված տերմինների և որոշված նեղացման պարամետրերի հիման վրա: Սա շատ օգտակար է լոգիկական համակարգերում, որտեղ անհրաժեշտ է ցուցադրել տարբեր նշանակության լիտերալներ՝ մոդելավորելու մոդելային լոգիկական արտահայտություններ:

* def generate\_complementary\_literal\_pair(self) -> Tuple[Literal, Literal]:

Այս ֆունկցիան պատրաստում է complementary literal-ների զույգ, որոնք նախատեսված են unification-ի միջոցով լուծելու համար, ինչը նշանակում է, որ դրանք ունեն հակասող նեղացում (մեկն ինչի նեղացումը հակասում է մյուսի նեղացման հետ) և իրենց տերմինները կարող են լինել unification-ի միջոցով համատեղելի: Ստորև մանրամասն բացատրություն կա:

1. **Նախնական predicate-ի ընտրություն և arity-ի սահմանում.**
   * Սկզբում պատահական ընտրում է predicate-ի անուն՝ վերոնշյալ ցուցակից, ինչպես նաև որոշում է տերմինների քանակը (arity), որը պատահական կերպով է ընտրվում 1-ից մինչև սահմանված առավելագույն արժեքը:
2. **Առաջին literal-ի ստեղծում.**
   * Ստեղծվում են առաջին literal-ի տերմինները՝ terms1, որտեղ յուրաքանչյուր տերմին պատահական կերպով է գրվում:
   * Մեկ-մեկ տերմինի համար 70% հավանականությամբ՝ պարտադիր ֆորսվող փոփոխական է օգտագործվում, ինչը հնարավորություն է տալիս póź дальнейшем unification-ին, քանի որ փոփոխականները ավելի εύκολα համադրվում են կոնստանտների կամ այլ փոփոխականների հետ:
   * Վերջապես, առաջին literal-ը ձևավորվում է Literal օբյեկտի միջոցով՝ պարունակում է ընտրված predicate-ը, ստացված տերմինները, և պատահական ընտրված նեղացման արժեքը (true կամ false):
3. **Երկրորդ literal-ի ստեղծում՝ terms-ի համապատասխանեցմամբ.**
   * Այս փուլում, որպես հիմք վերցվում է առաջին literal-ի տերմինների ցանկը:
   * Եթե առաջին literal-ի տերմինը փոփոխական է, ապա ընտրության տարբերակները ներառում են՝ նույն փոփոխականը, այլ փոփոխական կամ կոնստանտ:
   * Եթե առաջին literal-ի տերմինը կոնստանտ է, ապա ընտրվում է կամ նույն կոնստանտը (հակախնդիրության հնարավորության համար), կամ փոփոխական, կամ այլ կոնստանտ, որը կարող է հանգեցնել unification-ի ձախողմանը (հետաքրքիր դեպք, քանի որ դա նաև հաճախ էլ զույգերի տարբերակ է):
   * Այս ընտրությունը կատարվում է պատահական, որոշակի հավանականություններով՝ ապահովելով, որ շատ դեպքերում unification-ի համար անհրաժեշտ համապատասխանություն լինի:
4. **Լիտերալների հակասող նեղացում.**
   * Երկրորդ literal-ի negated հատկությունը սահմանվում է այդպիսով, որ հակասում է առաջին literal-ի նեղացման արժեքին: Սա կարևոր է, քանի որ կլոզերի լուծման (resolution) մեջ անհրաժեշտ է, որ հակասող լիտերալները ունենան հակասող նեղացում:
5. **Զույգի վերադարձ.**
   * Վերջում ֆունկցիան վերադարձնում է tuple, որտեղ առաջինը ներկայացնում է առաջին literal-ը և երկրորդը՝ երկրորդ literal-ը, որոնք կարող են unification-ի միջոցով լուծվել:

Այս մեթոդը օգտակար է լոգիկական համակարգերում (օրինակ՝՝ ռեզոլյուցիային հիմնված ապահովումներում), որտեղ անհրաժեշտ է ստեղծել համադրվող լիտերալների զույգեր, որոնք ավելի առաջադիմյալ unification-ի գործողություններն իրականացնելու համար:

* def can\_unify(self, term1: Term, term2: Term) -> bool:

Այս ֆունկցիան ստուգում է՝ արդյոք երկու տերմիններ (Term) կարող են համադրվել unification-ի գործընթացում։

* **Կոնստանտ-կոնստանտ դեպքում:**  
  Եթե երկուսն էլ չեն փոփոխական (is\_variable == False), ապա դրանք համադրվում են միայն եթե իրենց անունները նույնն են։ Այդուհետև «a» համադրվում է միայն «a»-ի հետ։
* **Փոփոխական-կոնստանտ կամ փոփոխական-փոփոխական դեպքում:**  
  Եթե որևէ մեկը փոփոխական է, ապա այն կարող է համադրվել ցանկացած տերմին՝ քանի որ փոփոխականը կարող է ընդունել ցանկացած արժեք։ Այսինքն՝ փոփոխականը կարող է փոխարինվել կոնստանտով կամ այլ փոփոխականով unification-ի ընթացքում։

Այսպիսով, այս կոդը հաստատում է, որ կոնստանտները չեն փոխվում (հեռորդվում) unification-ի ժամանակ, իսկ միայն փոփոխականները կարող են փոխարինվել այլ արժեքներով՝ ապահովելու համադրությունը։  
Հենց այդպիսով, կոնստանտը միշտ մնայում է իր հաստատված արժեքը, և փոփոխականը է այն էլեմենտը, որը կարող է ընդունել տարբեր արժեքներ unification-ի ընթացքում:

* def compute\_mgu(self, lit1: Literal, lit2: Literal) -> Optional[Dict[str, Term]]:

Այս ֆունկցիան հաշվարկում է երկու լիտերալի (հայտարարությունների) ամենամեծ ընդհանուր միավորիչը (MGU), եթե դրանք կարող են unification-ի միջոցով համադրվել՝ հետևյալ քայլերով:

* **Predicate-ի և նեղացման ստուգում.**  
  Նախ վերաձևվում է, թե արդյոք երկուսում նույն predicate-ը է կիրառված և արդյոք դրանց նեղացման արժեքները հակասում են (մեկը նեղացված, մյուսը ոչ): Եթե այդ պայմանները չեն բավարարվում, վերադարձվում է None:
* **Տերմինների քանակի ստուգում.**  
  Համոզվում է, որ երկուսում նույնպես նույն քանակությամբ տերմիններ կան: Եթե չհամընկնում, unification-ը չեն հնարավոր:
* **Փոխարինումների տողով անցնում.**  
  Ստեղծվում է դատարկ օբյեկտ substitution, որտեղ կստեղծվեն փոփոխականների փոխարինումներ: Հետևաբար, յուրաքանչյուր համապատասխան տերմինի զույգում (հայտարարությունների առանձին տերմինները) վերլուծվում է՝ արդյոք նրանք կարող են համադրվել:
* **Տերմինների համադրման գործընթաց.**
  + Եթե առաջին տերմինը փոփոխական է, ապա այն հնարավոր է փոխարինել երկու-ի արժեքով, եթե արդեն չի սահմանվել որևէ փոխարինում:
  + Եթե երկրորդ տերմինը փոփոխական է, ապա անդեռապես անում է նման ստուգում:
  + Եթե երկուսն էլ կոնստանտներ են, ապա պետք է ունենան նույն անունը, հակառակ դեպքում unification-ը ձախողվում է:
* **Վերջնական արդյունք.**  
  Եթե բոլոր տերմինների զույգերը համադրվում են, ապա վերադարձվում է այն substitution բառարանը, որն ընդունում է այն բոլոր փոխարինումները, որոնք թույլ են տալիս երկու լիտերալները փոխարինել այդ mapping-ի միջոցով:

Այսպիսով, ֆունկցիան ապահովում է, որ միայն անհրաժեշտ փոփոխականները փոխարինվեն, և կոնստանտները իրենց արժեքներով մնան անփոփոխ, ուստի առկա կլինի ամենաբազմակողմանի լուծում, որը կարելի է կիրառել դրանց unification-ին:

* def apply\_substitution(self, literal: Literal, substitution: Dict[str, Term]) -> Literal:

Այս ֆունկցիան նպաստում է փոփոխականների փոխարինման (substitution) կիրառմանը լիտերալի վրա՝ լիտերալում ընդգրկված յուրաքանչյուր տերմինի համար:

1. **Տերմինների անցում և փոխարինում:**  
   Ֆունկցիան անցնում է լիտերալի բոլոր տերմինների վրա։
   * Եթե տերմինը փոփոխական է և դրա անունը գտնվում է substitution բառարանի մեջ, ապա տերմինը փոխարինվում է substitution-ում նշված արժեքով (հետաքրքիր է, որ այդ արժեքը նույնպես Term է):
   * Եթե տերմինը ոչ թե փոփոխական է, կամ փոփոխական է, բայց դրա անունը բացակայում է substitution-ում, ապա այն դառնում է նույնը, ինչ սկզբնականում էր:
2. **Նոր լիտերալի ձևավորում:**  
   Ստեղծվում է նոր Literal օբյեկտ՝ նույն predicate-ով, նույն նեղացման արժեքով, բայց նոր տերմիններով, որոնք կարող են ունենալ փոխարինված արժեքներ:

Այսպես, ֆունկցիան հնարավորություն է տալիս պարզել, թե ինչպես կիրառվել են փոխարինումները լիտերալի մեջ՝ նպաստելով unification-ի և resolution-ի գործողություններին լոգիկական համակարգերում:

* def resolve\_clauses(self, clause1: Clause, clause2: Clause, lit\_idx1: int, lit\_idx2: int, mgu: Dict[str, Term]) -> Clause:

Այս ֆունկցիան իրականացնում է կլոզերի լուծման (resolution) գործընթացի մի կարևոր փուլ, որտեղ երկու կլոզերից հեռացվում են unification-ի համար ընտրված հակասող լիտերալները և դառնում է նոր կլոզ՝ մնացած լիտերալները հավաքելով՝ տեղադրված mgu փոխարինումներով։

**Գործման ընթացքում կատարված քայլերը**

1. **Հիմնական համակարգի ստեղծում.**  
   Ստեղծվում է դատարկ ցանկ resolved\_literals, որտեղ կհավաքվեն բոլոր այն լիտերալները, որոնք չհավասար են unification-ի համար ընտրված կամ լուծված լիտերալներին։
2. **Առաջին կլոզի լիտերալները.**
   * Ցիկլում անցնում ենք առաջին կլոզի բոլոր լիտերալների վրա։
   * Եթե այդ լիտերալը գտնվում է ինդեքսում, որը համապատասխանում է lit\_idx1-ին (հայտարարությունը, որը պետք է լուծվի և հեռացվի), ապա այն առանձին չի գործարկվում։
   * Մնացած բոլոր լիտերալների վրա օգտագործվում է apply\_substitution ֆունկցիան՝ mgu-ի համապատասխան փոխարինումները կիրառելու համար, ուստի ստացած լիտերալները ավելացնում են resolved\_literals–ում։
3. **Երկրորդ կլոզի լիտերալները.**
   * Նույն գործընթացն իրականացվում է երկրորդ կլոզի համար՝ սրաց կոչվող ինդեքսը lit\_idx2 է:
   * Երբ պատահում է, որ լիտերալի ինդեքսը տարբեր է lit\_idx2–ից, ապա այդ լիտերալին նույնպես կիրառվում է mgu-ի փոխարինումը, ուստի ստացված լիտերալը ավելանում է resolved\_literals–ում։
4. **Նոր կլոզի կազմումը.**
   * Վերջում, բոլոր մնացած, mgu-ի փոխարինումներ կիրառված լիտերալները կազմում են նոր Clause օբյեկտ, որը վերադարձվում է որպես այս դասի լուծված կլոզը։

Այս քայլերը ապահովում են, որ անցյալ կլոզերում առկա այն լիտերալները, որոնք unification-ի ընթացքում անմիջականորեն ընդհասարակվում են (հակասող լիտերալները), էլ հեռացվեն, իսկ մնացած լիտերալները՝ mgu-ի դիմաց, կհավաքվեն նոր կլոզում, որը օգտակար է ապագայում լուծման շարունակական գործընթացներում:

* def generate\_clause\_pair(self) -> ClausePair:

Այս ֆունկցիան, generate\_clause\_pair, նպատակ ունի ստեղծել երկու կլոզ, որոնցից կարելի է լուծել unification-ի և resolution-ի միջոցով ստանալ նոր կլոզ (resolvent): Ահա թե ինչպես է աշխատում պրոցեսը՝ քայլերով.

dataset\_generator\_saves.py

class MGUDatasetCreator:

* def format\_clause\_pair(self, pair: ClausePair) -> dict:

Այս մեթոդը նախատեսված է, որպեսզի որպես վերջնական ձևավորման քայլ, ձևավորի կլոզերի զույգը (ClausePair) տեքստային ձևաչափում, որն օգտագործվում է մոդելի ուսուցման համար։ Այն վերափոխում է գեներացված օբյեկտը լեզվական ներկայացմամբ, նախնական և դուրսած տվյալներ, ինչպես նաև ստանձնել ռաութ (հղված) տվյալներ՝ ամբողջական հետազոտության համար։

 **Մուտքային տեքստի ձևավորում:**

* Միավորված են երկու կլոզերը՝ Clause1: ... և Clause2: ... տեքստերով, որոնք ներկայացվում են նոր տողերով։

 **Հակասող լիտերալների լրացուցիչ կոնտեքստ:**

* Ընդհանրորեն, գեներացված կլոզերի զույգում առաջին դիրքում գտնվող լիտերալները, որոնք unification-ի համար ընտրվել են, ձևավորված են որպես առանձնակի տեքստ, օրինակ՝ P(x) և ¬P(a)։

 **MGU-ի (հավանական ամենամեծ ընդհանուր միավորիչի) ձևավորում:**

* Ստեղծվում է տեքստային ներկայացում, որտեղ յուրաքանչյուր փոխարինումը ցուցադրվում է key ← value ձևաչափով, օրինակ՝ {x←a, y←b}։

 **Լուծված կլոզի ձևավորում:**

* Կլոզի մնացած լիտերալները, որոնք ստացվել են unification-ի և resolution-ի արդյունքում, ներկայացվում են էլ որպես տեքստ՝, օրինակ՝ Q(a) ∨ R(a)։

 **Վերադարձվող տվյալների բազա:**

* Վերադարձվող dict-ը պարունակում է նաև օրիգինալ կլոզերի և MGU-ի «ռաութ» տեքստային ներկայացումները՝"clause1", "clause2" և "mgu\_raw", որպեսզի այլ գործընթացներում կամ հետազոտության համար էլ հասանելի լինեն տվյալները։
* def generate\_dataset(self, train\_size=8000, val\_size=1000, test\_size=1000):

Այս մեթոդը նախատեսված է ամբողջ dataset-ի (համարժեք՝ ուսուցման, վալիդացիայի և թեստավորման) գեներացիայի համար և պարունակում է հետևյալ հիմնական քայլերը՝

1. **Հավաստի և բաժանում (split) զույգերի գեներացիա**
   * Վերլուծում է ուսուցման, վալիդացիայի և թեստավորման համար համապատասխան չափերով՝
     + Ուսուցման համար՝ train\_size (օրինակ՝ 8000) կլոզերի զույգ,
     + Վալիդացիայի համար՝ val\_size (օրինակ՝ 1000) կլոզերի զույգ,
     + Թեստավորման համար՝ test\_size (օրինակ՝ 1000) կլոզերի զույգ:
   * Այս աշխատանքին համար օգտագործվում է self.generator.generate\_dataset(size) մեթոդը, որը գեներացնում է անհրաժեշտ թվով ClausePair օբյեկտներ:

* def generate\_variable\_complexity\_dataset(self,

                                            sizes=[2000, 2000, 2000, 2000],

                                            clause\_lengths=[2, 3, 4, 5],

                                            term\_arities=[2, 2, 3, 3]):

Այս մեթոդը «կլոկուլի ուսուցման» (curriculum learning) համար ձևավորում է տարբեր բարդության dataset-ներ, որտեղ յուրաքանչյուր dataset-ը ունի տարբեր սահմանափակումներ՝ կոնկրետ՝ տարբեր մակարդակներում սահմանված կլոզի երկարություն (լիտերալների քանակ) և լիտերալի տերմինների քանակ (term arity)։

**Ստեղծման գործընթացի քայլերը**

1. **Պարամետրերի ստուգում:**
   * Մեթոդը վերցնում է երեք ցուցակ՝ sizes, clause\_lengths և term\_arities։
   * Այս ցուցակների երկարությունը պետք է համընկնեն, քանի որ ամեն բարդության մակարդակի համար պետք է սահմանվի dataset-ի չափը, առավելագույն կլոզի երկարությունը և առավելագույն term arity-ն։
2. **Բարդության մակարդակների անցում:**
   * for ցիկլում enumerate-ով անցնում ենք յուրաքանչյուր բարդության մակարդակի համար, որտեղ i նշանակում է մակարդակի ինդեքսը (հիմնականում 0-ից սկսած) և յուրաքանչյուր մակարդակի համար ունեն ենք հետևյալ պարամետրերը՝
     + size: այդ մակարդակի համար ստեղծվող օրինակների քանակը (օրինակ՝ 2000)
     + max\_length: այդ մակարդակի համար սահմանված առավելագույն կլոզի երկարությունը
     + max\_arity: այդ մակարդակի համար սահմանված առավելագույն term arity-ն
3. **Գեներատորի պարամետրերի թարմացում:**
   * Միջবর্তী կարգավորման միջոցով թարմացվում են self.generator օբյեկտի պարամետրերը՝
     + max\_clause\_length փոխվում է max\_length-ի,
     + max\_term\_arity փոխվում է max\_arity-ի համար։
4. **Dataset-ի գեներացիա և ձևավորում:**
   * Հետագայում, այդ մակարդակի համար գեներացվում են size քանակի կլոզերի զույգեր self.generator.generate\_dataset(size)-ով։
   * Յուրաքանչյուր կլոզերի զույգ՝ ClausePair օբյեկտը, ձևավորվում է format\_clause\_pair մեթոդով, որը վերափոխում է տվյալները տրամաբանական արտահայտությունների, MGU-ի, տարբեր լրացուցիչ տեղեկություններով, տեքստային (string) ձևաչափի մեջ։
5. **Արտադրված տվյալների պահպանում:**
   * Ֆորմատավորված տվյալների ցուցակը (հարցավանդակ data) պահպանվում է՝ յուրաքանչյուր բարդության մակարդակի համար, առանձնացված ֆայլում՝ level\_{մակարդակի\_հերթականություն}.jsonl ֆայլային անվանումով։
6. **Metadata-ի ստեղծում:**
   * Ստեղծվում է նաև metadata.json ֆայլ, որտեղ պահվում է յուրաքանչյուր բարդության մակարդակի մասին տեղեկություն՝ ինչպես dataset-ի չափը, սահմանված առավելագույն կլոզի երկարությունը և term arity-ն, ինչպես նաև օգտագործվող predicates, փոփոխականներ և կոնստանտներ։

**Եզրակացություն**

Այս մեթոդը հնարավորություն է տալիս ուսուցիչի համար դասակարգված, փոխհարաբերական բարդության տվյալների հավաքածու կազմել։ Օրինակ, կարող եք ուսուցանել մոդելը սկզբում պարզ խնդիրներով (հնարավոր թե 2 լիտերալ, 2 term arity), իսկ հետո անցնել ավելի բարդ խնդիրների վրա (օրինակ՝ 5 լիտերալ, 3 term arity), ինչը շատ օգտակար է մոդելի تدريبي (հետևողական ուսուցման) գործընթացում: