ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра математичного моделювання соціально-економічних процесів

КУРСОВА РОБОТА

на тему:

Автоматизована перевірка доведень у численні предикатів

Студента групи Пма-32 Брецка Д.Ю.,
напряму підготовки системний аналіз

Керівники: проф. Сеньо П.С, ас. Коркуна А.М.

Національна шкала _____
Кількість балів: _____Оцінка ECTS _____

Львів – 2020

ЗМІСТ

ВСТУП

Математична логіка як самостійний розділ сучасної математики сформувався відносно нещодавно - на межі дев’ятнадцятого і двадцятого століть. Виникнення і швидкий розвиток математичної логіки були пов’язані з так званою кризою основ (засад) математики, одним з проявів якої є відомі парадокси або антиномії канторівської теорії множин. Головним предметом у дослідженнях, присвячених «ліквідуванню» кризи і «рятуванню» математики, стали принципи або правила побудови математичних тверджень і математичних теорій, зокрема, пошук відповіді на питання типу: «як повинна бути побудована теорія, щоб у ній не виникало суперечностей або антиномій?», «які властивості повинні мати методи доведення, щоб їх можна було вважати строгими?» тощо.

Для будь-кого, хто вивчає сучасну логіку, логіка першого порядку може здатися цілком природним об'єктом вивчення, а її відкриття – неминуче. Вона є семантично завершеною, адекватною для аксіоматизації всієї математики, а теорема Ліндстрома[1] показує, що це максимальна логіка, що задовольняє властивості компактності та властивості Левенгайма-Сколема[2]. Тож не дивно, що логіка першого порядку давно розглядалася як "правильна" логіка для дослідження основ математики. Вона посідає центральне місце в сучасних підручниках математичної логіки, а інші системи відведені в кулуари[3].

Останніми роками розміри доведень зростають, до прикладу доведення Великої Теореми Ферма запропоноване Ендрю Уайлзом займає 129 сторінок тексту було визнано правильним лише через 21 рік після виходу остаточної версії.

Створення систем для виведення логічних тверджень не є новою ідеєю. Про це замислювалися ще математики 17 століття, такі як Лейбніц і Хоббс, які вважали, що істинність твердження може бути встановлена за допомогою числення. Зараз відомо, що істинність теорем в логіці висловлювань може бути встановлена. Тоді як для перевірки логіки предикатів є процедури, які можуть

зайняти нескінченну кількість часу. Системи автоматизованої перевірки надають набагато швидший і надійніший спосіб перевірки доведень, оскільки майже повністю виключають людський фактор з процесу перевірки.

РОЗДІЛ 1. ЛОГІКА ТА ЇЇ ВЛАСТИВОСТІ

Логіка, як наука про міркування, визначає істинність або хибність математичного твердження, вважаючи, що певний список тверджень є істинним. Їх прийнято називати аксіомами. Існує безліч застосувань логіки: від вирішення побутових проблем до доведення математичних теорем. У цьому розділі детальніше розглянемо логіку висловлювань.

1.1 Висловлювання та дії над ними

Висловлювання – це розповідне речення, про яке можна сказати, чи істинне воно, чи хибне, але не одне й інше водночас[4]. Прикладом висловлювання може слугувати таке речення: «Мукачево – столиця України», бо можемо встановити значення істинності цього твердження. Ми позначатимемо істинність висловлювання великою латинською буквою «T», а хибність буквою «F».

Висловлювання поділяють на прості та складні. Висловлювання, які неможливо розділити на простіші, називають атомарними формулами (або атомами). Складне висловлювання – це речення, яке складається з декількох висловлювань об’єднаних в одне.

Об’єднання виконується за допомогою 5 логічних операцій:

1) Заперечення (читають „не” та позначають знаком „¬”);
2) Кон’юнкцію (читають „і (та)” й позначають знаком „∧”);
3) Диз'юнкцію (читають „або (чи)” та позначають знаком „∨”);
4) Імплікацію (читають “якщо..., то” та позначають знаком „→”);
5) Еквівалентність (читають „тоді й лише тоді” та позначають знаком „~”).

1.2 Формули алгебри висловлювань та їх еквівалентність

Будь-яке складне висловлювання є формулою. Так само будь-який атом є формулою. Розглядають два основних аспекти формул: синтаксис та семантику.

Синтаксис — це сукупність правил, які дають змогу будувати формули та розпізнавати правильні формули серед послідовностей символів. Формули в логіці висловлювань позначаються так:

- атом — це формула;
- якщо p — формула, то $(\neg p)$ — також формула;
- якщо p та q формули, то $(p \wedge q)$, $(p \vee q)$, $(p \rightarrow q)$, $(p \sim q)$ - формули;
- формули можуть бути породжені тільки скінченною кількістю застосувань указаних правил.

Формули, позначають малими латинськими літерами, у разі потреби – з індексами.

Семантика — це сукупність правил, за якими формулам присвоюють значення істинності. Нехай p та q — формули. Тоді значення істинності формул $(\neg p)$, $(p \wedge q)$, $(p \vee q)$, $(p \rightarrow q)$ та $(p \sim q)$ залежать від значень істинності формул p та q так:

1) Формула $(\neg p)$ істинна, коли p хибна, і хибна, коли формула p істинна.
2) Формула $(p \wedge q)$ істинна, якщо $(p \wedge q)$ водночас істинні. У всіх інших випадках формула $(p \wedge q)$ хибна.
3) Формула $(p \vee q)$ хибна, якщо p та д водночас хибні. У всіх інших випадках $(p \vee q)$ істинна.
4) Формула $(p \rightarrow q)$ хибна, якщо формула p істинна, а q — хибна. У всіх інших випадках вона істинна. Формулу $(p \rightarrow q)$ називають імплікацією, атом p — припущенням імплікації, а q — її висновком.
5) Формула $(p \sim q)$ істинна, якщо p та q мають однакові значення істинності. У всіх інших випадках формула $(p \sim q)$ хибна.

Щоб присвоїти значення істинності для складного висловлювання потрібно надати значення істинності всім атомам, які містить відповідна формула. Набір значень істинності всіх атомів формули називають її інтерпретацією. Формулу f логіки висловлювань називають *загальнозначущою* чи *тавтологією*, якщо вона виконується в усіх інтерпретаціях. Якщо формула f — тавтологія, то використовують позначення $\vDash f$.

Формули f і g називають *еквівалентними*, *рівносильними* чи *тотожними* (позначають $f = g$), якщо значення їх істинності є одними і тими самим для усіх інтерпретацій цих формул. Формули f і g еквівалентні тоді й лише тоді, коли формула $(f \sim g)$ загальнозначуща, тобто $f \vDash g$. Така властивість називається властивістю еквівалентності.

Розглянемо список формул (еквівалентних), які задають правила для перетворень висловлювань, їх називають *законами логіки висловлювань*.

Закони комутативності	$p \vee q = q \vee p;$ $p \wedge q = q \wedge p$
Закони асоціативності	$(p \vee q) \vee r = p \vee (q \vee r)$ $(p \wedge q) \wedge r = p \wedge (q \wedge r)$
Закони дистрибутивності	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$
Закон суперечності	$p \wedge \neg p = F$
Закон виключеного третього	$p \vee \neg p = T$
Закон подвійного заперечення	$\neg\neg p = p$
Закони ідемпотентності	$p \vee p = p$ $p \wedge p = p$
Закони де Моргана	$\neg(p \vee q) = \neg p \wedge \neg q$ $\neg(p \wedge q) = \neg p \vee \neg q$
Закони поглинання	$(p \vee q) \wedge p = p$

	$(p \wedge q) \vee p$ =p
Співвідношення для сталих	$p \vee T = T;$ $p \wedge T = p$ $p \vee p = p;$ $p \wedge$ F = F

1.3 Булеві алгебри

Теорія булевих алгебр – це розділ алгебри, який дуже тісно пов'язаний з логікою. Важливим структурним елементом будь-якої булевої алгебри є визначені в ній алгебраїчні операції. Нехай A - довільна множина. Для того, щоб задати на множині A алгебраїчну операцію * необхідно виконати дві умови:

1) Потрібно визначити правило, за яким будь-яким двом елементам x і y з множини A ставився б у відповідність єдиний для цієї пари елемент, при цьому ми вважаємо, що x, y – кортеж;
2) Цей елемент повинен належати множині A. У цьому випадку говорять, що множина A замкнута щодо даної операції *.

Булева алгебра – це система, яка складається з таких елементів: множини A та двома бінарними алгебраїчними операціями, які визначені на ній, а саме $\vee$ та $\wedge$. Ці операції повинні задовольняти наступні аксіоми:

- $x \vee x = x;$
- $x \wedge x = x;$
- $x \vee y = x \vee y;$
- $x \wedge y = x \wedge y;$
- $(x \vee y) \vee z = x \vee (y \vee z);$
- $(x \wedge y) \wedge z = x \wedge (y \wedge z);$
- Існує елемент $\mathbf{0}$ такий, що $x \vee \mathbf{0} = x$ для всіх $x;$
- Існує елемент $\mathbf{1}$ такий, що $x \wedge \mathbf{1} = x$ для всіх $x;$
- $x \vee (x \wedge y) = x;$

- $x \wedge (x \vee y) = x$;
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$;
- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$;
- Для будь-якого x існує елемент $\neg x$ такий, що $x \vee \neg x = 1$ і $x \wedge \neg x = 0$.

У кожній булевій алгебрі істинними є такі твердження:

- Елементи $\mathbf{0}$ та $\mathbf{1}$ – є єдиними;
- Кожен елемент має єдине заперечення (доповнення);
- Для кожного елемента x, $\neg\neg x = x$;
- $\mathbf{\neg 0 = 1},\ \neg \mathbf{1 = 0.}$

РОЗДІЛ 2. ЧИСЛЕННЯ ПРЕДИКАТІВ

2.1 Логіка предикатів

В основі числення предикатів лежить логіка предикатів. Предикат – це твердження про те, чи має об’єкт з предметної області певну властивість. Так само як і логіка висловлювань, логіка предикатів поділяється на дві частини: синтаксис і семантику. Синтаксис мови логіки предикатів включає в себе:

- логічні символи;
- великі та маленькі латинські літери;
- допоміжні символи;

У свою чергу логічні символи складаються з:

- власне логічних символів, які використовувалися в логіці висловлювань ($\vee$,$\wedge$, $\rightarrow$, $\neg$, $\sim$);
- кванторів.

Існує два квантори:

- існування, який позначається символом "$\exists$";
- загальності, який позначається символом "$\forall$".

Де квантор загальності визначає поняття «для всіх, кожен», а існування – «існує, хоча б для одного». Маленькими латинськими літерами позначають змінні, константи та функції. Великими латинськими літерами позначаємо предикати.

Структура логіки предикатів складається з: змінних, констант, функцій, термів, атомарних формул і формул. *Змінна* позначає якийсь об'єкт з предметної області, *константа* позначає конкретний об'єкт з предметної області, а *функція* – це відображення з множини параметрів у предметну область. Визначення *терму* задається індуктивно:

- Будь-яка змінна – це *терм*;
- Будь-яка константа – це *терм*;
- Якщо $t_1, t_2, \dots t_n$ – терми, тоді n-арна функція $f(t_1, t_2, \dots t_n)$ – є *термом*.

Нехай $t_1, t_2, \dots t_n$ – терми, тоді n-арний предикат $P(t_1, t_2, \dots t_n)$ – є *атомарною формулою*. Тепер можемо надати визначення *формули* логіки предикатів.

- Будь-яка атомарна формула є *формулою*;
- Якщо A – це формула, тоді $\neg A$ також є формулою;
- Якщо A і B – це формули, тоді $A \vee B, A \wedge B, A \to B, A \sim B$ також є формулами;
- Якщо A – це формула, а x – змінна, то $(\forall x)A(x)$ і $(\exists x)A(x)$ також є формулами.

2.2 Числення секвенцій

Також в основі числення предикатів лежить поняття секвенцій. Секвенція – це рядок вигляду:

$A_1, A_2 \dots A_k \vdash B_1, B_2 \dots B_m$,

де $\forall\, i = 1{:}\,k, \forall\, j = 1{:}\,m$:

$A_i, B_j -$ формули, і має таку інтерпретацію:

$$\bigwedge_{i=1}^{k} A_i \to \bigvee_{j=1}^{m} B_j$$

Ліва частина секвенції називається *антецедент.* Права частина секвенції називається *саксцедент.* Тоді як ліва і права частини секвенцій називаються *цедентами.* Цеденти будемо позначати великими грецькими літерами Γ, Π, Δ. Доведення в численні секвенцій складається з дерева, вузлами якого є секвенції. Коренем дерева є секвенція, яку ми хочемо довести, а листками є початкові секвенції або аксіоми (зазвичай єдиною аксіомою є $A \vdash A$), кожна секвенція в доведенні, крім початкових, повинна бути виведена з попередніх за допомогою правил виводу. Правила виводу позначаються у вигляді фігури: $\frac{S_1}{S}$ АБО $\frac{S_1 \ S_2}{S}$, і означають, що з S_1 можна вивести S , АБО з S_1 і S_2 можна вивести S. У правилах виводу $\frac{S_1 \ S_2}{S}$, S_1 та S_2 – верхні секвенції, а S – нижня секвенція.

Правила виводу поділяються на *слабкі* та *сильні*:

Слабкі	
Обміну: ліве $\frac{\Gamma,A,B,\Pi\vdash\Delta}{\Gamma,B,A,\Pi\vdash\Delta}$	Обміну: праве $\frac{\Gamma\vdash A,B,\Pi,\Delta}{\Gamma\vdash B,A,\Pi,\Delta}$
Спрощення: ліве $\frac{A,A,\Gamma\vdash\Delta}{A,\Gamma\vdash\Delta}$	Спрощення: праве $\frac{\Gamma\vdash\Delta,A,A}{\Gamma\vdash\Delta,A}$
Послаблення: ліве $\frac{\Gamma\vdash\Delta}{A,\Gamma\vdash\Delta}$	Послаблення: праве $\frac{\Gamma\vdash\Delta}{\Gamma\vdash\Delta,A}$
Обтинання: $\frac{\Gamma\vdash\Delta,A \quad \Gamma,A\vdash\Delta}{\Gamma\vdash\Delta}$	
Сильні	
Заперечення: ліве $\frac{\Gamma\vdash\Delta,A}{\neg A,\Gamma\vdash\Delta}$	Заперечення: праве $\frac{A,\Gamma\vdash\Delta}{\Gamma\vdash\Delta,\neg A}$
Кон’юнкції: ліве $\frac{A,B,\Gamma\vdash\Delta}{A\wedge B,\Gamma\vdash\Delta}$	Кон’юнкції: праве $\frac{\Gamma\vdash\Delta,A \quad \Gamma\vdash\Delta,B}{\Gamma\vdash\Delta,A\wedge B}$
Диз’юнкції: ліве $\frac{A,\Gamma\vdash\Delta \quad B,\Gamma\vdash\Delta}{A\vee B,\Gamma\vdash\Delta}$	Диз’юнкції: праве $\frac{\Gamma\vdash\Delta,A,B}{\Gamma\vdash\Delta,A\vee B}$
Імплікації: ліве $\frac{\Gamma\vdash\Delta,A \quad B,\Gamma\vdash\Delta}{A\to B,\Gamma\vdash\Delta}$	Імплікації: праве $\frac{A,\Gamma\vdash\Delta,B}{\Gamma\vdash\Delta,A\to B}$

Загальності: ліве $\frac{A(t),\Gamma \vdash \Delta}{(\forall x)A(x),\Gamma \vdash \Delta}$	Загальності: праве $\frac{\Gamma \vdash \Delta,A(b)}{\Gamma \vdash \Delta,(\forall x)A(x)}$
Існування: ліве $\frac{A(b),\Gamma \vdash \Delta}{(\exists x)A(x),\Gamma \vdash \Delta}$	Існування: праве $\frac{\Gamma \vdash \Delta,A(t)}{\Gamma \vdash \Delta,(\exists x)A(x)}$

Формула C є предком формули D тоді і тільки тоді, коли існує ланцюг з ≥ 0 предків від D до C.

Аналогічно задається означення нащадка. Називатимемо формулу C прямим предком формули D, якщо C предок D і якщо C є тою ж самою формулою що і D. Аналогічно задається означення прямого нащадка.

РОЗДІЛ 3. ОПИС ПРОГРАМИ

Програма написана на мові програмування Python. Жодних сторонніх бібліотек при створенні програми використано не було.

Для реалізації перевірки доведень в численні логіки висловлювань, було створено систему класів, які програмно реалізують об’єктну структуру логіки висловлювань. До системи входять такі класи: Constant, Variable, Function, Term, AtomicFormulae, Formula, Sequence, InferenceCheck, Node, Proof. Кожен з цих класів відповідає об’єктам: константа, змінна, функція, терм, атомарна формула, формула, секвенція, перевірка формул доведень, вузол дерева доведення, та доведення відповідно. Програма, в основному, має ієрархічну структуру, завдяки чому зручно використовувати побудову класів рекурсивними методами. Велика кількість методів можуть повторювати в різних класах, така структура надає змогу використовувати властивості поліморфізму і полегшувати роботу програми для користувача.

Клас Constant: на вхід приймає параметр symbol. Параметр набуває значення тільки з певної множини символів: малих латинських літер та

довільної кількості натуральних чисел, які можуть виступати в ролі індексу. Всі обмеження встановлені згідно означення синтаксису логіки предикатів.

Методи класу: text та print. Вони виконують функції виведення текстової форми формули на екран та, власне, перетворення вхідних даних у текст.

Програмна реалізація класу Constant:

```
class Constant:
    def __init__(self, symbol):
        # Checking that our symbols type match
        assert len(symbol) > 0
        if len(symbol) > 1:
            assert symbol[0] in constant_symbols
            for i in range(1, len(symbol)):
                assert symbol[i] in index_symbols
        elif len(symbol) == 1:
            assert symbol in constant_symbols

        self.symbol = symbol

    def print(self):
        print(self.text())

    def text(self):
        return self.symbol
```

Клас Variable: на вхід приймає параметр symbol. Параметр набуває значення тільки з певної множини символів: малих грецьких літер та довільної кількості натуральних чисел, які можуть виступати в ролі індексу. Всі обмеження встановлені згідно означення синтаксису логіки предикатів.

Методи класу: text та print. Вони виконують функції виведення текстової форми формули на екран та, власне, перетворення вхідних даних у текст.

Програмна реалізація класу Variable:

```
class Variable:
    def __init__(self, symbol):
        # Checking that our symbols type match
        assert len(symbol) > 0
        if len(symbol) > 1:
            assert symbol[0] in variable_symbols
            for i in range(1, len(symbol)):
                assert symbol[i] in index_symbols
        elif len(symbol) == 1:
            assert symbol in variable_symbols

        self.symbol = symbol

    def print(self):
        print(self.text())
```

```
    def text(self):
        return self.symbol
```

Клас Function: на вхід приймає параметри symbol та масив даних param_arr. Symbol набуває значення тільки з певної множини символів: малих латинських літер та довільної кількості натуральних чисел, які можуть виступати в ролі індексу. Всі обмеження встановлені згідно означення синтаксису логіки предикатів. Масив param_arr може приймати значення лише типу Constant або Variable.

Методи класу: text, print та get_P_t_f_var. Перші два методи виконують функції виведення текстової форми формули на екран та, власне, перетворення вхідних даних у текст. Метод get_P_t_f_var – це наслідок поліморфізму великої кількості класів, на вхід приймає значення масиву, і перевіряє значення вхідних даних у param_arr та записує їх у вхідний масив.

Програмна реалізація класу Function:

```
class Function:
    def __init__(self, symbol, param_arr):
        # Checking that our symbols type match
        assert len(symbol) > 0
        if len(symbol) > 1:
            assert symbol[0] in function_symbols
            for i in range(1, len(symbol)):
                assert symbol[i] in index_symbols
        elif len(symbol) == 1:
            assert symbol in function_symbols

        # Checking that our parameters type match
        for param in param_arr:
            assert isinstance(param, Constant) or isinstance(param, Variable)

        self.symbol = symbol
        self.param_arr = param_arr  # variables or constants

    def print(self):
        print(self.text())

    def text(self):
        return self.symbol + '(' + ','.join([param.text() for param in self.param_arr]) + ')'

    def get_P_t_f_var(self, arr):
        for i in range(len(self.param_arr)):
            if isinstance(self.param_arr[i], Variable):
                arr.append(self.param_arr[i])
```

Клас Term: на вхід приймає параметри symbol та масив даних param_arr. Symbol набуває значення тільки з певної множини символів: малих латинських літер та довільної кількості натуральних чисел, які можуть виступати в ролі індексу. Всі обмеження встановлені згідно означення синтаксису логіки предикатів. Масив param_arr може приймати значення лише типу Variable, Function або Term.

Методи класу: text, print та get_P_t_f_var. Перші два методи виконують функції виведення текстової форми формули на екран та, власне, перетворення вхідних даних у текст. Метод get_P_t_f_var – це наслідок поліморфізму великої кількості класів, на вхід приймає значення масиву, перевіряє значення вхідних даних у param_arr, записує їх у вхідний масив, та викликає себе ж у елементах масиву param_arr.

Програмна реалізація класу Term:

```
class Term:
    def __init__(self, symbol, param_arr):
        # Checking that our symbols type match
        assert len(symbol) > 0
        if len(symbol) > 1:
            assert symbol[0] in function_symbols
            for i in range(1, len(symbol)):
                assert symbol[i] in index_symbols
        elif len(symbol) == 1:
            assert symbol in function_symbols

        # Checking that our parameters type match
        for param in param_arr:
            assert isinstance(param, Function) or isinstance(param, Variable) or isinstance(param, Term)

        self.symbol = symbol
        self.param_arr = param_arr  # functions with variables or constants

    def print(self):
        print(self.text())

    def text(self):
        return self.symbol + '(' + ','.join([param.text() for param in self.param_arr]) + ')'

    def get_P_t_f_var(self, arr):
        for i in range(len(self.param_arr)):
            if isinstance(self.param_arr[i], Term):
                arr.append(self.param_arr[i])
                self.param_arr[i].get_P_t_f_var(arr)
            if isinstance(self.param_arr[i], Function):
                arr.append(self.param_arr[i])
                self.param_arr[i].get_P_t_f_var(arr)
```

```
            if isinstance(self.param_arr[i], Variable):
                arr.append(self.param_arr[i])
```

Клас AtomicFormulae: на вхід приймає параметри symbol та масив даних term_arr. Symbol набуває значення тільки з певної множини символів: малих латинських літер та довільної кількості натуральних чисел, які можуть виступати в ролі індексу. Всі обмеження встановлені згідно означення синтаксису логіки предикатів. Масив term_arr може приймати значення лише типу Term або Variable.

Методи класу: text, print та get_P_t_f_var. Перші два методи виконують функції виведення текстової форми формули на екран та, власне, перетворення вхідних даних у текст. Метод get_P_t_f_var – це наслідок поліморфізму великої кількості класів, на вхід приймає значення масиву, перевіряє значення вхідних даних у term_arr, записує їх у вхідний масив, та викликає себе ж у елементах масиву term_arr.

Програмна реалізація класу AtomicFormulae:

```
class AtomicFormulae:
    def __init__(self, symbol, term_arr):
        # Checking that our symbols type match
        assert len(symbol) > 0
        if len(symbol) > 1:
            assert symbol[0] in predicate_symbols
            for i in range(1, len(symbol)):
                assert symbol[i] in index_symbols
        elif len(symbol) == 1:
            assert symbol in predicate_symbols

        # Checking that our parameters type match
        for param in term_arr:
            assert isinstance(param, Term) or isinstance(param, Variable)

        self.symbol = symbol
        self.term_arr = term_arr  # terms

    def print(self):
        print(self.text())

    def text(self):
        return self.symbol + '(' + ','.join([term.text() for term in self.term_arr]) + ')'

    def get_P_t_f_var(self, arr):
        for i in range(len(self.term_arr)):
            if isinstance(self.term_arr[i], Term):
                arr.append(self.term_arr[i])
                self.term_arr[i].get_P_t_f_var(arr)
```

```
            if isinstance(self.term_arr[i], Function):
                arr.append(self.term_arr[i])
                self.term_arr[i].get_P_t_f_var(arr)
            if isinstance(self.term_arr[i], Variable):
                arr.append(self.term_arr[i])
```

Клас Formulae: на вхід приймає масиви даних formulae_arr і quantifier. У конструкторі клас визначає, чи містить формула в собі предикати (у масиві quantifier) та яка довжина формули, і, в залежності від цього, продовжує свою роботу. Масив formulae_arr може приймати значення лише типу Formulae, AtomicFormulae або логічні символи ‘¬’ , ‘∨’, ‘∧’, ‘→’. Конструктор надає 10 способів створення формули з уже наявної:

- У formulae_arr входить лише одна формула або висловлення.
- formulae_arr має вигляд [‘¬’, F], де F формула. Таким чином можна реалізувати заперечення формули F
- formulae_arr має вигляд [F_1, ‘∨’, F_2], де F_1 та F_2 формули. Таким чином можна реалізувати диз’юнкцію формул F_1 та F_2
- formulae_arr має вигляд [F_1, ‘∧’, F_2], де F_1 та F_2 формули. Таким чином можна реалізувати кон’юнкцію формул F_1 та F_2
- Коли formulae_arr має вигляд [F_1, ‘→’, F_2], де F_1 та F_2 формули. Таким чином можна реалізувати імплікацію формул F_1 та F_2

І ще 5 таких самих, але з використанням предикатів та предикатних змінних.

Методи класу: text, print та get_P_t_f_var. Перші два методи виконують функції виведення текстової форми формули на екран та, власне, перетворення вхідних даних у текст. Метод get_P_t_f_var – це метод, з якого починаються аналогічні методи в інших класах. На вхід він приймає значення масиву, перевіряє значення вхідних даних у formulae_arr, записує їх у вхідний масив, викликає себе ж у елементах масиву formulae_arr, та повертає готовий масив.

Програмна реалізація класу Formulae:

```
class Formulae:
    def __init__(self, quantifier, formulae_arr):
```

```
        # Checking that quantifier has appropriate length (2 or 0)
        assert len(quantifier) == 2 or len(quantifier) == 0

        # Checking that our logical part is appropriate without quantifier
        if len(quantifier) == 0:
            assert len(formulae_arr) == 2 or len(formulae_arr) == 3 or len(formulae_arr) == 1
            if len(formulae_arr) == 2:
                assert formulae_arr[0] == l_not
                assert isinstance(formulae_arr[1], AtomicFormulae) or isinstance(formulae_arr[1], Formulae)
            if len(formulae_arr) == 3:
                assert isinstance(formulae_arr[0], AtomicFormulae) or isinstance(formulae_arr[0], Formulae)
                assert formulae_arr[1] in logical_symbols_binary
                assert isinstance(formulae_arr[2], AtomicFormulae) or isinstance(formulae_arr[2], Formulae)
            if len(formulae_arr) == 1:
                assert isinstance(formulae_arr[0], AtomicFormulae)
        # Checking that our logical part is appropriate with quantifier
        else:
            assert len(formulae_arr) == 1
            assert isinstance(formulae_arr[0], Formulae) or isinstance(formulae_arr[0], AtomicFormulae)
            assert quantifier[0] in quantifier_symbols
            assert isinstance(quantifier[1], Variable)

        self.formulae_arr = formulae_arr  # formulas and connectives (a and b)
        self.quantifier = quantifier  # first element = quantifier, second = variable

    def text(self):
        txt = ''
        if len(self.quantifier) == 0:
            for i in range(len(self.formulae_arr)):
                if self.formulae_arr[i] in logical_symbols:
                    txt += ' ' + self.formulae_arr[i] + ' '
                else:
                    txt += self.formulae_arr[i].text()
        else:
            txt = '(' + self.quantifier[0] + self.quantifier[1].text() + ')('
            txt += self.formulae_arr[0].text() + ')'
        return txt

    def print(self):
        print(self.text())

    def get_P_t_f_var(self, arr):
        for i in range(len(self.formulae_arr)):
            if isinstance(self.formulae_arr[i], Formulae):
                self.formulae_arr[i].get_P_t_f_var(arr)
            if isinstance(self.formulae_arr[i], AtomicFormulae):
                arr.append(self.formulae_arr[i])
                self.formulae_arr[i].get_P_t_f_var(arr)
        return arr
```

Клас Sequence: на вхід приймає параметри antecedent і succedent. Параметри відіграють роль *саксцедент* та *антецедента* (правої та лівої частини секвенції) відповідно. Можуть приймати тільки єдине значення – екземпляр клас Formulae.

Методи класу: text та print. Вони виконують функції виведення текстової форми формули на екран та, власне, перетворення вхідних даних за синтаксисом секвенцій у текст.

Програмна реалізація класу Sequence:

```
class Sequence:
    def __init__(self, antecedent, succedent):
        for formula in antecedent:
            assert isinstance(formula, Formulae)
        for formula in succedent:
            assert isinstance(formula, Formulae)
        self.antecedent = antecedent  # left part
        self.succedent = succedent  # right part

    def text(self):
        return ', '.join([formula.text() for formula in self.antecedent]) + '  ' + l_inference + '  ' +\
               ', '.join([formula.text() for formula in self.succedent])

    def print(self):
        print(self.text())
```

Набір функцій з файлу InferenceCheck: це файл, у якому зберігаються всі формули, які потрібні для доведення теорем у системах числення предикатів.

Для зручності користування програмою, функції, які відповідають слабким структурним правилам виведення, можуть перевірити багаторазове застосування правила. Таким чином зменшується кількість монотонних дій під час процесу конструювання доведення. Функції повертають значення True, якщо при виконанні перетворень верхньої секвенції отримується нижня.

Функція	**Правило**
exchange_left	Обміну: ліве
exchange_right	Обміну: праве
contraction_left	Скорочення: ліве
contraction_right	Скорочення: праве
weakening_left	Послаблення: ліве
weakening_right	Послаблення: праве
cut	Відтинання

not_left	Заперечення: ліве
not_right	Заперечення: праве
and_left	Кон’юнкції: ліве
and_right	Кон’юнкції: праве
or_left	Диз’юнкції: ліве
or_right	Диз’юнкції: праве
imp_left	Імплікації: ліве
imp_right	Імплікації: праве
univ_left	Загальності: ліве
univ_right	Загальності: праве
exist_left	Існування: ліве
exist_right	Існування: праве

Функція exchange_left: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції exchange_left:

```
def exchange_left(s_upper, s_lower):
    if s_upper.succedent != s_lower.succedent:
        return False
    if len(s_upper.antecedent) != len(s_lower.antecedent):
        return False
    antecedent_uniq = []
    for i in range(len(s_upper.antecedent)):
        if not s_upper.antecedent[i] in antecedent_uniq:
            antecedent_uniq.append(s_upper.antecedent[i])
        if not s_lower.antecedent[i] in antecedent_uniq:
            antecedent_uniq.append(s_lower.antecedent[i])
    for uniq in antecedent_uniq:
        counter_up = 0
        counter_low = 0
        for i in range(len(s_upper.antecedent)):
            if uniq == s_upper.antecedent[i]:
                counter_up += 1
            if uniq == s_lower.antecedent[i]:
                counter_low += 1
        if not counter_up == counter_low:
            return False
    return True
```

Функція exchange_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції exchange_right:

```
def exchange_right(s_upper, s_lower):
    if s_upper.antecedent != s_lower.antecedent:
        return False
    if len(s_upper.succedent) != len(s_lower.succedent):
        return False
    succedent_uniq = []
    for i in range(len(s_upper.succedent)):
        if not s_upper.succedent[i] in succedent_uniq:
            succedent_uniq.append(s_upper.succedent[i])
        if not s_lower.succedent[i] in succedent_uniq:
            succedent_uniq.append(s_lower.succedent[i])
    for uniq in succedent_uniq:
        counter_up = 0
        counter_low = 0
        for i in range(len(s_upper.succedent)):
            if uniq == s_upper.succedent[i]:
                counter_up += 1
            if uniq == s_lower.succedent[i]:
                counter_low += 1
        if not counter_up == counter_low:
            return False
    return True
```

Функція contraction_left: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції contraction_left:

```
def contraction_left(s_upper, s_lower):
    if s_upper.succedent != s_lower.succedent:
        return False
    if len(s_upper.antecedent) <= len(s_lower.antecedent):
        return False
    set_upper = set(s_upper.antecedent)
    set_lower = set(s_lower.antecedent)
    if not set_lower == set_upper:
        return False
    for uniq in set_upper:
        counter_up = 0
        counter_low = 0
        for i in range(len(s_upper.antecedent)):
            if uniq == s_upper.antecedent[i]:
                counter_up += 1
            if i < len(s_lower.antecedent):
                if uniq == s_lower.antecedent[i]:
                    counter_low += 1
        if not counter_up >= counter_low:
            return False
    return True
```

Функція contraction_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції contraction_right:

```
def contraction_right(s_upper, s_lower):
    if s_upper.antecedent != s_lower.antecedent:
        return False
    if len(s_upper.succedent) <= len(s_lower.succedent):
        return False
    set_upper = set(s_upper.succedent)
    set_lower = set(s_lower.succedent)
    if not set_lower == set_upper:
        return False
    for uniq in set_upper:
        counter_up = 0
        counter_low = 0
        for i in range(len(s_upper.succedent)):
            if uniq == s_upper.succedent[i]:
                counter_up += 1
            if i < len(s_lower.succedent):
                if uniq == s_lower.succedent[i]:
                    counter_low += 1
        if not counter_up >= counter_low:
            return False
    return True
```

Функція weakening_left: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції weakening_left:

```
def weakening_left(s_upper, s_lower):
    if s_upper.succedent != s_lower.succedent:
        return False
    if len(s_upper.antecedent) >= len(s_lower.antecedent):
        return False
    antecedent_uniq = []
    for i in range(len(s_lower.antecedent)):
        if i < len(s_upper.antecedent):
            if not s_upper.antecedent[i] in antecedent_uniq:
                antecedent_uniq.append(s_upper.antecedent[i])
        if not s_lower.antecedent[i] in antecedent_uniq:
            antecedent_uniq.append(s_lower.antecedent[i])
    for uniq in antecedent_uniq:
        counter_up = 0
        counter_low = 0
        for i in range(len(s_lower.antecedent)):
            if i < len(s_upper.antecedent):
                if uniq == s_upper.antecedent[i]:
                    counter_up += 1
            if uniq == s_lower.antecedent[i]:
                counter_low += 1
        if counter_up > counter_low:
```

```
            return False
    return True
```

Функція weakening_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції weakening_right:

```
def weakening_right(s_upper, s_lower):
    if s_upper.antecedent != s_lower.antecedent:
        return False
    if len(s_upper.succedent) >= len(s_lower.succedent):
        return False
    succedent_uniq = []
    for i in range(len(s_lower.succedent)):
        if i < len(s_upper.succedent):
            if not s_upper.succedent[i] in succedent_uniq:
                succedent_uniq.append(s_upper.succedent[i])
        if not s_lower.succedent[i] in succedent_uniq:
            succedent_uniq.append(s_lower.succedent[i])
    for uniq in succedent_uniq:
        counter_up = 0
        counter_low = 0
        for i in range(len(s_lower.succedent)):
            if i < len(s_upper.succedent):
                if uniq == s_upper.succedent[i]:
                    counter_up += 1
            if uniq == s_lower.succedent[i]:
                counter_low += 1
        if counter_up > counter_low:
            return False
    return True
```

Функція cut: на вхід приймає параметри s_upper1, s_upper2 і s_lower, які приймають значення екземпляру класу Sequence і відповідають за дві верхні та нижню секвенції відповідно.

Програмна реалізація функції cut:

```
def cut(s_upper1, s_upper2, s_lower):
    if s_upper1.antecedent != s_upper2.antecedent[:-1]:
        return False
    if s_upper1.succedent[:-1] != s_upper2.succedent:
        return False
    if s_lower.succedent != s_upper2.succedent:
        return False
    if s_lower.antecedent != s_upper1.antecedent:
        return False
    if s_upper1.succedent[-1] != s_upper2.antecedent[-1]:
        return False
    return True
```

Функція not_left: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції not_left:

```
def not_left(s_upper, s_lower):
    if s_upper.antecedent != s_lower.antecedent[1:]:
        return False
    if s_upper.succedent[:-1] != s_lower.succedent:
        return False
    A = s_upper.succedent[-1]
    not_A = s_lower.antecedent[0]
    if not_A.quantifier:
        return False
    if not_A.formulae_arr[0] != symb.l_not:
        return False
    if not_A.formulae_arr[1] != A:
        return False
    return True
```

Функція not_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції not_right:

```
def not_right(s_lower, s_upper):
    if s_upper.antecedent != s_lower.antecedent[1:]:
        return False
    if s_upper.succedent[:-1] != s_lower.succedent:
        return False
    not_A = s_upper.succedent[-1]
    A = s_lower.antecedent[0]
    if not_A.quantifier:
        return False
    if not_A.formulae_arr[0] != symb.l_not:
        return False
    if not_A.formulae_arr[1] != A:
        return False
    return True
```

Функція and_left: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції and_left:

```
def and_left(s_upper, s_lower):
    if s_upper.antecedent[2:] != s_lower.antecedent[1:]:
        return False
    if s_upper.succedent != s_lower.succedent:
        return False
```

```
    A_and_B = s_lower.antecedent[0]
    if not A_and_B.formulae_arr[1] == symb.l_and:
        return False
    if A_and_B.quantifier:
        return False
    A = s_upper.antecedent[0]
    B = s_upper.antecedent[1]
    if A != A_and_B.formulae_arr[0]:
        return False
    if B != A_and_B.formulae_arr[2]:
        return False
    return True
```

Функція and_right: на вхід приймає параметри s_upper1, s_upper2 і s_lower, які приймають значення екземпляру класу Sequence і відповідають за дві верхні та нижню секвенції відповідно.

Програмна реалізація функції and_right:

```
def and_right(s_upper1, s_upper2, s_lower):
    if s_upper1.succedent[:-1] != s_upper2.succedent[:-1]:
        return False
    if s_upper1.succedent[:-1] != s_lower.succedent[:-1]:
        return False
    if s_upper1.antecedent != s_upper2.antecedent:
        return False
    if s_upper1.antecedent != s_lower.antecedent:
        return False
    A = s_upper1.succedent[-1]
    B = s_upper2.succedent[-1]
    A_and_B = s_lower.succedent[-1]
    if A != A_and_B.formulae_arr[0]:
        return False
    if A_and_B.formulae_arr[1] != symb.l_and:
        return False
    if A_and_B.quantifier:
        return False
    if B != A_and_B.formulae_arr[2]:
        return False
    return True
```

Функція or_left: на вхід приймає параметри s_upper1, s_upper2 і s_lower, які приймають значення екземпляру класу Sequence і відповідають за дві верхні та нижню секвенції відповідно.

Програмна реалізація функції or_left:

```
def or_left(s_upper1, s_upper2, s_lower):
    if not s_upper1.succedent == s_upper2.succedent and s_upper2.succedent == s_lower.succedent:
        return False
    if not s_upper1.antecedent[1:] == s_upper2.antecedent[1:]:
        return False
    A_or_B = s_lower.antecedent[0]
    if not A_or_B.formulae_arr[1] == symb.l_or:
        return False
```

```
    if A_or_B.quantifier:
        return False
    A = s_upper1.antecedent[0]
    B = s_upper2.antecedent[0]
    if A != A_or_B.formulae_arr[0]:
        return False
    if B != A_or_B.formulae_arr[2]:
        return False
    return True
```

Функція or_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції or_right:

```
def or_right(s_upper, s_lower):
    if s_upper.succedent[:-2] != s_lower.succedent[:-1]:
        return False
    if s_upper.antecedent != s_lower.antecedent:
        return False
    A_or_B = s_lower.succedent[-1]
    if not A_or_B.formulae_arr[1] == symb.l_or:
        return False
    if A_or_B.quantifier:
        return False
    A = s_upper.succedent[-2]
    B = s_upper.succedent[-1]
    if A != A_or_B.formulae_arr[0]:
        return False
    if B != A_or_B.formulae_arr[2]:
        return False
    return True
```

Функція imp_left: на вхід приймає параметри s_upper1, s_upper2 і s_lower, які приймають значення екземпляру класу Sequence і відповідають за дві верхні та нижню секвенції відповідно.

Програмна реалізація функції imp_left:

```
def imp_left(s_upper1, s_upper2, s_lower):
    if s_upper1.antecedent != s_upper2.antecedent[1:]:
        return False
    if s_upper1.antecedent != s_lower.antecedent[1:]:
        return False
    if s_upper1.succedent[:-1] != s_upper2.succedent:
        return False
    if s_upper1.succedent[:-1] != s_lower.succedent:
        return False
    A = s_upper1.succedent[-1]
    B = s_upper2.antecedent[0]
    A_imp_B = s_lower.antecedent[0]
    if A_imp_B.formulae_arr[1] != symb.l_implication:
        return False
    if A_imp_B.quantifier:
```

```
        return False
    if A != A_imp_B.formulae_arr[0]:
        return False
    if B != A_imp_B.formulae_arr[2]:
        return False
    return True
```

Функція imp_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції imp_right:

```
def imp_right(s_upper, s_lower):
    if s_upper.succedent[:-1] != s_lower.succedent[:-1]:
        return False
    if s_upper.antecedent[1:] != s_lower.antecedent:
        return False
    A_imp_B = s_lower.succedent[-1]
    if not A_imp_B.formulae_arr[1] == symb.l_implication:
        return False
    if A_imp_B.quantifier:
        return False
    A = s_upper.antecedent[0]
    B = s_upper.succedent[-1]
    if A != A_imp_B.formulae_arr[0]:
        return False
    if B != A_imp_B.formulae_arr[2]:
        return False
    return True
```

Функція univ_left: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence та мають у собі предикат і предикатну змінну і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції univ_left:

```
def univ_left(s_upper, s_lower):
    if s_upper.antecedent[1:] != s_lower.antecedent[1:]:
        return False
    if s_upper.succedent != s_lower.succedent:
        return False
    A_upper = s_upper.antecedent[0]
    A_lower = s_lower.antecedent[0]

    if not A_lower.quantifier:
        return False
    if A_lower.quantifier[0] != symb.q_universal:
        return False
    quantified_variable = A_lower.quantifier[1]
    A_lower = s_lower.antecedent[0].formulae_arr[0]
    upper_arr = A_upper.get_P_t_f_var([])
    lower_arr = A_lower.get_P_t_f_var([])
    term = 0
    for i in range(len(upper_arr)):
```

```
        if lower_arr[i] == quantified_variable:
            term = upper_arr[i]
            break
    if not (isinstance(term, Term) or isinstance(term, Variable)):
        return False
    A_lower_text = A_lower.text()
    A_lower_text = A_lower_text.replace(quantified_variable.text(), term.text())
    if A_lower_text != A_upper.text():
        return False
    return True
```

Функція exist_left: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції exist_left:

```
def exist_left(s_upper, s_lower):
    if s_upper.antecedent[1:] != s_lower.antecedent[1:]:
        return False
    if s_upper.succedent != s_lower.succedent:
        return False
    A_upper = s_upper.antecedent[0]
    A_lower = s_lower.antecedent[0]

    if not A_lower.quantifier:
        return False
    if A_lower.quantifier[0] != symb.q_existential:
        return False
    quantified_variable = A_lower.quantifier[1]
    A_lower = s_lower.antecedent[0].formulae_arr[0]
    upper_arr = A_upper.get_P_t_f_var([])
    lower_arr = A_lower.get_P_t_f_var([])
    variable = 0
    for i in range(len(upper_arr)):
        if lower_arr[i] == quantified_variable:
            variable = upper_arr[i]
            break
    if not isinstance(variable, Variable):
        return False
    A_lower_text = A_lower.text()
    A_lower_text = A_lower_text.replace(quantified_variable.text(), variable.text())
    if A_lower_text != A_upper.text():
        return False
    return True
```

Функція univ_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції univ_right:

```
def univ_right(s_upper, s_lower):
    if s_upper.succedent[:-1] != s_lower.succedent[:-1]:
        return False
```

```
    if s_upper.antecedent != s_lower.antecedent:
        return False
    A_upper = s_upper.succedent[-1]
    A_lower = s_lower.succedent[-1]

    if not A_lower.quantifier:
        return False
    if A_lower.quantifier[0] != symb.q_universal:
        return False
    quantified_variable = A_lower.quantifier[1]
    A_lower = s_lower.succedent[-1].formulae_arr[0]
    upper_arr = A_upper.get_P_t_f_var([])
    lower_arr = A_lower.get_P_t_f_var([])
    variable = 0
    for i in range(len(upper_arr)):
        if lower_arr[i] == quantified_variable:
            variable = upper_arr[i]
            break
    if not isinstance(variable, Variable):
        return False
    A_lower_text = A_lower.text()
    A_lower_text = A_lower_text.replace(quantified_variable.text(), variable.text())
    if A_lower_text != A_upper.text():
        return False
    return True
```

Функція exist_right: на вхід приймає параметри s_upper і s_lower, які приймають значення екземпляру класу Sequence і відповідають за верхню та нижню секвенції відповідно.

Програмна реалізація функції exist_right:

```
def exist_right(s_upper, s_lower):
    if s_upper.succedent[:-1] != s_lower.succedent[:-1]:
        return False
    if s_upper.antecedent != s_lower.antecedent:
        return False
    A_upper = s_upper.succedent[-1]
    A_lower = s_lower.succedent[-1]

    if not A_lower.quantifier:
        return False
    if A_lower.quantifier[0] != symb.q_existential:
        return False
    quantified_variable = A_lower.quantifier[1]
    A_lower = s_lower.succedent[-1].formulae_arr[0]
    upper_arr = A_upper.get_P_t_f_var([])
    lower_arr = A_lower.get_P_t_f_var([])
    term = 0
    for i in range(len(upper_arr)):
        if lower_arr[i] == quantified_variable:
            term = upper_arr[i]
            break
    if not (isinstance(term, Term) or isinstance(term, Variable)):
        return False
    A_lower_text = A_lower.text()
    A_lower_text = A_lower_text.replace(quantified_variable.text(), term.text())
    if A_lower_text != A_upper.text():
```

```
        return False
    return True
```

Клас Node: на вхід приймає значення sequence, parent_sequence, descendant_sequence_arr та rule, де sequence – секвенція, над якою проводиться перевірка, parent_sequence – секвенція, з якої була виведена sequence, descendant_sequence_arr – масив секвенцій, які будуть виведені з sequence та rule – правило, за яким виконується виведення. Екземпляри класу Node грають роль вузлів, з яких безпосередньо будується дерево доведення.

Метод класу: check_node – саме за допомогою цього методу виконується перевірка виведення, повертає значення True або False, у залежності від того, чи правильно здійснене виведення формул на цьому вузлі.

Програмна реалізація класу Node:

```
class Node:
    def __init__(self, sequence, parent_sequence, descendant_sequence_arr, rule):
        self.value = sequence
        self.parent_sequence = parent_sequence
        self.descendant_sequence_arr = descendant_sequence_arr
        self.rule = rule

    def check_node(self):
        if self.rule == 'exchange left':
            if len(self.descendant_sequence_arr) != 1:
                return False
            return check.exchange_left(self.descendant_sequence_arr[0], self.value)

        if self.rule == 'exchange right':
            if len(self.descendant_sequence_arr) != 1:
                return False
            return check.exchange_right(self.descendant_sequence_arr[0], self.value)

        if self.rule == 'contraction left':
            if len(self.descendant_sequence_arr) != 1:
                return False
            return check.contraction_left(self.descendant_sequence_arr[0], self.value)

        if self.rule == 'contraction right':
            if len(self.descendant_sequence_arr) != 1:
                return False
            return check.contraction_right(self.descendant_sequence_arr[0], self.value)

        if self.rule == 'weakening left':
            if len(self.descendant_sequence_arr) != 1:
                return False
            return check.weakening_left(self.descendant_sequence_arr[0], self.value)

        if self.rule == 'weakening right':
            if len(self.descendant_sequence_arr) != 1:
                return False
            return check.weakening_right(self.descendant_sequence_arr[0], self.value)
```

```
if self.rule == 'cut':
    if len(self.descendant_sequence_arr) != 2:
        return False
    return check.cut(self.descendant_sequence_arr[0], self.descendant_sequence_arr[1], self.value)

if self.rule == 'not left':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.not_left(self.descendant_sequence_arr[0], self.value)

if self.rule == 'not right':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.not_right(self.descendant_sequence_arr[0], self.value)

if self.rule == 'and left':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.and_left(self.descendant_sequence_arr[0], self.value)

if self.rule == 'and right':
    if len(self.descendant_sequence_arr) != 2:
        return False
    return check.and_right(self.descendant_sequence_arr[0], self.descendant_sequence_arr[1], self.value)

if self.rule == 'or left':
    if len(self.descendant_sequence_arr) != 2:
        return False
    return check.or_left(self.descendant_sequence_arr[0], self.descendant_sequence_arr[1], self.value)

if self.rule == 'or right':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.or_right(self.descendant_sequence_arr[0], self.value)

if self.rule == 'imp left':
    if len(self.descendant_sequence_arr) != 2:
        return False
    return check.imp_left(self.descendant_sequence_arr[0], self.descendant_sequence_arr[1], self.value)

if self.rule == 'imp right':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.imp_right(self.descendant_sequence_arr[0], self.value)

if self.rule == 'univ left':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.univ_left(self.descendant_sequence_arr[0], self.value)

if self.rule == 'exist left':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.exist_left(self.descendant_sequence_arr[0], self.value)

if self.rule == 'univ right':
    if len(self.descendant_sequence_arr) != 1:
        return False
    return check.univ_right(self.descendant_sequence_arr[0], self.value)

if self.rule == 'exist right':
```

```
            if len(self.descendant_sequence_arr) != 1:
                return False
            return check.exist_right(self.descendant_sequence_arr[0], self.value)

        if self.rule == '':
            if self.value.antecedent != self.value.succedent:
                return False
            return True
```

Клас Proof: володіє лише одним полем типу Dictionary і не приймає на вході жодних параметрів. Цей клас і є так званим деревом доведення і реалізувати це допомагають його методи.

Методи класу: add_node, print та check_proof.

add_note – використовується для додавання нових вузлів у дерево доведень, на вхід приймає значення node та level, які відповідають за вузол (екземпляр класу Node) та рівень, на якому він знаходиться відповідно. Перед додаванням нового вузла завжди проводиться перевірка, чи можна додати вузол на цей рівень у поточній конструкції дерева. Якщо вузол буде першим або останнім, то значення батьківського вузла або нащадка повинні бути порожні, всі перевірки проводяться всередині методу.

print – метод виведення текстової форми цілого дерева на екран.

check_proof – головний механізм перевірки дерева доведень на істинність, він перевіряє, чи дане доведення є правильним і повертає значення True або False. При значенні False додатково повертає вузол, у якому було зроблено помилку.

Програмна реалізація класу Proof:

```
class Proof:
    def __init__(self):
        self.proof = {}

    def add_node(self, node, level):
        if level == 0:
            assert node.parent_sequence == None
        if level > 0:
            assert node.parent_sequence in [node.value for node in self.proof['Level ' + str(level - 1)]]
        if 'Level ' + str(level) in self.proof:
            self.proof['Level ' + str(level)] = self.proof['Level ' + str(level)].append(node)
        else:
            self.proof['Level ' + str(level)] = [node]

    def print(self):
        for level in self.proof:
            for node in self.proof[level]:
```

```
            print(node.value.text(), end='', sep='')
            print('(' + node.rule + ')', '', '  ')
        print('')

    def check_proof(self):
        for level in self.proof:
            for node in self.proof[level]:
                if not node.check_node():
                    print("Error here :")
                    print(level)
                    print(node.value.text())
                    desc_arr = [seq.text() for seq in node.descendant_sequence_arr]
                    print(node.rule)
                    for i in range(len(desc_arr)):
                        print(desc_arr[i], sep='   ', end='   ')
                    return False
        return True
```

РОЗДІЛ 4. РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТІВ

Задля перевірки написаної програми було проведено 2 експерименти, де в першому експерименті правильне доведення, а в другому навмисно зроблено помилку.

4.1 Експеримент перший

Розглянемо доведення секвенції $\exists y(\forall x(P(x,y))) \vdash \forall x(\exists y(P(x,y)))$:

- $\exists y\left(\forall x(P(x,y))\right) \vdash \forall x\left(\exists y(P(x,y))\right)$ – (загальності праве)
- $\exists y\left(\forall x(P(x,y))\right) \vdash (\exists y(P(x,y))$ – (існування ліве)
- $\forall x(P(x,y)) \vdash (\exists y(P(x,y))$ – (існування праве)
- $\forall x(P(x,y)) \vdash P(x,y)$ – (загальності ліве)
- $P(x,y) \vdash P(x,y)$

Програмна реалізація:

```
x = Variable('x')
y = Variable('y')
p = AtomicFormulae('P', [x, y])
p = Formulae([], [p])
univ_x_p = Formulae([m.Symb.q_universal, x], [p])
exist_y_p = Formulae([m.Symb.q_existential, y], [p])
exist_y_univ_x_p = Formulae([m.Symb.q_existential, y], [univ_x_p])
univ_x_exist_y_p = Formulae([m.Symb.q_universal, x], [exist_y_p])

Sequence1 = Sequence([exist_y_univ_x_p], [univ_x_exist_y_p])
Sequence2 = Sequence([exist_y_univ_x_p], [exist_y_p])
Sequence3 = Sequence([univ_x_p], [exist_y_p])
Sequence4 = Sequence([univ_x_p], [p])
Sequence5 = Sequence([p], [p])

Node1 = Node(Sequence1, None, [Sequence2], 'univ right')
Node2 = Node(Sequence2, Sequence1, [Sequence3], 'exist left')
Node3 = Node(Sequence3, Sequence2, [Sequence4], 'exist right')
Node4 = Node(Sequence4, Sequence3, [Sequence5], 'univ left')
Node5 = Node(Sequence5, Sequence4, [], '')

Proof1 = Proof()

Proof1.add_node(Node1, 0)
Proof1.add_node(Node2, 1)
Proof1.add_node(Node3, 2)
Proof1.add_node(Node4, 3)
Proof1.add_node(Node5, 4)
Proof1.print()
print(Proof1.check_proof())
```

Результати роботи програми:

```
(∃y)((∀x)(P(x,y)))  ⊢  (∀x)((∃y)(P(x,y)))(univ right)

(∃y)((∀x)(P(x,y)))  ⊢  (∃y)(P(x,y))(exist left)

(∀x)(P(x,y))  ⊢  (∃y)(P(x,y))(exist right)

(∀x)(P(x,y))  ⊢  P(x,y)(univ left)

P(x,y)  ⊢  P(x,y)()

True
```

4.2 Експеримент другий

Розглянемо доведення секвенції, у якій була допущена помилка

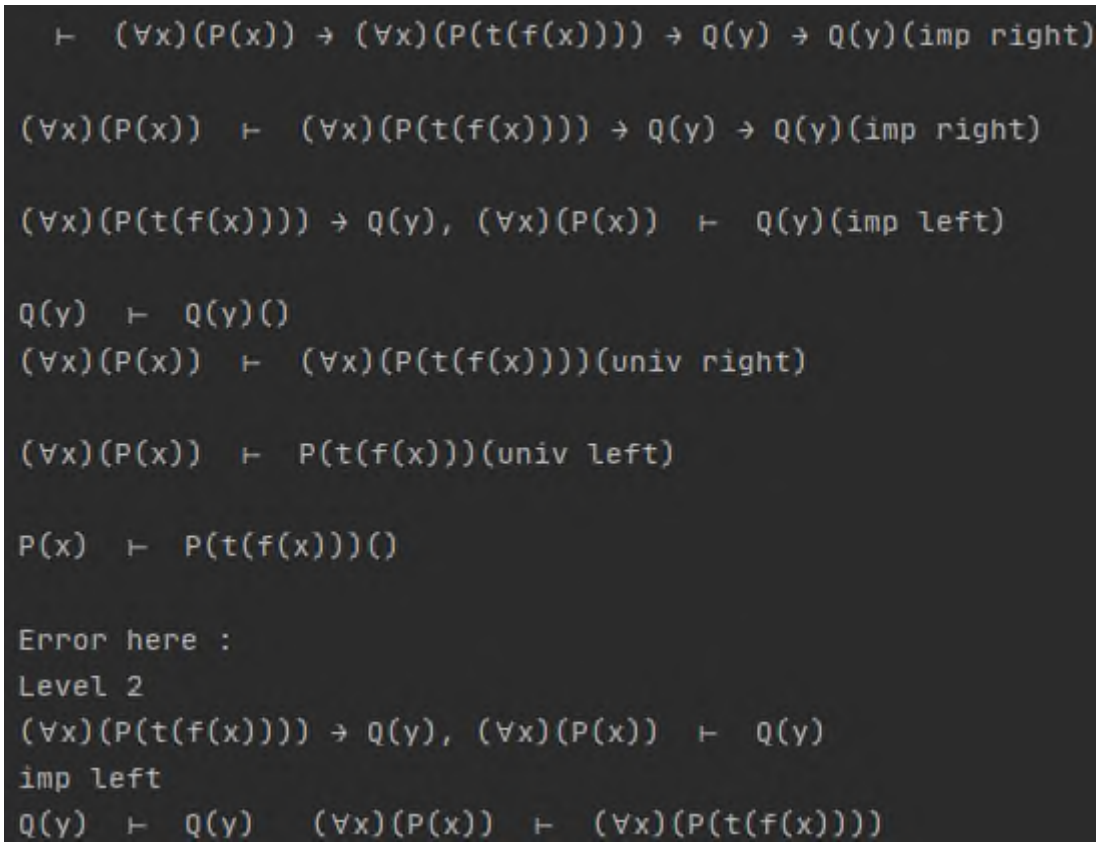
- $\vdash \forall x\big(P(x)\big) \to \Big(\forall x\Big(P\big(f(x)\big)\Big) \to Q(y)\Big) \to Q(y)$:
- $\vdash \forall x\big(P(x)\big) \to \Big(\forall x\Big(P\big(f(x)\big)\Big) \to Q(y)\Big) \to Q(y)$ – (імплікації праве)
- $\forall x\big(P(x)\big) \vdash \Big(\forall x\Big(P\big(f(x)\big)\Big) \to Q(y)\Big) \to Q(y)$ – (імплікації праве)
- $\forall x\big(P(x)\big), \Big(\forall x\Big(P\big(f(x)\big)\Big) \to Q(y)\Big) \vdash Q(y)$ – (імплікації ліве)
  - $Q(y) \vdash Q(y)$
  - $\forall x\big(P(x)\big) \vdash \forall x\Big(P\big(f(x)\big)\Big)$ – (загальності праве)
- $\forall x\big(P(x)\big) \vdash P\big(f(x)\big)$ – (загальності ліве)
- $P\big(f(x)\big) \vdash P\big(f(x)\big)$

Програмна реалізація:

```
x = Variable('x')
y = Variable('y')
f = Function('f', [x])
f = Term('f', [f])
q = AtomicFormulae('Q', [y])
q = Formulae([], [q])
p_f = AtomicFormulae('P', [f])
p = AtomicFormulae('P', [x])
p_f = Formulae([], [p_f])
p = Formulae([], [p])
univ_p = Formulae([q_universal, x], [p])
univ_p_f = Formulae([q_universal, x], [p_f])
```

```
imp_univ_p_f_Q = Formulae([], [univ_p_f, l_implication, q])
imp_imp_univ_p_f_Q_Q = Formulae([], [imp_univ_p_f_Q, l_implication, q])
imp_imp_univ_p_univ_p_f_Q_Q = Formulae([], [univ_p, l_implication, imp_imp_univ_p_f_Q_Q])

Sequence1 = Sequence([], [imp_imp_univ_p_univ_p_f_Q_Q])
Sequence2 = Sequence([univ_p], [imp_imp_univ_p_f_Q_Q])
Sequence3 = Sequence([imp_univ_p_f_Q, univ_p], [q])
Sequence4_1 = Sequence([q], [q])
Sequence4_2 = Sequence([univ_p], [univ_p_f])
Sequence5 = Sequence([univ_p], [p_f])
Sequence6 = Sequence([p], [p_f])

Node1 = Node(Sequence1, None, [Sequence2], 'imp right')
Node2 = Node(Sequence2, Sequence1, [Sequence3], 'imp right')
Node3 = Node(Sequence3, Sequence2, [Sequence4_1, Sequence4_2], 'imp left')
Node4_1 = Node(Sequence4_1, Sequence3, [], '')
Node4_2 = Node(Sequence4_2, Sequence3, [Sequence5], 'univ right')
Node5 = Node(Sequence5, Sequence4_2, [Sequence6], 'univ left')
Node6 = Node(Sequence6, Sequence5, [], '')

Proof1 = Proof()

Proof1.add_node(Node1, 0)
Proof1.add_node(Node2, 1)
Proof1.add_node(Node3, 2)
Proof1.add_node(Node4_1, 3)
Proof1.add_node(Node4_2, 3)
Proof1.add_node(Node5, 4)
Proof1.add_node(Node6, 5)

Proof1.print()
Proof1.check_proof()
```

Результати роботи програми:



Помилка виявлена у 3 вузлі $\forall x(P(x)), \left(\forall x \left(P(f(x))\right) \rightarrow Q(y)\right) \vdash Q(y)$, так як було неправильно застосоване правило лівої імплікації і щоб довести це доведення, потрібно застосувати більше перетворень.

ВИСНОВОК

У результаті цієї роботи вдалося створити застосунок, який дозволяє користувачу перевіряти дерева доведень у численні предикатів. Такого роду програми для автоматизованої перевірки доведень можуть бути дуже корисним інструментом під час роботи з логікою предикатів та численням секвенцій, особливо при зростанні довжини дерева доведення та складності виведень у цілому. Ця програма буде хорошим помічником як і науковцям, які стикнулися з перевіркою доведення певної теореми, так і викладачам та студентам, які вивчають новий матеріал чи викладають матеріал.

На жаль, використання цієї програми потребує певних навичок та знань мови програмування і в загальному інтерфейс не є інтуїтивним, також повинен виконуватися ряд умов та правил введення доведення у програму задля її стабільної роботи. Проте ця робота має великий ряд логічних продовжень та покращень у вигляді: використання можливостей штучного інтелекту, задля повноцінного автоматичного доведення теорем, створення алгоритму для перетворення різного роду текстової теорії чи теореми у вигляд логіки предикатів та, очевидно, створення більш інтуїтивного та зручнішого інтерфейсу програми, з подальшим його перетворенням у веб-застосунок.

Список джерел

1. https://en.wikipedia.org/wiki/Lindström%27s_theorem
2. https://en.wikipedia.org/wiki/Löwenheim–Skolem_theorem
3. http://www.cyb.univ.kiev.ua/library/training-materials/discrete-mathematics/foundations-of-mathematical-logic.pdf
4. Samuel R. Buss: Handbook of proof theory / Samuel R. Buss, S. Abramsky, S. Artemov, R.A. Shore, A.S. Troelstra.- Departments of Mathematics and Computer Science, University of California, San Diego La Jolla, California 92093-0112, USA - 1988. – 811 p.
5. Нікольський Ю.В. Дискретна математика: /Ю.В. Нікольський, В.В. Пасічник, Ю.М. Щербина. - Львів.: Видавництво "Магнолія - 2006", 2013. - 432 с.
6. Дрозд Ю.А. Основи математичної логіки: навч. посібник /Ю.А. Дрозд. - К.: Київський університет, 2003. - 96 с.
7. Logic as a tool (S. Abramsky, S. Artemov, R. A. Shore, A. S. Troelstra), - Stockholm University, Sweden, 2016. – 358 p.