Міністерство освіти і науки України
Національний університет водного господарства
та природокористування

М. В. Бойчура, А. І. Сидор

ЛОГІЧНЕ ПРОГРАМУВАННЯ НА МОВІ PROLOG

Навчальний посібник

Рівне – 2024

УДК 004.42(075)
Б77

Рецензенти:
Івасьєв Степан Володимирович, к.т.н., доцент, доцент кафедри кібербезпеки Західноукраїнського національного університету;
Турбал Юрій Васильович, д.т.н., професор, професор кафедри комп'ютерних наук та прикладної математики Національного університету водного господарства та природокористування.

Рекомендовано вченою радою Національного університету водного господарства та природокористування. Протокол № 4 від 24 квітня 2024 р.

Бойчура М. В., Сидор А. І.
Б77 Логічне програмування на мові Prolog : навч. посіб. [Електронне видання]. – Рівне : НУВГП, 2024. – 142 с.
ISBN 978-966-327-590-1

Навчальний посібник присвячений вивченню концепції декларативного програмування на прикладі мови логічного програмування Prolog. Обґрунтовано вибір саме цієї мови, викладено її синтаксис, що включає структуру програми, принципи побудови складених правил, складених доменів, доменів з альтернативами, різні види повторюваних обчислень (включаючи рекурсію), списки та рядки. Посібник характеризується великою кількістю прикладів програмних кодів стосовно викладених тем, а також детальним роз'ясненням ходу обчислень. Окрім того наведено контрольні запитання до кожного з розділів.

Посібник призначений для студентів-магістрів спеціальності 123 «Комп'ютерна інженерія» у межах першого модуля курсу Логічне та функціональне програмування. Може бути корисним у вивченні концепції декларативного програмування й іншими особами при реалізації задач штучного інтелекту.

УДК 004.42(075)

ISBN 978-966-327-590-1

ЗМІСТ

ВСТУП

Процес людського мислення включає прийняття певних рішень на основі аналізу попередньо відомих фактів про цей Світ. При цьому процес обдумування базується на основі набутого раніше досвіду, який формально можна описати так: «Якщо … і …, то …». Звичайно ж велика кількість рішень приймається, свідомо не обдумуючи всі деталі, а інтуїтивно й майже моментально. Всі ці принципи у певній мірі та сенсі реалізовані у мові логічного програмування Prolog (PROgramming in LOGic). Іншими словами мова Prolog дозволяє імітувати процес мислення людини, базуючись на логічних умовиводах.

Процес написання будь-якої програми на Prolog передбачає наступне:

- декларування низки фактів (аксіом);
- декларування низки правил, які посилаючись на вже відомі факти (і навіть інші правила), дозволяють прийняти логічне рішення;
- задання цільової мети, тобто того, що ми конкретно хочемо дізнатись саме у даний момент.

Процес здійснення умовиводів («мислення») програми насправді не передбачає опису детального алгоритму роботи, – низка речей відбувається без втручання користувача (порівнюючи із людським мисленням – підсвідомо).

Таким чином даний навчальний посібник присвячений питанню програмування (імітування) мислиннєвої діяльності за допомогою логічних умовиводів на мові Prolog.

У найпростіших випадках за допомогою логічних умовиводів Prolog дозволяє визначати усі цикли в графах,

шукати всіх n-рідних братів деякої особи у генеалогічному дереві тощо. Тоді як у складніших випадках при великій кількості даних можливо робити логічні умовиводи у криміналістиці, ставити діагнози пацієнтам тощо.

Мова Prolog належить до категорії декларативних мов програмування. Це означає, що замість звичних детальних алгоритмічних конструкцій з вираженою наказовістю характеру, властиву популярним мовам C++, C#, Java тощо, парадигма декларативного програмування передбачає описання самої проблеми і очікуваного результату, без детальної реалізації алгоритму розв'язання (низка дій здійснюється «автоматизовано»). Наприклад, тут немає таких термінів, як ітерація чи змінна (у традиційному розумінні), присвоєння тощо.

Цікаво, що до декларативних мов належать також мова структурних запитів SQL, мова розмітки гіпертексту HTML тощо. Але вони не володіють повнотою за Тюрінгом (тобто не всі обчислювальні задачі можливо вирішити), на відміну від Prolog. Тому дана мова у достатній мірі підходить для вивчення парадигми декларативного програмування.

Навчальний посібник складається із шести розділів у відповідності до шести (половини) лекцій курсу Логічне і функціональне програмування.

У першому розділі проаналізовано відмінності між імперативними та декларативними мовами програмування, здійснено короткий огляд декларативних мов, на прикладі наведено елементи логіки предикатів, обґрунтовано для подальшого розгляду вибір саме логічної мови програмування Prolog, здійснено опис середовища розробки Visual Prolog v. 5.2.

Другий розділ присвячений поясненню базового синтаксису мови Prolog, включаючи структуру програми,

формалізовано (запрограмовано) приклад, наведений у першому розділі, розглянуто складніший випадок програми на мові Prolog.

В межах третього розділу систематизовано і поглиблено знання, отримані у другому розділі, а також вивчено такі конструкції, як складені правила, складені домени і домени з альтернативами.

Про способи управління повторюваними обчисленнями, включаючи основний вид такого роду конструкцій – рекурсію, йдеться у розділі 4. Окрім того, на прикладах продемонстровано підхід до відлагодження програми.

П’ятий розділ присвячений одній з найважливіших конструкцій у Prolog – списки. Там описаний синтаксис роботи зі списками та наводяться програмні коди основних (найпоширеніших) алгоритмів обробки списків.

У шостому розділі дещо поглиблено знання по роботі з рядками, перелічено, описано та досліджено стандартні твердження по роботі з рядками, наведені приклади програмної реалізації основних (найпоширеніших) алгоритмів обробки рядків.

Решта курсу Логічне і функціональне програмування стосується функціонального програмування, якому й буде присвячений наступний посібник авторів.

Логічне програмування викладається у магістрів спеціальності 123 «Комп’ютерна інженерія» в межах курсу Логічне і функціональне програмування. Матеріали, викладені у навчальному посібнику (вміст розділів, контрольні запитання, приклади кодів програм тощо), готувались у ході викладання дисципліни та аналізувались зі студентами. Тому автори сподіваються, що даний посібник стане студентам у нагоді при вивченні згаданого курсу, і допоможе осягнути особливості декларативного програмування загалом.

Розділ 1. Теорія та засоби логічного програмування

1.1. Імперативні та декларативні мови програмування

Хоча даний навчальний посібник присвячений лише логічному програмуванню, все ж, нелишнім є зазначити, що, в межах відповідного предмету Логічне і функціональне програмування, передбачені 2 модулі: Логічне програмування і Функціональне програмування. Аналогічно предмет передбачає вивчення двох мов програмування. Перша з них – це, однозначно, Prolog (PROgramming in LOGic), оскільки, на сьогодні, не такий вже й великий є вибір серед мов логічного програмування. Тоді як із мовами функціонального програмування – ситуація значно краща. По перше, елементи функціонального програмування наявні у низці найпопулярніших сучасних мов програмування (наприклад, C#, Java, Python тощо). По друге, згідно низки статистичних оцінювань, такі функціональні мови програмування, як Go, Lua, Scala, F#, Haskell входять до двадцятки найпопулярніших мов програмування.

Загалом існують 2 великі класи мов програмування: імперативні та декларативні. До імперативних мов відносяться: C++, C#, Pascal, Java, Delphi, Basic тощо. Імперативні мови характеризуються наказовістю написання відповідного коду (вираженим наказовим характером). Наприклад, «Програма, зроби те!», «Програма, передай це!», «Програма, поверни те!». Наочний приклад на мові C++:

```
for (int i = 0; i < 10; ++i)
{
	cout << i << endl;
}
```

Тут оголошуємо цикл `for`. Далі «наказуємо» програмі оголосити змінну `i`, присвоїти їй значення `0`; «наказуємо»

програмі працювати до того часу, поки `i < 10`; здійснюємо збільшення `i`. У тілі циклу «наказуємо» вивести `i` та перейти на наступний рядок. Таким чином реалізовано детальний алгоритм по виведенню чисел від 1 до 10 на екран.

У декларативних мовах загалом – все по-іншому. У багатьох таких мовах відсутнє поняття циклу, змінної (у традиційному розумінні), присвоювання тощо. Дана парадигма передбачає описання самої проблеми і очікуваного результату, без детальної реалізації алгоритму розв'язання (низка дій здійснюється «автоматизовано»). Наприклад:

1. У Prolog немає такого поняття, як цикли чи ітерація, проте уніфікаційні підпрограми Prolog й без цього самостійно можуть виконувати подібну функцію, здійснюючи обхід частини або всіх задекларованих раніше даних.

2. Мова структурних запитів SQL також є декларативною. Тут серед величезної кількості даних, без явного доступу до використання циклів, можемо отримати, наприклад, кількість літальних апаратів Лелека-100, які є на аеропорті:

```
SELECT count(*)
FROM airport
WHERE aircraft = 'Лелека-100'
```

3. Мову розмітки гіпертексту HTML інколи, залежно від контексту, називають мовою програмування, а точніше – декларативною мовою програмування. В HTML реалізованою є низка раніше задекларованих тегів, за допомогою яких здійснюється опис того, яку ми хочемо отримати Web-сторінку, не вникаючи в сам процес.

Хоча варто відмітити, що в низці декларативних мова програмування тим чи іншим чином передбачені обхідні шляхи до здійснення циклічного обходу, накопичення значень (переприсвоєння) тощо. Наприклад, у Prolog для реалізації наведеного користуються рекурсією (і пов'язаним із нею механізмом відкату). Рекурсія у Prolog має специфічний, у порівнянні з імперативними мовами програмування, синтаксис. Тут програміст задається запитанням: «Що саме і у якій комбінації потрібно використати серед раніше задекларованого, щоб досягнути бажаної мети?». При цьому, фраза «серед раніше задекларованого» є дуже важливою, оскільки вихідними (апріорними) даними у програмі є задекларовані саме програмістом твердження типу: фрукт яблуко (таким чином, з'являється перший об'єкт у нашій програмі), колір червоний (маємо вже 2 об'єкти, з якими є можливість працювати далі). Далі можемо певним чином прив'язати червоний колір до яблука. Можемо добавити ще властивість яблука: форма кругла. Аналогічно можемо задекларувати об'єкт грушу, колір жовтий, вказати форму груші тощо. Тоді користувач може задати запитання: «Що лежить переді мною червоного кольору і кругле?». Програма, посилаючись на раніше задеклароване факти, має зробити логічний умовивід, що це – яблуко.

Наперед забігаючи, зазначимо, що у мові Prolog є твердження-факти (наприклад, яблуко червоне, Андрій любить апельсини) і твердження-правила (наприклад, якщо об'єкт червоний і круглий, то це яблуко; Андрій їсть апельсини, якщо їх купив). Тоді, для прикладу із яблуком, фразу «Що саме і у якій комбінації потрібно використати серед раніше задекларованого, щоб досягнути бажаної мети?», можна розбити на 3 запитання:

1. Яку маємо мету? Перевірити який фрукт є одночасно

і червоним, і круглим.

2. Що саме потрібно використати серед раніше задекларованого? Усі твердження-факти про назви фруктів, єдине твердження-факт про колір і єдине твердження-факт про форму.

3. У якій комбінації? Послідовно перевіряємо всі фрукти чи вони червоні і (логічне і) чи вони круглі.

Описана послідовність запитань і відповідей дозволяє успішно формалізувати програми на Prolog.

Ще одним прикладом імперативного мислення є наступний:

- купити продукт 1, продукт 2, …;
- помити продукт 1, порізати продукт 2, …;
- зварити продукт 1 і продукт 2, …;
- …

Декларативно може все звучати значно простіше: хочу капусняк.

Таким чином, програмуючи імперативно програміст постійно задається запитанням: «Як щось зробити?». Тоді як у декларативній парадигмі прийнято вказувати: «Що зробити?», – не вникаючи в усі деталі алгоритму. Також варто зазначити, що часто до декларативних мов програмування відносять ще й функціональні мови та мови обмежень. В межах даного навчального посібника їх розглядати не будемо, оскільки про перші з них йтиметься у наступному посібнику, а другі – й так засновані на логічному програмуванні.

1.2. Сфери застосування логічних мов програмування

Найвідомішою мовою логічного програмування є Prolog. Вона, в першу чергу, використовується для програмування інтелектуальних систем (або систем штучного інтелекту). Пов’язано це із тим, що початково у

програмі ми задаємо низку властивостей об'єктів, задаємо зв'язки між ними і на основі цього будуються умовиводи. Під штучним інтелектом ми розумітимемо програмно-апаратний комплекс, який призначений для виконання якихось цілей таким чином, щоб імітувати мисленнєву діяльність людини. Хоча, очевидно, люди часто мислять дуже нелогічно. Логічне програмування стосується лише тієї частини мисленнєвої діяльності людини, яка стосується логіки (виходячи із попередньо задекларованих фактів).

В минулому, коли Prolog був значно затребуванішою мовою програмування, то його із досить великою ефективністю використовували для написання програм класифікації і розпізнавання образів, прогнозування поведінки складних систем (наприклад, прогноз погоди), при стисненні і відновлені даних, при імітації асоціативної пам'яті людини, при розробці логічних ігрових програм (із виграшною стратегією; наприклад, шахи), при діагностиці хвороб (виходячи, наприклад, із переліку симптомів, можна робити логічний умовивід чи хвора людина на Covid-19), при доведенні теорем тощо. Цікаво, що у 80-х роках минулого століття Японія виділяла велике фінансування на розвиток Prolog, вважаючи його мовою майбутнього. Ефективність розробки програм на Prolog зокрема полягає у тому, що, в порівнянні з імперативними мовами програмування, у низці випадків, достатнім є написання у рази менше рядків коду. Проте, з іншого боку, Prolog-програми часто потребують від програміста витрачання значної кількості часу для написання всього кількох рядків коду. Тоді як в імперативних мовах, навпаки, доводиться писати велику кількість рядків коду, але «не задумуючись».

1.3. Елементи логіки предикатів

Як вже раніше зазначалось, ми користуватимемось термінами твердження-факт і твердження-правило. Твердження-факти завжди повертають лише значення `Yes` (тобто true), оскільки твердження-факт ми вважаємо аксіомою, від якої надалі відштовхуватимемось, наприклад, при побудові тверджень-правил. Останні можуть повертати як значення `Yes` чи `No`, так і перелік результатів тих чи інших типів даних.

Додатково введемо термін предикат. Предикатом називатимемо шаблон, за яким будуються твердження-факти і твердження-правила. Наприклад, для тверджень Іван любить апельсини, Андрій любить апельсини, Захар любить сливи можна створити єдиний шаблон: хтось любить щось. Тут слово любить називатимемо ідентифікатором відношення, а хтось і щось – властивостями відношення. В залежності від поставленої задачі, властивостей може бути 0 чи 1, чи 2, чи десятки.

Наведемо далі приклад послідовності тверджень, записаних зрозумілою (поки що не формалізованою) людською мовою:

Андрій	любить	апельсини.		
Богдан	любить	яблука.		
Богдан	любить	апельсини.		
Василь	любить	яблука.		
Григорій	любить	те ж саме, що	любить	Богдан
		або	любить	Василь.
Євгеній	любить	те ж саме, що	любить	Богдан
		і	любить	Василь.
Дмитро	любить	те ж саме, що	любить	Григорій,
		але не те, що	любить	Василь.

У результаті логічних умовиводів (аналізу наведених тверджень) робимо наступні висновки:

Богдан любить яблука і любить апельсини.
Андрій любить апельсини.
Василь любить яблука.
Григорій любить яблука і любить апельсини.
Євгеній любить яблука.
Дмитро любить апельсини.

Трансформуємо (формалізуємо) наведені твердження на проміжну мову (ближчу до мови Prolog):

Любить(Андрій, апельсини).
Любить(Богдан, яблука).
Любить(Богдан, апельсини).
Любить(Василь, яблука).
Любить(Григорій, щось), якщо Любить(Богдан, щось)
або Любить(Василь, щось).
Любить(Євгеній, щось), якщо Любить(Богдан, щось)
і Любить(Василь, щось).
Любить(Дмитро, щось), якщо Любить(Григорій, щось)
і не Любить(Василь, щось).

Як бачимо, ідентифікатор твердження Любить поставлено на першу позицію, а в дужки поміщено властивості тверджень. При цьому, наприклад, перше із тверджень-правил варто розуміти наступним чином: Григорій любить щось, якщо Богдан любить те саме щось або (логічне або) Василь любить те саме щось. У наведеному прикладі забезпечено «хороший тон» при перелічуванні тверджень: спочатку розміщені твердження-факти, а потім – твердження-правила. Також варто зазначити, що послідовність властивостей у відношенні є

принциповою для Prolog. Наприклад, якщо ми маємо задекларованe твердження виду: Друг(Жанна, Захар), але не маємо задекларованого твердження Друг(Захар, Жанна), то результатом для Друг(Жанна, Захар) буде `Yes`, а для Друг(Захар, Жанна) – `No`. Тобто якщо ми забажаємо встановити взаємнооднозначне відношення дружби, то доведеться задекларувати пару тверджень.

1.4. Середовище розробки Visual Prolog v. 5.2

Перепишемо тепер наведений вище приклад із використанням саме синтаксису мови Prolog у середовищі розробки Visual Prolog v. 5.2. Спочатку створюємо новий проект. Для цього спершу у меню `Project` обираємо пункт `New Project`…. У вікні, що з'явилось, даємо назву проекту (виходячи з наведеного вище прикладу, даємо назву `Likes`), а в сусідній вкладці `Target` вказуємо у якості `UI Strategy` значення `TextMode`. Це означає, що ми працюватимемо не у «візуальному» режимі, а в звичайному (консольному). Решту параметрів залишаємо без змін. Після чого натискаємо на кнопку Create. У створеному проекті двічі натискаємо на файл Likes.pro (*.pro-файли містять вихідні тексти розроблюваних програм). У ньому вже автоматично міститься інформація про створений проект (у закоментованому вигляді за допомогою синтаксису `/*коментар*/` або `%коментар`) та шаблон найпростішої програми. При цьому, відштовхуючись від назви проекту, автоматично створюється предикат `likes()`, твердження-правило `likes()` і мета `likes()` у відповідних розділах `predicates`, `clauses` та `goal` (даний синтаксис буде пояснено згодом).

Якщо запустити цю програму на виконання (за допомогою клавіші F9 або, що те саме, `Project->Run`), то:

1. По перше, з'явиться вікно із пропозицією безкоштовно зареєструвати екземпляр програми або купити версію Professional Edition. Реєстрація екземпляру програми дозволить позбутися появи даного вікна при кожному новому запуску проекта на виконання.

2. По друге, після натискання на кнопку `Continue Evaluation` на дуже короткий час з'явиться консольне вікно і зникне. Це означає, що програма здійснила вдалий запуск і успішно завершила своє виконання. Щоб, все ж, запобігти автоматичному зниканню консольного вікна доцільно внести деякі модифікації у код програми, про які буде йтись згодом.

Середовище розробки Visual Prolog v. 5.2 надає можливість запуску програми й в режимі відлагоджувача. Для цього можна скористатись комбінацією клавіш Ctrl+Shift+F9 або, що те саме, пунктом меню `Project->Debug`. Головний інструментарій відлагодження реалізується клавішею F7 (`Project->Trace Into`), а саме проходження по рядках коду реалізується у вікні, яке викликається комбінацією клавіш Ctrl+E (`View->Go to Executing Predicate Source`). Серед низки допоміжних вікон для відлагодження варто виділити наступні: `Call Stack` (відкривається за допомогою Alt+6 і `View->Call Stack`) та `Variables For Current Clause` (відкривається за допомогою Alt+7 і `View->Local Variables`).

Варто зазначити, що «логічно» програмувати можна й у зв'язці із іншими мовами програмування, наприклад, виконуючи Prolog-вставки у код C++, C#, Java тощо. Проте загальний синтаксис суттєво ускладняється. Це викликає суттєві незручності саме при вивченні мови Prolog.

Контрольні запитання

1. Які серед перелічених мов програмування є імперативними?

а) C++;

б) Java;

в) F#;

г) Prolog;

д) Lisp.

2. Які серед перелічених мов програмування є декларативними?

а) C++;

б) Java;

в) F#;

г) Prolog;

д) Lisp.

3. Як функціонують цикли у мові Prolog?

а) виконується повний перебір тверджень у порядку їх декларування;

б) виконується повний перебір тверджень у зворотному порядку відносно їх декларування;

в) за замовчуванням цикл складається з єдиної ітерації; у кінці ж твердження необхідно вказати умову продовження циклу;

г) по алфавіту відбувається перебір задекларованих раніше тверджень;

д) у мові Prolog немає такого поняття, як цикли.

4. В чому перевага декларативних мов програмування, у порівнянні з імперативними?

а) швидкість виконання коду;

б) компактність коду;

в) низькорівневість коду;

г) витрати часу на написання рядка коду є мінімальними;

д) очевидним є повний алгоритм роботи програми.

5. Чим відрізняються твердження-факт і твердження-правило?

а) кожне нове правило може будуватись лише на основі існуючих фактів, тоді як факти – це аксіоми;

б) кожне твердження-факт будується із тверджень-правил;

в) твердження-факти завжди повертають лише значення `Yes`, тоді як твердження-правила – інколи `Yes`, а інколи – `No`;

г) твердження-правила завжди повертають лише значення `Yes`, тоді як твердження-факти – інколи `Yes`, а інколи – `No`;

д) вірне використання факту чи правила лише підвищує зрозумілість коду.

6. Що таке предикат?

а) це твердження, яке не потребує доведення;

б) ідентифікатор твердження, істинність якого потрібно довести;

в) послідовність властивостей у відношенні;

г) шаблон, за яким будуються твердження-факти і твердження-правила;

д) проміжна мова програмування.

7. Як поставити коментар у Prolog?

а) `/*коментар*/`;

б) `//коментар`;

в) `<!--коментар--!>`;

г) `*{коментар*}`;

д) `%коментар`.

8. Як запустити відлагоджувач у Prolog?

а) Ctrl+F9;

б) F5;

в) Ctrl+Shift+F9;

г) Project->Debug;

д) F7.

9. Яка клавіша використовується для проходження по рядках коду в процесі відлагодження?

а) Ctrl+F9;

б) F5;

в) Ctrl+Shift+F9;

г) Project->Debug;

д) F7.

10. Як запустити програму на виконання?

а) F9;

б) Ctrl+F9;

в) Ctrl+Shift+F9;

г) F5;

д) Project->Run.

Розділ 2. Програмування на мові Prolog

2.1. Основи синтаксису мови Prolog

Перепишемо, нарешті, наведені вище твердження-факти та твердження-правила із застосуванням синтаксису мови Prolog. При цьому, просте копіювання текстової інформації із, наприклад, Word у Prolog є безсенсовим, оскільки зазвичай редактори Prolog не сприймають українські букви. Також варто бути обережними у процесі редагування тексту, оскільки комбінація клавіш Ctrl+Y може призвести до незвичного видалення вмісту рядка, в якому знаходиться курсор. Комбінація ж клавіш Ctrl+Z працює звично.

Наведені раніше (див. підрозділ 1.3) твердження-факти та твердження-правила помістимо в розділ `clauses` у наступному вигляді:

```
clauses

    likes(andrii, orange).
    likes(bohdan, apple).
    likes(bohdan, orange).
    likes(vasil, apple).
    likes(hrihorii, X)  :-  likes(bohdan, X);
                            likes(vasil, X).
    likes(ievhenii, X)  :-  likes(bohdan, X),
                            likes(vasil, X).
    likes(dmitro, X)    :-  likes(hrihorii, X),
                            not(likes(vasil, X)).
```

Тут, для полегшення візуального сприйняття коду, кожне твердження-правило та твердження-факт наводиться в окремому рядку. Пропонується й в подальших викладках оформлювати код за таким же принципом. Хоча в деяких

випадках, якщо це виправдано, допускається відхід від такої пропозиції.

Нижче пояснено синтаксис тих складових мови Prolog, які стосуються наведеної вище послідовності тверджень.

- Як бачимо, ідентифікатор відношення `likes` записано із малої букви. Це не випадково, оскільки ідентифікатори відношень можуть записуватись лише з малої букви. Аналогічна ситуація із константами символьного (`symbol`) типу `andrii`, `bohdan` тощо. З великої ж букви записуються змінні (`X`, `Y`, `ZZ` тощо). Варто зазначити, що у даному прикладі можна було б скористатись типом `string`, але тему Рядки розглядатимемо лише у наступних розділах.
- Послідовність символів `:-` (замість неї можна друкувати `if`) відповідає зворотній імплікації і означає, що праворуч від `:-` записане якесь правило. Тоді увесь такий рядок називатимемо твердженням-правилом. При цьому, перше із згаданих тверджень-правил можна читати наступним чином: Григорій любить `X`, якщо Богдан любить той самий `X` або Василь любить той самий `X`.
- Символ `;` (або ключове слово `or`) відповідає логічному або.
- Символ `,` (або ключове слово `and`) відповідає логічному і.
- Послідовність символів `not(`твердження`)` відповідає логічному запереченню.
- Кожне твердження-факт і твердження-правило закінчується крапкою.

Окрім наведеного синтаксису ще варто виділити символ нижнього підкреслення (`_`), який називається анонімною

змінною і означає, що замість нього можна підставити будь-яку змінну.

Для забезпечення можливості успішної компіляції програми необхідно задати ще й шаблони (предикати) для всіх тверджень-фактів і тверджень-правил. Здійснюється це у розділі `predicates`.

```
predicates

    likes(symbol, symbol)
```

Тут вказується, що між двома параметрами `symbol` та `symbol` встановлено відношення `likes`.

Якщо ще й запрограмувати певне твердження у розділі `goal`, то можна буде переходити до запуску написаної програми. Хоча у випадку наведеного прикладу може з'явитись помилка `Nondeterministic clause: likes`. Це означає, що предикат `likes` оголошений у стандартному детермінованому режимі, при якому забороняється повернення від тверджень більше, ніж одного рішення. Оскільки, згідно постановки задачі, у нас допускається кілька рішень, то потрібно або зліва від оголошення предиката `likes(symbol, symbol)` добавити ключове слово `nondeterm`, або обрати опцію `Nondeterm` в меню `Options->Project->Compiler options->Warnings->Default Predicate Type`.

У якості прикладів виконання програми в розділ `goal` помістимо, наприклад, одну із наступних послідовностей тверджень (запитань):

1. Хто любить апельсини?

```
likes(X, orange), write(X, " likes orange")
```

Відповіддю буде, очікувано, `andrii likes orange`. Але даний результат ми не встигнемо побачити, оскільки консольне вікно з'явиться лише на мить, і одразу програма успішно завершить свою роботу. Щоб відбувалась затримка добавимо ще твердження `readchar(_)` до попереднього твердження у вигляді:

```
likes(X, orange), write(X, " likes orange"),
    readchar(_).
```

Тепер виведення буде оформлено в очікуваному для користувача вигляді. Тут, як бачимо, змінна `X` – це якась гіпотетична особа, яка любить апельсини. Як тільки серед переліку тверджень знайдено таку особу, то відповідне значення поміщається в змінну `X` (`X=andrii`). Далі застосовується логічне і. Оскільки твердження `likes(X, orange)` є коректним, то переходимо до наступного твердження `write(X, " likes orange")`, в яке підставляємо `X` та виводимо на екран фразу `andrii likes orange`. Далі знову застосовується логічне і. Оскільки результатом виконання твердження `write(X, " likes orange")` виявилось значення `Yes`, то переходимо до твердження `readchar(_)`. Воно очікує від користувача введення (або невведення) довільного символу та, головне, натиснення клавіші enter. Після цього програма коректно завершує свою роботу.

Запис наведеної послідовності тверджень має 2 суттєві недоліки:

- Якщо би ніхто не любив апельсини, то на екран не вивелось би жодного повідомлення. Різні варіанти вирішення такого роду проблем будуть запропоновані у наступних розділах.

- Якщо би було задекларовано декілька любителів апельсин, то відбулось би виведення лише першого із перелічених у розділі `clauses` любителя. Щоб мати можливість виводу низки результатів використовують, так званий, механізм відкату (про нього детально буде йтись мова у підрозділі 3.1), задавши штучне хибне твердження `1=2` (або, що те саме, `fail`). Остаточно рядок у розділі `goal` може мати або такий вигляд:

```
likes(X, orange), write(X, " likes orange"),
    readchar(_), 1=2; 1=1.
```

або такий:

```
likes(X, orange), write(X, " likes orange"),
    nl, 1=2; readchar(_).
```

Тут `nl` – це твердження, яке поміщає у вихідний потік послідовність символів переходу на новий рядок, а штучне твердження `1=1` вставлено спеціально, щоб результат «умовиводів» приймався успішним. Результати будуть наступними:

```
andrii likes orange
bohdan likes orange
hrihorii likes orange
dmitro likes orange
```

2. Що любить Григорій?

```
likes(hrihorii, X), write("Hrihorii likes ",
    X), readchar(_), 1=2; 1=1.
```

У результаті матимемо:

```
Hrihorii likes apple
Hrihorii likes orange
Hrihorii likes apple
```

3. Чи хтось любить апельсини?

```
likes(_, orange), write("Yes"), readchar(_);
   write("No"), readchar(_).
```

Оскільки нам підійде довільна перша із відповідей, то відкат `1=2` тут робити не потрібно. Результат буде наступним:

```
Yes
```

4. Чи любить Василь апельсини?

```
likes(vasil, orange), write("Yes"),
   readchar(_); write("No"), readchar(_).
```

У результаті матимемо:

```
No
```

5. Хто що любить?

```
likes(X, Y), write(X, " likes ", Y),
   readchar(_), 1=2; 1=1.
```

Тут отримується 9 результатів (2 з них просто повторюються, а саме: Григорій любить яблука), які повністю (хоч із надлишковістю) відповідають умовиводам, наведеним у підрозділі 1.3. Механізми вирішення надлишковості будуть запропоновані у

наступних розділах. А стосовно даного прикладу маємо наступні результати:

```
andrii likes orange
bohdan likes apple
bohdan likes orange
vasil likes apple
hrihorii likes apple
hrihorii likes orange
hrihorii likes apple
ievhenii likes apple
dmitro likes orange
```

6. Чи хтось любить щось взагалі?

```
likes(_,    _),    write("Yes"),    readchar(_);
   write("No"), readchar(_).
```

У результаті матимемо:

```
Yes
```

7. Хто любить хтось хоч щось?

```
likes(X,  _),  write(X,  "  likes  something"),
   readchar(_); 1=1.
```

Результати будуть наступними:

```
andrii likes something
bohdan likes something
bohdan likes something
vasil likes something
hrihorii likes something
hrihorii likes something
hrihorii likes something
```

```
ievhenii likes something
dmitro likes something
```

2.2. Структура програми на мові Prolog

Порядок розміщення розділів у програмі є принциповим. При цьому, оголошення чергового розділу сприймається Prolog-ом одночасно й як завершення попереднього.

1. Окрім тих розділів, із якими ми вже ознайомились раніше, можливим є використання ще й такого (не обов'язкового) розділу, як `domains`. Він призначений для оголошення користувацьких типів даних. Синтаксис наступний (тут терм – це назва):

терм домену = опис домену

Наприклад:

1) щоб явно не користуватись типом `symbol` можемо створити його домен-синонім:

```
s = symbol
```

Відтепер у розділі `predicates` маємо можливість замість запису

```
likes(symbol, symbol)
```

використовувати його укорочений вигляд:

```
likes(s, s)
```

У низці випадків це суттєво спрощує процес написання коду і підвищує його «читабельність».

2) можемо перейменувати й, наприклад, стандартний домен `integer`

```
int = integer
```

3) такий тип, як список цілих чисел (детальніше див. розділ 5), оголошується так:

```
lint = integer*
```

4) для оголошення складеного домену (див. підрозділ 3.3) «книжка» взагалі доцільно виконати низку записів:

```
book = b(author, title, year)
author, title = string
year = integer
```

Основні стандартні доменні типи, за допомогою яких, зокрема, будуються користувацькі домени, наведені у табл. 2.1.

Таблиця 2.1

Основні стандартні доменні типи

Домен	Опис	Розмір (байт)	Приклад
`char`	Символ	1 байт	`'A', '\65', '\41'`
`integer`	Цілі числа зі знаком	4 байти	`-2147483648...2147483647`
`real`	Дійсні числа	8 байтів	`5E-324...1.7E+308`
`string`	Рядки символів	До 250 символів; в деяких випадках – до 4Гб	`"AB\65\$44"`

продовження табл. 2.1

`symbol`	Символічні імена та ідентифікатори	–	• Послідовність букв, цифр, символів підкреслення, яка починається з малої букви (`apple`, `andrii`); • Аналогічно до `string`: `"AB\65\$44"`
`file`	Файли	–	Файлові змінні

2. Розділ оголошення предикатів динамічної бази даних `database`. Особливістю даного розділу є те, що відповідні для нього твердження можна додавати та видаляти у процесі виконання програми. Зокрема, є можливість зчитувати/записувати на дисковий простір твердження. Даний розділ є необов'язковим. Синтаксис наступний:

терм предикату [(терм домену[, терм домену[, терм домену[, …[, терм домену]]]]…]

3. Розділ оголошення предикатів статичної бази даних `predicates`. Ми про нього вже згадували вище. Він є обов'язковим розділом. Синтаксис оголошення предикатів є таким же, як і для розділу `database`. Кількість властивостей (термів домену), які ми задаємо для терма предикату, називається арністю. Предикат `likes(s, s)` є 2-арним. Можна, наприклад, оголосити предикат без властивостей. Тоді такий предикат є 0-арним. Хоч і виглядає нелогічним використання такого роду записів, проте у низці випадків (див., напр., підрозділ 4.1) відповідні твердження виявляються необхідними для виконання поставленої задачі. Загалом предикат може бути й 50-арним.

Тут важливо зазначити, що оголошений предикат повинен хоча б 1 раз використатись у розділі `clauses`, інакше – виникне помилка.

4. Розділ оголошення тверджень `clauses`. Даний розділ є обов'язковим. У ньому розміщуються всі твердження-факти і твердження-правила (статичної бази даних), які використовуватимуться у програмі. Порядок декларування тверджень одного і того ж предикату у деяких випадках може бути принциповим. Наприклад, це стосується рекурсії.

Тут є важливий момент. Якщо у розділі `predicates` задати 2 предикати (шаблони для тверджень-фактів і тверджень-правил)

```
nondeterm likes(s, s)
nondeterm student(s, s, integer)
```

а в розділі `clauses` запрограмувати наступне:

```
likes(andrii, orange).
student(andrii, ki51, 95).
likes(bohdan, apple).
student(viktor, ki51, 99).
```

то з'явиться помилка `causes for the same predicate should be grouped student`. Це означає, що усі твердження, які стосуються деякого предиката, повинні бути згруповані (розміщені один біля одного). З одного боку, це викликає певні незручності, але з іншого – дозволяє уникати серйозних помилок.

Одним із прикладів використання 0-арних тверджень є наступний (варто нагадати, що відповідний 0-арний

предикат має бути попередньо оголошеним у розділі predicates):

```
run :- write("Hello world"), readchar(_).
```

Перевірка такого твердження у розділі goal є еквівалентною написанню

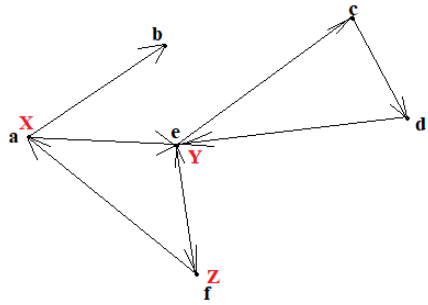
```
write("Hello world"), readchar(_).
```

5. Розділ цільової мети goal. Даний розділ є обов'язковим. У ньому реалізується виклик стандартних та описаних раніше тверджень-фактів і/або тверджень-правил. В найпростішому випадку обирається перше (у розділі clauses) підходяще твердження-факт чи твердження-правило і здійснюється вивід результату у консольному вікні. Проте у випадку використання рекурсії, відкату тощо, вибір підходящих тверджень стає менш тривіальним. Варто зазначити, що у розділі goal може бути лише одна цільова мета (одне твердження, яке закінчується крапкою).

Таким чином, Prolog-програма зазвичай має до 5 розділів (domains, predicates, database, clauses, goal), 3 з яких (predicates, clauses, goal) – обов'язкові. Окрім цього, можливим є задання розділів constants, class, implement, abstract class; facts є синонімом до database.

2.3. Приклад програми «Трикутник»

Наведемо ще один приклад програми на Prolog. Нехай на площині розміщено деяку множину точок. Між ними задано напрямлені відрізки у вигляді стрілок (див. рис.). Необхідно встановити всі можливі трикутники.



Розв’язок

Розв’язок будується із міркувань, що якщо існує шлях від вершини X до вершини Y, від вершини Y до вершини Z і від вершини Z до вершини X, то маємо трикутник.

```
predicates

     nondeterm arrow(symbol, symbol)
     nondeterm triangle(symbol, symbol, symbol)

clauses

     arrow(a, b).
     arrow(a, e).
     arrow(e, c).
     arrow(e, f).
     arrow(f, a).
     arrow(f, e).
     arrow(c, d).
     arrow(d, e).
     triangle(X, Y, Z) :- arrow(X, Y), arrow(Y, Z),
        arrow(Z, X).

goal

     triangle(X, Y, Z), write("Triangle ", X, ", ",
        Y, ", ", Z), readchar(_), 1=2; 1=1.
```

Тут твердження-правило `triangle(X, Y, Z)` можна читати наступним чином:

Фігура із вершинами `X`, `Y`, `Z` є трикутником,
якщо
існує напрямлений відрізок `XY` із початком `X` та кінцем `Y`
і
існує напрямлений відрізок `YZ` із початком `Y` та кінцем `Z`
і
існує напрямлений відрізок `ZX` із початком `Z` та кінцем `X`

Або, згадуючи фразу «Що саме і у якій комбінації потрібно використати серед задекларованого, щоб досягнути бажаної мети?», логіку написання того ж самого твердження-правила пояснимо наведеним нижче чином. Серед задекларованих тверджень-фактів `arrow(…)` використано (що саме?) їх всі у комбінації `arrow(X, Y), arrow(Y, Z), arrow(Z, X)` для знаходження циклічних трикутників (бажаної мети). Або ж згадану фразу можна «розібрати» й на такі 3 запитання з відповідями на них:

1. Яка мета? Знайти циклічні трикутники за допомогою твердження виду `triangle(X, Y, Z)`;
2. Що саме використано із раніше задекларованого? Всі задекларовані твердження-факти доменного типу `arrow(symbol, symbol)`;
3. У якій комбінації? `arrow(X, Y), arrow(Y, Z), arrow(Z, X)`.

Тут замість пар `XY`, `YZ`, `ZX` підставляються всеможливі (серед задекларованих) пари відрізків із використанням механізму відкату (дечим схожого на рекурсію). У результаті маємо 6 відповідей. Хоча кількість циклічних

трикутників рівна 2. Це пов'язано з тим, що механізм відкату (`1=2`) сприяє перебору усіх можливих розв'язків. Погодьтесь, що `aef` та `efa` – це формально різні розв'язки, але фактично – тут йдеться про один і той же трикутник. Механізми вирішення даної проблеми будуть запропоновані у наступному розділі.

Контрольні запитання

1. Яким чином доцільно сформулювати твердження: `a` та `b` – це друзі?

а) `friend(a, b).`;

б) `friend(b, a).`;

в) `friend(a, b, b, a).`;

г) `friend(a, b). friend(b, a).`;

д) `friend(a, a, b, b)..`

2. Що означає помилка `Nondeterministic clause: friend`?

а) предикат `friend` оголошений у нетермінованому режимі;

б) предикат `friend` оголошений у детермінованому режимі;

в) предикат `friend` оголошений у термінованому режимі;

г) предикат `friend` оголошений у терміновому режимі;

д) предикат `friend` оголошений у недискримінантному режимі.

3. Що з переліченого є змінними мови Prolog?

а) X;

б) Zminna;

в) variable;

г) Var;

д) var.

4. Що може означати даний цільовий запит: `friend(X,`

```
Y), write(X, " friend ", Y), readchar(_), 1=2;
1=1.
```

а) вивести всі пари взаємних друзів;

б) вивести будь-якого одного друга;

в) вивести всіх друзів по відношенню до осіб `X`;

г) вивести всіх друзів по відношенню до осіб `Y`;

д) немає вірної відповіді.

5. Який доменний тип відповідає слову `apple`?

а) `char`;

б) `integer`;

в) `real`;

г) `string`;

д) `symbol`.

6. Що може означати запис: `book = b(author, title, integer)`?

а) змінній `book` присвоюємо конкретну книгу;

б) оголошуємо складений доменний тип;

в) порівнюємо 2 книги;

г) даний запис є некоретним для мови Prolog;

д) оголошуємо нове твердження `book`.

7. Які з розділів є обов'язковими?

а) `predicates`;

б) `domains`;

в) `goal`;

г) `database`;

д) `clauses`.

8. У якому з випадків розділ цільової мети записаний невірно?

а) `run :- write("Hello world"). readchar(_).;`

б) `run :- write("Hello world"), readchar(_).;`

в) `run :- write("Hello world"); readchar(_).;`

г) `run :- write("Hello world") and readchar(_).;`

д) `run :- write("Hello world"), 1=2,`

`readchar(_)`..

9. Яка різниця між символом `;` та ключовим словом `or` у Prolog?

а) вказаний символ означає логічне *і*, а `or` – логічне *або*;

б) вказаний символ є ознакою закінчення рядка, а `or` – це логічне *або*;

в) ключового слова `or` немає в Prolog;

г) символу `;` немає в Prolog;

д) у більшості застосуваннях немає різниці.

10. Що потрібно підставити замість `X1`, `X2`, …, `X9` у правило `triangle(X1, X2, X3) :- arrow(X4, X5), arrow(X6, X7), arrow(X8, X9).`, щоб його доцільно було використовувати для пошуку трикутників?

а) `X1=X5=X9`, `X2=X4=X6`, `X3=X7=X8`;

б) `X1=X2=X3`, `X4=X5=X6`, `X7=X8=X9`;

в) `X1=X5=X6`, `X2=X7=X8`, `X3=X4=X9`;

г) `X1=X4=X9`, `X2=X5=X6`, `X3=X7=X8`;

д) `X1=X7=X9`, `X2=X4=X6`, `X3=X5=X8`.

Розділ 3. Детальне вивчення базового синтаксису мови Prolog

У даному розділі детально розглянемо особливості роботи із:

- змінними;
- конструкцією, яка нагадує оператор розгалуження (`if`…`else`) в C++;
- конструкцією, яка нагадує структури (`struct`) в C++;
- конструкцією, яка нагадує об'єднання (`union`) в C++.

Зокрема синтаксиси оголошення та принцип декларування тверджень-фактів для двох останніх конструкцій є інтуїтивно зрозумілими (див. підрозділи 3.3 і 3.4). Складнішою є побудова відповідних тверджень-правил.

Також, для зручності, надалі замість термінів твердження-факт і твердження-правило переважно вживатимуться слова факт і правило.

3.1. Складені правила

Структуру будь-якого правила можна розбити на 3 частини:

голова правила якщо тіло правила

В залежності від того, у якій комбінації використовуємо кон'юнкцію (`,`), диз'юнкцію (`;`), заперечення (`not`) і відсікання (`!`), ми отримуємо ту чи іншу алгоритмічну конструкцію. Ключовими з них є: лінійні (або, що те саме, послідовні) обчислення, обчислення з розгалуженням та повторювані обчислення.

1. Почнемо із найпростішої, на перший погляд, конструкції, а саме: лінійні обчислення. Відповідний приклад задання такої конструкції на мові Prolog має

вигляд (тут `statem` – вважатимемо скороченням від слова statement):

```
rule(X, Y, Z) :- statem1(X, Y), statem2(X, Y),
    statem3(X, Z), statem4(X, Y, Z), statem5.
```

Оскільки даний вираз стосується лінійних обчислень, то, здавалося б, що послідовно мали би перевірятись усі наведені твердження (`statem1`, `statem2`, `statem3`, `statem4`, `statem5`), і у разі хибності одного із них, усе правило вважалось би таким, що є з хибним результатом. Але у мові Prolog при отриманні хибного значення деякого твердження запускається, так званий, механізм відкату в останню точку розгалуження (якщо така є). Таким чином уніфікаційні підпрограми Prolog здійснюють всеможливий перебір комбінацій параметрів тверджень, об'єднаних кон'юнкцією, до тих пір, поки не досягається істинний результат або не вичерпуються наявні варіанти. Іншими словами, уніфікаційні підпрограми Prolog перед кожним розглядуваним у даний момент твердженням у тілі досліджуваного правила ставлять точку розгалуження, яка обирає для перевірки чергове твердження серед раніше задекларованих (із тим же ідентифікатором) у випадку хибності поточного. Схематично деяка «ітерація» правила, тіло якого реалізує лінійні обчислення, може мати вигляд (тут $_{тр}$ – це точка розгалуження):

```
rule   :-  трstatem1,  трstatem2,  трstatem3,
    statem4, statem5.
```

На мові C++ подібна послідовність обчислень може бути представленою за допомогою циклу `foreach` у вигляді:

```
foreach (item1 in statem1)
{
    if (item1 == false) continue;
    foreach (item2 in statem2)
    {
        if (item2 == false) continue;
        foreach (item3 in statem3)
        {
            if (item3 == false) continue;
            foreach (item4 in statem4)
            {
                if (item4 == false) continue;
                foreach (item5 in statem5)
                {
                    if (item5 == true) return true;
                }
            }
        }
    }
}
return false;
```

Тобто перевірка (виконання) правила `rule` відбувається у наступній послідовності:

1) перевіряється чергове твердження (спочатку `statem1`);

2) якщо дане твердження є вірним (`Yes`), то переходимо до перевірки наступного по порядку твердження, інакше – повторюємо перевірку поточного твердження, але вже при інших параметрах (взятих, наприклад, із задекларованого раніше переліку правил);

3) така послідовність перевірок тверджень здійснюється допоки не досягається один з двох випадків:

- вичерпано всі комбінації параметрів задекларованих у програмі тверджень (фактів і правил) `statem1`, `statem2`, `statem3`, `statem4`, `statem5` (здійснено всеможливий перебір), й так і не досягнуто істинного результату; тоді правило `rule` набуває остаточно хибного результату;
- здійснено перевірку тверджень `statem1`, `statem2`, `statem3`, `statem4`, `statem5`, і при деякому наборі

параметрів (значень змінних) досягнуто істинного результату; тоді правило `rule` набуває остаточно істинного результату.

2. Перейдемо тепер до обчислень із розгалуженням. Як відомо, розгалуження в C++ виглядає, наприклад, так:

```
if (cond) operator1 else operator2;
```

Спробуємо переписати дану конструкцію на Prolog:

```
rule(X) :- cond(X), statem1; not(cond(X)),
    statem2.
```

Як бачимо, якщо умова `cond(X)` є істинною, то, оскільки далі слідує кон'юнкція, переходимо до перевірки твердження `statem1`. У даному випадку, диз'юнкція у Prolog (і в багатьох інших мовах програмування) працює так, що все, що знаходиться після `;` виконуватись не буде. Тобто перша частина тіла правила завжди функціонуватиме дещо схоже до варіанта на C++. Якщо ж `cond(X)` є хибною, то, здавалося б, відбуватиметься перехід до перевірки твердження `not(cond(X))`, після чого виконуватиметься твердження `statem2`. Але насправді на операцію `not` компілятор накладає низку обмежень, наприклад, щодо некоректності використання вільної змінної (змінної без значення) у тілі `not`.

З іншого боку, у тілі правила `rule` насправді здійснюється пошук такої комбінації параметрів серед задекларованих раніше тверджень, при якій досягається істинний результат, із використанням механізму відкату (всеможливого перебору). Це не відповідає наведеному вище прикладу оператора `if…else`. Таким чином, замість наведеного правила `rule` запишемо наступний вираз:

```
rule(X) :- cond(X), statem1, !; statem2, !.
```

У даному випадку використано операцію відсікання (!). Вона видаляє усі утворені раніше в межах правила rule точки розгалуження, чим унеможливлює здійснення всеможливих переборів. Це є подібним до того, що у вкладених foreach у потрібному місці застосувати оператор return. Результатом виконання операції ! є Yes.

Таким чином, при перевірці правила rule створюється точка розгалуження перед cond(X), і далі відбувається наступне:

- перевіряється на істинність cond(X);
- якщо умова cond(X) є істинною, то виконується твердження statem1 і видаляється створена раніше точка розгалуження;
- якщо умова cond(X) є хибною, то виконується твердження statem2 і видаляється створена раніше точка розгалуження.

Тобто записане правило відповідає тому, що ми очікували (забезпечена схожість на оператор if…else).

Розглянемо випадок використання оператора розгалуження із вкладеннями на мові C++:

```
if (cond) if (cond1) operator11 else operator12;
else
          if (cond2) operator21 else operator22;
```

Спробуємо наведений оператор розгалуження записати із використанням синтаксису мови Prolog:

```
rule :- cond, cond1,statem11,!; statem12,!,!;
              cond2,statem21,!; statem22,!,!.
```

У даному випадку, ніколи не виконається ні statem21, ні statem22, оскільки хибність умови cond призведе лише до виконання твердження statem12. Використання ж дужок виду

```
rule :- cond,(cond1,statem11,!; statem12,!),!;
             (cond2,statem21,!; statem22,!),!.
```

у Prolog заборонене. Тому замість виразів у дужках варто створювати окремі правила у вигляді:

```
rule11 :- cond1, statem11, !; statem12, !.
rule21 :- cond2, statem21, !; statem22, !.
rule :- cond, rule11, !; rule21, !.
```

Цікаво, що останнє записане правило є аналогом описаного раніше правила:

```
rule(X) :- cond(X), statem1, !; statem2, !.
```

Тобто 1 вкладений оператор розгалуження звели до трьох простих.

3. Обчислення з повторенням. Як вже згадувалось у попередніх розділах, повторювані обчислення у Prolog можна здійснювати як із застосуванням рекурсій, так і з застосуванням механізму відкату. Принциповою відмінністю цих двох способів є те, що при відкаті до точки розгалуження відповідні змінні звільняються від своїх значень. Це є завадою для проведення обчислень із накопиченням значень (що є принциповим, наприклад, при пошуку факторіала числа). Тоді як для рекурсій немає таких обмежень. Більш детально про повторювані обчислення йтиметься у розділі 4.

4. Використання заперечень (not). Не зважаючи на очевидність застосування заперечень на C++, на мові Prolog накладаються певні обмеження на цей механізм. Не можна застосовувати not щодо вільних змінних (змінних без значення). Також небажаним є застосування not на початку тіла твердження. Тобто замість

```
rule(X) :- not(statem1(X)).
```

краще, якщо це є алгоритмічно доцільним, записати щось подібне до

```
rule(X) :- statem1(X), not(X = ...).
```

Більше того, на початковому етапі вивчення мови Prolog бажано уникати застосування not.

3.2. Змінні у Prolog

Дослідимо глибше яким чином варто застосовувати змінні у Prolog.

Як вже згадувалось раніше, змінні у Prolog записуються із великої букви (наприклад, X, YY, Zminna). В момент оголошення змінної X її тип не задається і не відбувається ініціалізації її значення. Тобто це є вільна змінна. Як тільки виконується, наприклад,

```
X=5
```

змінній X надається тип integer, їй присвоюється значення 5, а сама змінна відтепер є змінною зі значенням. Таким чином, змінна може перебувати в одному з двох станів: вільна змінна і змінна зі значенням. Із першого до другого стану змінна може перейти лише у результаті виконання

операції присвоєння, а в зворотній бік – у результаті застосування механізму відкату (організувати процес переприсвоєння даним інструментарієм не вдасться).

Цікаво, що залежно від ситуації, в якій ми застосовуємо знак = як до змінних, так і до констант, матимуть місце різні випадки реакції уніфікаційних підпрограм Prolog. Розглянемо низку відповідних тверджень (для зручності у трьох наступних прикладах під послідовністю символів := розумітимемо присвоєння, а під == – порівняння):

```
1. X=5, Y=X, Y=3.          %X:=5, Y:=5, Y==3
2. X=Y.                    %error
3. X=5, X=5+0.             %X:=5, X==5
```

У випадку 1 для змінних X та Y послідовно присвоюється значення 5, після чого відбувається порівняння значення змінної Y та числа 3. Через хибність останнього твердження усе твердження завершується із остаточно хибним значенням (No). У випадку 2 взагалі має місце помилка. Таким чином:

- застосування операції = між вільною змінною і константою призводить до присвоєння значення цієї змінної;
- застосування знаку = між вільною змінною і змінною із значенням призводить до присвоєння значення цієї змінної;
- застосування знаку = між змінною зі значенням і константою означає порівняння відповідних значень;
- застосування операції = є неприпустимим, якщо обидва операнди є вільними змінними.

Окремий випадок у роботі зі змінними, на який варто звернути увагу – це використання змінних у якості аргументів тверджень. Наприклад:

```
clauses

    rule(X, Y, 5, 6). %X = 3, ..., 6=6

goal

    rule(3, Y, Z, 6). %Z = 5
```

Уніфікаційні підпрограми Prolog забезпечать реагування на такий запис наступним чином:

- при передачі константи у позицію розміщення вільної змінної відбудеться присвоєння (`X=3`);
- при передачі вільної змінної у позицію розміщення вільної змінної нічого принципово важливого не відбудеться;
- внаслідок передачі вільної змінної у позицію розміщення константи при істинності твердження `rule` відбудеться присвоєння (`Z=5`);
- при передачі константи у позицію розміщення константи відбудеться порівняння констант (`6=6`).

Як бачимо, на передачу параметрів у твердження і на застосування операції `=` у виразах Prolog реагує дещо по-різному.

3.3. Складені домени

Розглянемо далі складніші випадки задання доменних типів, а саме: складені домени та домени з альтернативами.

Складені домени схожі до структур (`struct`) у C++. Доцільність (і корисність) застосування складених доменів у Prolog випливає:

- або із зручності (у багатьох випадках) поміщення відношення структурування в інше складне за структурою відношення;
- або при бажанні конкретизувати подальші факти, які використовуватимуться у розділі `clauses`.

Загальна схема оголошення складеного домену наступна:

терм домену = функтор(домен, домен, домен)

Тут терм домену – це назва (ідентифікатор) нового доменного типу, яка починається з малої букви, а функтор – це терм, який також задається програмістом. При цьому, для терму домену і для функтора можна задавати як однакові, так і різні назви. Уніфікаційні підпрограми Prolog самостійно визначать по контексту входження де терм домену, а де – функтор. Також у якості властивостей функтора можуть виступати інші складені домени.

Наприклад:

1. Якщо маємо справу із книжками, то оголошення складеного домену може мати вигляд:

```
domains

    book = book(author, title, price)
    author, title = string
    price = real
```

Тут ми створили структуру даних `book` із її властивостями `author`, `title`, `price`. При цьому, якщо домени функтора є

користувацькими (не є стандартними), то їх необхідно довизначати обов'язково.

2. Припустимо, що має місце двозначність висловлень про те, якої природи ключ бачить людина, тобто що має місце наступний фрагмент програми:

```
predicates

     see(string, string)

clauses

     see("Andrii", "kluch").
     see("Janna", "kluch").
     see("Zakhar", "kluch").
```

Не виключено, що Андрій є музикантом і в даний момент бачить скрипічний ключ; щодо Жанни, можливо, у контексті ключа маються на увазі птахи; Захар же може бачити ключ від дверей. Тому розглянемо наступний приклад застосування складених доменів (для випадку конкретизації):

```
domains

     music=music(string)
     door=door(string)
     birds=birds(string)

predicates

     seeMusic(string, music)
     seeBirds(string, birds)
     seeDoor(string, door)

clauses
```

```
seeMusic("Andrii", music("kluch")).
seeMusic("Zakhar", music("sol")).
seeBirds("Janna", birds("kluch")).
seeDoor("Zakhar", door("kluch")).
seeDoor("Andrii", door("sol")). % у сенсі: сіль
seeDoor("Zakhar", door("sol")).
```

У якості прикладів цільових запитів наведемо наступні:

1. Які музичні символи бачить Андрій?

```
seeMusic("Andrii", music(X)), write(X), nl,
    1=2; readchar(_).
```

Відповідь:

```
kluch
```

2. Хто бачить сіль?

```
seeMusic(X, music("sol")), write(X), nl, 1=2;
seeBirds(X, birds("sol")), write(X), nl, 1=2;
seeDoor(X, door("sol")), write(X), nl, 1=2;
readchar(_).
```

Відповідь:

```
Zakhar
Andrii
Zakhar
```

3. Що бачить Андрій?

```
seeMusic("Andrii", X), write(X), nl, 1=2;
seeBirds("Andrii", X), write(X), nl, 1=2;
seeDoor("Andrii", X), write(X), nl, 1=2;
```

```
readchar(_).
```

Відповідь:

```
music("kluch")
door("sol")
```

Таким чином, отримали однозначність у твердженнях, не зважаючи на таку велику кількість використаних доменів і предикатів. Хоча, здавалося б, навіщо вводити таку велику їх кількість. Але, по перше, у низці випадків такий уточнюючий функціонал може бути дуже корисним, а, по друге, складені домени плавно підводять нас до наступної теми: домени з альтернативами (за допомогою яких можна значно спростити наведений вище приклад).

3.4. Домени з альтернативами

Синтаксис оголошення домену з альтернативами може мати вигляд:

терм домену = функтор1(домен, домен, …);
функтор2(домен, домен, …);
функтор3(домен, домен, …)

Тут назви функторів можуть співпадати як між собою, так і з термом домена. Важливо лише, щоб уніфікаційні підпрограми Prolog могли розрізнити різні функтори по контексту входження.

Даний вид доменних типів схожий на `union` в C++. За допомогою цього інструментарію предикати `seeMusic`, `seeBirds`, `seeDoor` можна уніфікувати в єдиний предикат `see`, а у якості його другого параметра доцільно вказати новий терм домену (з альтернативами) `seen`. Наприклад:

```
domains

     seen = music(string);
            door(string);
            birds(string)

predicates

     see(string, seen)

clauses

     see("Andrii", music("kluch")).
     see("Zakhar", music("sol")).
     see("Janna", birds("kluch")).
     see("Zakhar", door("kluch")).
     see("Andrii", door("sol")).
     see("Zakhar", door("sol")).
```

Нижче наведено ті ж приклади цільових запитів, що були у темі Складені домени, але вже для випадку використання доменів з альтернативами:

1. Які ноти бачить Андрій?

```
see("Andrii", music(X)), write(X), nl, 1=2;
    readchar(_).
```

2. Хто бачить соль?

```
see(X, music("sol")), write(X), nl, 1=2;
see(X, birds("sol")), write(X), nl, 1=2;
see(X, door("sol")), write(X), nl, 1=2;
readchar(_).
```

3. Що бачить Андрій?

```
    see("Andrii", X), write(X), nl, 1=2;
    readchar(_).
```

Результати виконання усіх трьох запитів співпадають із тими, що були у темі Складені домени. Тоді як дещо спрощеними є записи цільових запитів стосовно перших двох прикладів, і значно компактнішим є третій запит.

Наведемо тепер ще один приклад:

- оголосимо домен `item`, який міститиме 3 функтори (з альтернативами): `book(author, title, year, price)`, `transport(type, model, price)`, `realty(type, address, area)`;
- дооголосимо властивості `author`, `title`, `type`, `model`, `address`, `price`, `area`, `year`;
- оголосимо предикат `person`, який міститиме імена усіх «дійових осіб»;
- оголосимо предикат `owner`, який, природньо, може володіти або всіма зразу, або частиною речей (`item`) типу `book`, `transport`, `realty`;
- задамо кілька фактів про `person` та `owner`;
- наведемо кілька прикладів цільових запитів.

```
domains

    item = book(author, title, year, price);
           transport(type, model, price);
           realty(type, address, area)
    author, title, type, model, address = string
    price, area = real
    year = integer

predicates
```

```
    person(string)
    owner(string, item)

clauses

    person("Andrii").
    person("Zakhar").
    owner("Andrii", book("Markov", "Visual Prolog
        7.5", 2016, 625.50)).
    owner("Andrii",     transport("MLRS",     "M142
        HIMARS", 3800000)).
    owner("Zakhar",     realty("bomb     shelter",
        "Underground Str, 1", 322)).
```

Нижче наведено приклади відповідних цільових запитів:

1. Чим володіє Андрій?

```
    Xconst = "Andrii", owner(Xconst, Y),
    write(Xconst, " owns the ", Y), nl, 1=2;
        readchar(_).
```

Відповідь:

```
    Andrii owns the book("Markov","Visual Prolog
        7.5",2016,625.50)
    Andrii    owns    the    transport("MLRS","M142
        HIMARS",3800000)
```

2. Якими книжками володіє Андрій?

```
    Xconst = "Andrii", owner(Xconst, book(_, Y,
        _, _)),
    write(Xconst, " owns the ", Y), nl, 1=2;
        readchar(_).
```

Відповідь:

```
Andrii owns the Visual Prolog 7.5
```

Контрольні запитання

1. Яким чином варто читати дане твердження `rule :- statem1, statem2 and statem3 or statem4, statem5`?

а) `rule` *якщо* `statem1` *і* `statem2` *і* `statem3` *або* `statem4` *і* `statem5`;

б) `rule` *якщо* `statem1` *і не* `statem2` *і* `statem3` *або* `statem4` *і не* `statem5`;

в) `rule` *для* `statem1` *та* `statem2` *і* `statem3` *або* `statem4` *та* `statem5`;

г) `rule` *якщо* `statem1`, `rule` *якщо* `statem2` *і* `statem3` *або* `statem4`, `rule` *якщо* `statem5`;

д) `rule` *для* `statem1` *і* `statem2` *і* `statem3` *або* `statem4` *і* `statem5`.

2. Які алгоритмічні конструкції розглядають у мові Prolog?

а) кон’юктивні;

б) лінійні;

в) обчислення з розгалуженням;

г) універсальні;

д) повторювані обчислення.

3. Яке серед тверджень знаписано некоректно згідно синтаксису Prolog?

а) `rule(X) :- cond(X), statem1; not(cond(X)), statem2.`;

б) `rule(X) :- cond(X), (statem1; not(cond(X))), statem2.`;

в) `rule(X) if cond(X), statem1, not(cond(X)), statem2.`;

г) `rule(X) :- cond(X), statem1; not(cond(X)),`

`statem2, !.;`

д) `rule(X) :- cond(Y), statem1; not(cond(X)), statem2..`

4. Що з переліченого характеризує даний вираз: `rule(X, Y, Z) :- statem1(X, Y), statem2(X, Y), statem3(X, Z), statem4(X, Y, Z), statem5.`?

а) це є твердження-правило;

б) це є цільова мета;

в) дане твердження завжди є хибним;

г) серед логічних операцій тут є зворотна імплікація та диз'юнкція;

д) тут реалізовано, так звані, лінійні обчислення.

5. Яка операція призначена для видалення утворених раніше точок розгалуження?

а) анонімна;

б) імплікації;

в) розриву;

г) відсікання;

д) заперечення.

6. Яких значень набудуть змінні у результаті перевірки тіла твердження: `X=5, Y=X, Y=3`?

а) `X = 5, Y = 5`;

б) `X = 5, Y = 3`;

в) `X = 3, Y = 3`;

г) виникає помилка, але `X` встигає набути значення `3`;

д) виникає помилка і змінні ніяких значень не набувають.

7. Яким чином можна побудувати конструкцію такого виду, що другим аргументом твердження є складний вираз, наприклад: `seeDoor("Zakhar", door("key")).`?

а) використати складені домени;

б) використати зворотну імпілацію;

в) другий аргумент має бути символьного типу;

г) другий аргумент має бути об’єктного типу;

д) такий запис є недопустимим у Prolog.

8. Які можливості забезпечує запис: `item = book(author, title, year, real); transport(type, model, real); estate(type, address, area)`?

а) одна особа одночасно зможе володіти трьома видами речей;

б) для змінної item присвоюється перше твердження (з трьох), яке є істинним;

в) у деякому твердженні на місці предиката `item` зможе бути `book` або `transport`, або `estate`;

г) на початку роботи програми обирається сутність доменного типу `item` (`book` або `transport`, або `estate`) і використовується дана сутність до кінця роботи програми;

д) такий запис є недопустимим у Prolog.

9. До чого призводить застосування операції = між вільною змінною і константою?

а) значення змінної анулюється;

б) порівняння відповідних значень;

в) присвоєння значення цієї змінної;

г) нічого не відбудеться;

д) таке застосування є неприпустимим.

10. Що може означати запис `item = book(string, string, year, price); transport(string, string, price); realty(string, string, area)`?

а) для змінної `item` присвоюється перше твердження (з трьох), яке є істинним;

б) оголошено рядок `item`;

в) оголошено складений домен `item`;

г) оголошено домен з альтернативами `item`;

д) оголошено множина `item`.

Розділ 4. Повторювані обчислення

4.1. Способи управління повторюваними обчисленнями. Приклад відлагодження програми на Prolog

Як вже згадувалось раніше, повторювані обчислення на Prolog реалізуються або механізмом відкату, або рекурсією; операції, схожі на переприсвоєння, організовуються лише механізмами рекурсії. Механізми відкату дозволяють здійснювати більш елементарні речі, хоча у низці випадків є зручнішими та ефективнішими, ніж рекурсія. Тут варто відмітити, що рекурсія та відкат мають різні призначення, що свідчить про важливість вивчення обох цих тем.

Для виконання лабораторних робіт необхідні будуть навички організації рекурсивних правил. Хоча й механізми відкату теж інколи доводитиметься задіювати, наприклад, при реалізації меню.

Почнемо із простішої теми: реалізація повторюваних обчислень механізмами відкату. Взагалі існує 3 традиційні способи управління відкатом: відкат після невдачі (ВПН), відкат із відсіканням (ВВ), повторення визначене програмістом (ПВП). В останньому способі у невеликій мірі фігурують й рекурсивні елементи, але механізми їх роботи, у межах даної теми, глибоко досліджувати не будемо.

I. Механізми управління ВПН вже досить вичерпно згадувались у попередніх розділах. В межах даної теми доцільним є нагадати, що для задіяння ВПН зачасту задається штучно хибна умова (наприклад, `1=2`). Це означає, що мається на меті здійснення перебору всеможливих варіантів розв'язку шляхом підстановки чергового твердження в тіло правила і використання

відкату (завдяки штучно хибній умові `1=2`) у встановлені раніше (уніфікаційними підпрограмами Prolog) точки розгалуження. ВПН у розроблених програмах трапляється досить часто, наприклад, у випадку лінійних обчислень (коли декілька тверджень розміщені через `,`), де також здійснюються відкати в точки розгалуження.

II. Особливість ВВ є наступна: як і для ВПН тут теж виконується спроба здійснити всеможливий перебір, але у тіло відповідного правила має бути добавлена якась додаткова умова, при виконанні якої цей перебір зупиняється. Тобто, наприклад, замість 1000 варіантів розв'язків, які ми могли б вивести на екран (при ВПН), виведеться 327 чи 2, чи 5. Кількість таких розв'язків залежить від того, через скільки кроків (відкатів) виконається наперед задана умова. Послідовність тверджень, яка реалізує ВВ, є наступною:

```
rule :- repeating_statement, stop_cond, !;
   1=1.
```

Варто відзначити, що комбінування кількох способів управління відкатом у комплексі суттєво розширює функціонал програміста.

Розглянемо приклад, який стосується ВПН та ВВ. Нехай маємо певний перелік фактів із:

1) іменами осіб `person(name)`;

2) номерами лабораторних робіт, які виконали ці особи `lab(name, labNum)` (тут передбачено, що хтось із осіб може виконати ту чи іншу лабораторну роботу двома способами).

Потрібно забезпечити виконання наступних цільових запитів по виведенню даних на екран:

1) знайти будь-якого студента із довільним номером виконаної ним лабораторної роботи;

2) показати хто які лабораторні роботи виконав: у форматі `string passed integer laboratory work`;

3) перелічити тих, хто зробив першу і другу лабораторні роботи;

4) показати хто які лабораторні роботи виконав: у форматі `string passed integer, integer, ..., integer, laboratory work`;

5) знайти хто виконав хоч одну лабораторну роботу;

6) перелічити тих, хто здав першу лабораторну роботу (тут виведення кількох однакових відповідей не допускається).

Спершу опишемо розділи `domains`, `predicates` та частину розділу `clauses`, в якому розмістимо перелік фактів (вхідних даних):

```
domains

    name = string
    labNum = integer

predicates

    person(name)
    lab(name, labNum)

clauses

    person("Andrii").
    person("Yurii").
    person("Taras").
    person("Dmitro").
    lab("Andrii", 1).
    lab("Andrii", 2).
    lab("Andrii", 3).
    lab("Andrii", 5).
    lab("Yurii", 1).
```

```
lab("Yurii", 2).
lab("Yurii", 5).
lab("Taras", 2).
lab("Taras", 5).
lab("Andrii", 1).
```

Наведемо тепер правила, які забезпечать виконання поставлених завдань:

```
passed1 :- lab(X, Y), write(X, " passed ", Y,
    " laboratory work"), nl.
passed2 :- lab(X, Y), write(X, " passed ", Y,
    " laboratory work"), nl, 1=2.
passed3 :- lab(X, Y), Y<=2, write(X, " passed
    ", Y, " laboratory work"), nl, 1=2.
passed4 :- person(X), nl, write(X, " passed
    "), lab(X, Y), write(Y, ", "), 1=2.
passed5 :- write("Passed anything: "), nl,
    person(X), _passed5(X), write(X), nl, 1=2.
_passed5(X) :- lab(X, Y), !.
passed6 :- person(X), _passed6(X), write(X, "
    passed 1 laboratory work\n"), 1=2.
_passed6(X) :- lab(X, Y), Y=1, !.
```

Здійснивши почергово запуск програми із цільовими запитами до тверджень `passed1`, `passed2`, `passed3`, `passed4`, `passed5`, `passed6`, отримаємо, відповідно, наступні результати:

1. `Andrii passed 1 laboratory work`
2.
```
Andrii passed 1 laboratory work
Andrii passed 2 laboratory work
Andrii passed 3 laboratory work
Andrii passed 5 laboratory work
Yurii passed 1 laboratory work
Yurii passed 2 laboratory work
Yurii passed 5 laboratory work
```

```
Taras passed 2 laboratory work
Taras passed 5 laboratory work
Andrii passed 1 laboratory work
```

3.
```
Andrii passed 1 laboratory work
Andrii passed 2 laboratory work
Yurii passed 1 laboratory work
Yurii passed 2 laboratory work
Taras passed 2 laboratory work
Andrii passed 1 laboratory work
```

4.
```
Andrii passed 1, 2, 3, 5, 1,
Yurii passed 1, 2, 5,
Taras passed 2, 5,
Dmitro passed
```

5.
```
Passed anything:
Andrii
Yurii
Taras
```

6.
```
Andrii passed 1 laboratory work
Yurii passed 1 laboratory work
```

Як бачимо:

1. За допомогою правила `passed1`, очікувано, виведено у форматованому вигляді перший із перелічених фактів про `lab` у розділі `clauses`.

2. Додавання до правила `passed1` штучно хибної умови `1=2` (див. правило `passed2`) призвело до виведення усіх фактів про `lab`. Це пов'язано із задіянням механізму ВПН. Тут є очевидний недолік: надто часто повторюється ім'я тієї чи іншої особи. Така надлишковість на практиці може бути навіть шкідливою, наприклад, при схожості імен кількох осіб. Варіанти вирішення подібного роду проблем запропоновані у прикладах 4–6.

3. Проста заміна умови `1=2` на `Y<=2` у правилі `passed2` при наведених вхідних даних призведе до того, що виведеться лише перший факт про `lab` і програма успішно

(Yes) завершить своє виконання. Застосування лише ВВ теж не дозволить досягнути бажаного результату, оскільки нам потрібно здійснити усеможливий перебір (1=2); виводити ж необхідно лише те, що задовольняє умові Y<=2. Тому залишається єдиний варіант: якимось чином поєднати умови 1=2 та Y<=2 механізмом ВПН. У запропонованій вище реалізації правила passed3 передбачається відкат (до lab(X, Y)) як при Y>2, так і при 1=2; при Y<=2 відбувається ще й виведення результатів на екран.

Приклад реалізації правила passed3 є зручним для демонстрації можливостей механізму відлагодження (Debug), про який вже згадувалось у розділі 1. Нижче наведено детальний «розбір» послідовності, у якій уніфікаційні підпрограми Prolog забезпечують перебір тверджень для досягнення результату.

В першу чергу, для зручності, усі важливі для відлагодження твердження у тілах правил перенесемо у різні рядки:

```
passed3 :-
 lab(X, Y),
 Y<=2,
 write(X, " passed ", Y, " laboratory work"),
 nl,
 1=2.
```

Запускаємо програму у режимі Debug, наприклад, комбінацією клавіш Ctrl+Shift+F9. Далі здійснюємо покрокове відлагодження за допомогою клавіші F7. У даному процесі в твердженні passed3 виконується наступне:

- для твердження lab(X, Y) створюється точка розгалуження;

- відбувається перехід до факту `lab("Andrii", 1)`;
- оскільки параметри факту `lab("Andrii", 1)` не суперечать параметрам твердження `lab(X, Y)`, то продовжується перевірка твердження `passed3` із вже набутими значеннями змінних (`X="Andrii"`, `Y=1`);
- перевіряється умова `Y<=2` на істинність;
- оскільки `1<=2`, то виводиться перший рядок результатів;
- перевіряється умова `1=2`;
- дана умова є хибною, тому відбувається перехід до останньої (і, у нашому випадку, єдиної) точки розгалуження; змінні `X` та `Y`, при цьому, звільняються від набутих значень;
- оскільки у випадку факту `lab("Andrii", 1)` правило `passed3` набуло хибного значення, то уніфікаційні підпрограми Prolog обирають наступний факт `lab("Andrii", 2)` (припустивши, що при ньому правило `passed3` набуде значення `Yes`);
- знову ж послідовно відбувається перехід до твердження `1=2` і повернення до останньої точки розгалуження;
- на даному етапі обирається факт `lab("Andrii", 3)`, через що у тілі правила `passed3` відповідні вільні змінні набувають значень: `X="Andrii"`, `Y=3`;
- оскільки `3<=2`, то відбувається повернення до останньої точки розгалуження;
- такий процес повторюється до тих пір, поки не вичерпаються усі наявні факти про `lab(X, Y)`.

Варто зауважити, що у наявному вигляді правило `passed3` завершиться із остаточно хибним значенням. Це потрібно коректно враховувати у розділі `goal`.

4. Для виконання цільового запиту №4 необхідно здійснити всеможливий перебір осіб, до кожної з яких застосувати свій всеможливий перебір виконаних лабораторних робіт. Такий процес нагадує вкладені цикли у C++. У розділі 3 проводились аналогії між вкладеними циклами `foreach` та лінійними обчисленнями. Відповідно до цього, наведене вище правило для виконання цільового запиту №4 (`passed4`) можна переписати й у такому, еквівалентному, вигляді:

```
passed4 :- statem1(X), statem2(X), 1=2.
statem1(X) :- person(X), write(X, "\n passed
    ").
statem2(X) :- lab(X, Y), write(Y, ", ").
```

Тут точки розгалуження розміщуються і в позиції твердження `statem1(X)`, і в позиції твердження `statem2(X)`.

Варто зауважити, що правило `passed4` при даному наборі фактів приводить до виведення лишнього символа і лишнього рядка. Виправити це можливо, наприклад, введенням додаткових тверджень (через знак кон'юнкції); із застосуванням ВВ.

5. Якщо в останньому наведеному прикладі поставити знак відсікання `!`, наприклад, у кінці твердження `statem1(X)`, то після виконання правила `write` щодо першої із задекларованих осіб видалиться точка розгалуження у позиції `person(X)`. У результаті виведеться лише перша з осіб та усі лабораторні роботи, які вона здала. Розміщення відсікання вкінці `statem2` призведе до того, що по кожній з осіб буде виведено максимум по одному результату.

Виходячи із подібних міркувань побудоване твердження `passed5`. Тут знак відсікання `!` (пам'ятаємо,

що відсікання видаляє усі створені раніше точки розгалуження лише всередині конкретного твердження) розміщений таким чином, щоб при першому ж знайденому факті `lab(X, Y)` видалялись усі точки розгалуження правила `lab`, і твердження `_passed5` набувало остаточно істинного значення. А це й є єдиною «перепусткою» до твердження `write(X)`. Тоді як умова `1=2` вимагає від уніфікаційних підпрограм Prolog повернутись до точки розгалуження в позиції `person(X)`. Таким чином й реалізовується взаємодія ВПН та ВВ.

6. Правило `passed6` побудоване із аналогічних до `passed5` міркувань.

III. Наступний спосіб управління відкатом є ПВП. За допомогою нього можна, наприклад, створювати меню. Принципи роботи ПВП у Prolog та циклу `do…while` в C++ є дещо схожими. Для організації ПВП потрібно запрограмувати наступне:

- створити предикат, наприклад, із назвою `cycle` (насправді назва цього предикату може бути довільною);
- створити 2 твердження виду:

```
cycle.
cycle :- cycle.
```

- тіло іншого твердження, в якому маємо бажання організувати ПВП, сформувати у вигляді виду:

```
rule :- cycle, repeating_statement, stop_cond.
```

Виклик правила `rule` із розділу цільової мети призведе до:

- перевірки факту `cycle` (із попереднім встановленням тут точки розгалуження);
- виконання `repeating_statement`;

- перевірки умови зупинки повторюваних обчислень `stop_cond`;
- якщо `stop_cond` завершується із значенням `No`, то:
  - по причині наявності точки розгалуження розпочинається перевірка правила `cycle`; в ньому ж створюється власна точка розгалуження;
  - правило `cycle` перевіряє факт `cycle` і повертає у правило `rule` значення `Yes`;
  - при наступному випадку хибності умови `stop_cond` твердження `cycle` перевіряє (запускає) саме ж себе, що одночасно знищує стару і створює нову точку розгалуження;
  - правило `cycle` знову перевіряє факт `cycle` і повертає у правило `rule` значення `Yes`;
  - якщо `stop_cond` завершується із значенням `Yes`, то правило `rule` повертає остаточно істинне значення `Yes`.

На перший погляд може здатись, що після перевірки умови `stop_cond` знадобиться виконання операції відсікання `!` для видалення раніше створеної точки розгалуження, але це не є необхідним, оскільки ця точка розгалуження не знаходиться в самому правилі `rule`, а є всередині правила `cycle`. Таким чином, уніфікаційні підпрограми Prolog видалять наявну точку розгалуження одночасно із успішним завершенням перевірки правила `rule`.

Розглянемо приклад побудови меню. Тут при введенні з клавіатури цифри 1 має вивестись повідомлення `fulfilled statement:1`; при введенні цифри 2 – `fulfilled statement:2`; при введенні 0 – `exit`.

```
predicates

     dowhile
     menu
```

```
    _menu(integer)
    statem1
    statem2

clauses

    dowhile.
    dowhile :- dowhile.
    menu :- dowhile,
        write("Choose an option: \n"),
        write("1 - statement 1 \n"),
        write("2 - statement 2 \n"),
        write("0 - exit \n"),
        readint(X),
        _menu(X).
    _menu(X) :- X=1, statem1, 1=2;
                X=2, statem2, 1=2;
                X=0, 1=1.
    statem1 :- write("fulfilled statement:1"), nl.
    statem2 :- write("fulfilled statement:2"), nl.

goal

    menu.
```

4.2. Рекурсія

Рекурсія – це, подібно до інших мов програмування, така програмна конструкція, при якій у тілі певного твердження викликається це ж саме твердження (правильніше сказати: у тілі певного твердження викликається твердження з тим самим іменем). Досить специфічний вид рекурсії має місце при ПВП, де рекурсивний виклик відбувається завдяки відкату і послідовно оголошених факту та правила (з однаковими ідентифікаторами). Традиційно рекурсію організовують таким чином, щоб спочатку оголошувалось правило, а тоді вже факт (у ПВП відбувається навпаки).

Оскільки рекурсія є загалом складною для розуміння програмною конструкцією, то почнемо із простого прикладу, який, з одного боку, розкриває сутність рекурсії, а з іншого – приводить до обґрунтування її використання. Нехай за допомогою трьох почергово викликаних тверджень необхідно вивести 3 повідомлення виду:

```
hello 3 times
hello 2 times
hello 1 times
```

Відповідна програмна реалізація може мати вигляд:

```
predicates

    hello(integer)
    hello2(integer)
    hello3(integer)

clauses

    hello(X) :- write("hello ", X, " times\n"),
        PrevX = X - 1, hello2(PrevX).
    hello2(X) :- write("hello ", X, " times\n"),
        PrevX = X - 1, hello3(PrevX).
    hello3(X) :- write("hello ", X, " times\n").

goal

    hello(3), readchar(_); 1=1.
```

Як бачимо, тут кожне наступне твердження виконується (перевіряється) при все меншому X і за однаковим шаблоном. Лише останнє (заключне) твердження відрізняється від усіх попередніх. Використавши

відлагоджувач, можна помітити, що послідовність перевірки тверджень відбувається у наступному порядку:

- розпочинається перевірка твердження `hello(3)`;
- твердження `write("hello ", 3, " times\n")` завершується з істинним результатом;
- змінна `PrevX` набуває значення `X – 1`, тобто `2`;
- розпочинається перевірка твердження `hello2(2)` та одночасно завершується перевірка твердження `hello(3)`;
- твердження `write("hello ", 2, " times\n")` завершується із істинним результатом;
- нова змінна `PrevX` набуває значення `X – 1`, тобто `1`;
- розпочинається перевірка твердження `hello3(1)` та одночасно завершується перевірка твердження `hello2(2)`;
- твердження `write("hello ", 1, " times\n")` і `hello3(1)` завершуються із істинним результатом.

Очікувалося б, що після виведення на екран усіх вітань, мали б послідовно успішно завершуватись (повертати остаточно істинний результат) твердження `hello3(1)`, `hello2(2)` та `hello1(3)`. Але уніфікаційні підпрограми Prolog, проаналізувавши наперед, що на зворотному ході рекурсії ніяких тверджень перевіряти не доведеться, заздалегідь видаляють зі стеку «старе» твердження і тоді ж поміщають туди нове твердження.

Суттєвими недоліками наведеної вище програмної реалізації є наступні:

- якщо необхідно вивести просте повідомлення типу `hello` велику кількість разів: наприклад, 100 чи 1000, чи 1000000, то відповідна програма повинна містити понад 100 чи 1000, чи 1000000 рядків коду;
- відповідний файл із вихідним текстом займатиме надто великий розмір;

- неможливо наперед (на стадії розробки програмного забезпечення) передбачити скільки разів потрібно привітатися.

Інструментарієм для вирішення усіх перелічених проблем володіє рекурсія. Вона є основним методом програмування на Prolog та надає можливості для побудови одночасно компантних і ефективних програм. Це дозволяє у наведеному вище прикладі програмної реалізації усі твердження з однаковою структурою замінити на єдине. Тобто, наприклад, у рядку

```
hello(X) :- write("hello ", X, " times\n"),
    PrevX = X - 1, hello2(PrevX).
```

твердження `hello2(PrevX)` доцільно замінити на `hello(PrevX)`. Таким чином, перевірка `hello(PrevX)` у тілі `hello(X)` призведе до виклику правила із тією самою структурою; цього ми й добивались. У заключному правилі `hello3(X)` теж приберемо нумерацію та змінимо текст для виведення. Проте реалізація розділу `clauses`, навіть у такому вигляді

```
hello(X) :- write("hello ", X, " times\n"),
    PrevX = X - 1, hello(PrevX).
hello(X) :- write("all greetings are displayed
    above").
```

є невірною, оскільки матиме місце нескінченна кількість рекурсивних викликів:

```
hello 3 times
hello 2 times
hello 1 times
hello 0 times
hello -1 times
```

```
hello -2 times
hello -3 times
…
```

У програмі потрібно вказати якусь ще додаткову умову, яка завершувала би процес перевірки нових тверджень. У даному випадку, очевидно, доречно обрати умову X>0. Тобто остаточно програмна реалізація може мати вигляд:

```
predicates

    hello(integer)

clauses

    hello(X) :- X > 0, write("hello ", X, "
        times\n"), PrevX = X - 1, hello(PrevX).
    hello(X) :- write("all greetings are displayed
        above").

goal

    hello(3), readchar(_); 1=1.
```

Результати будуть наступними:

```
hello 3 times
hello 2 times
hello 1 times
all greetings are displayed above
```

Як бачимо, основна повторювана дія (вітання) відбувається ще до виклику рекурсивного правила. Такий вид рекурсії називається повторюванням на прямому ході рекурсії (ПХР).

Наведемо також простий приклад (без рекурсії), при якому основна повторювана дія виконується на зворотному ході. Для цього, у даному випадку, достатньо твердження `write` перемістити у позицію, яка знаходиться після тверджень виду `hello/*number*/(PrevX)`.

```
predicates

     hello(integer)
     hello2(integer)
     hello3(integer)

clauses

     hello(X) :- PrevX = X - 1, hello2(PrevX),
         write("hello ", X, " times\n").
     hello2(X) :- PrevX = X - 1, hello3(PrevX),
         write("hello ", X, " times\n").
     hello3(X) :- write("hello ", X, " times\n").

goal

     hello(3), readchar(_); 1=1.
```

Результати будуть наступними:

```
hello 1 times
hello 2 times
hello 3 times
```

Як бачимо, вітання відбуваються у зворотному порядку (на відміну від прямого ходу рекурсії). Більше того, у даному випадку, таке виведення є більш природним з людської точки зору.

Перепишемо даний приклад із використанням рекурсії. Виконавши аналогічні до попередніх випадків дії, отримаємо:

```
predicates

     hello(integer)

clauses

     hello(X)  :-  X  >  0,  PrevX  =  X  -  1,
         hello(PrevX),  write("hello  ",  X,  "
         times\n").
     hello(X) :- write("start\n").

goal

     hello(3), readchar(_); 1=1.
```

Тут виведеться наступне:

```
start
hello 1 times
hello 2 times
hello 3 times
```

Такий вид рекурсії (при якому основна повторювана дія відбувається після рекурсивного виклику правила) називається повторенням на зворотному ході рекурсії (ЗХР).

Застосувавши механізм відлагодження можна помітити, що повторення на зворотному ході рекурсії призводить до того, що утримуються всі рекурсивно викликані твердження у стеку аж до завершення роботи програми, що є невигідним з точки зору використання пам’яті.

Таким чином, на прямому ході рекурсії відбуваються дії по поступовому накопиченню значень, а на зворотному ході – по, у певному сенсі, поступовому «відкату» до початкових значень.

Виходячи із наведених вище прикладів стає зрозумілим, що оголошення рекурсивних правил відбуваються за шаблоном:

```
recursive_rule :-
    recursion_continuation_condition[,
    forward_recursion_statement],
    recursive_rule[,
    backward_recursion_statement].
recursive_rule :- recursion_termination_cond.
```

або, що є аналогічним, із використанням всього одного рекурсивного правила, розділеного диз'юнкцією:

```
recursive_rule :-
    recursion_continuation_condition[,
    forward_recursion_statement],
    recursive_rule[,
    backward_recursion_statement];
    recursion_termination_cond.
```

Як видно, рекурсивне правило може не мати твердження прямого ходу рекурсії або твердження зворотного ходу рекурсії, або мати обидва, або не мати обох (хоча цей варіант є безсенсовим). Окрім того, аргументами твердження не можуть бути вирази із математичними операціями у якості аргументів. Тобто запис виду `hello(X - 1)` призведе до помилки на етапі компіляції.

Таким чином, застосування рекурсивних правил дає можливість здійснення розрахунків із накопиченням

значень (що є альтернативою доступного в інших мовах перепрісвоєння).

Розглянемо ще низку прикладів організації рекурсивних правил для розв'язання суттєво складніших завдань. Нехай необхідним є обчислення факторіалу деякого числа. Ми знаємо, що $n!=1\cdot 2\cdot 3\cdot ...\cdot (n-1)\cdot n.$ Для набуття практичних навичок нижче наведено програмний код із чотирма варіантами побудови рекурсивних правил обчислення факторіалу. Тут:

а) предикат `fl_fr_incr` призначений для обчислення факторіалу деякого числа із використанням ПХР та у прямому порядку множення $(1\cdot 2\cdot 3\cdot ...\cdot (n-1)\cdot n)$;

б) предикат `fl_fr_decr` призначений для обчислення факторіалу деякого числа із використанням ПХР та у зворотному порядку множення $(n\cdot (n-1)\cdot ...\cdot 3\cdot 2\cdot 1)$;

в) предикат `fl_br_decr` призначений для обчислення факторіалу деякого числа із використанням ЗХР та у зворотному порядку множення $(n\cdot (n-1)\cdot ...\cdot 3\cdot 2\cdot 1)$;

г) предикат `fl_br_incr` призначений для обчислення факторіалу деякого числа із використанням ЗХР та у прямому порядку множення $(1\cdot 2\cdot 3\cdot ...\cdot (n-1)\cdot n)$.

```
domains

    int = integer

predicates

    nondeterm fl_fr_incr(int, int, int, int)%а
    nondeterm fl_fr_decr(int, int, int)%б
    nondeterm fl_br_decr(int, int, int)%в
    nondeterm fl_br_incr(int, int)%г

clauses
```

```
fl_fr_incr(I, N, TmpF, F) :- I<N,
    NextI = I + 1, TmpTmpF = I * TmpF,
    fl_fr_incr(NextI, N, TmpTmpF, F).
fl_fr_incr(N, N, TmpF, F) :- F = TmpF * N.

fl_fr_decr(N, TmpF, F) :- N>0,
    PrevN = N - 1, TmpTmpF = N * TmpF,
    fl_fr_decr(PrevN, TmpTmpF, F).
fl_fr_decr(0, F, F).

fl_br_decr(I, N, F) :- I<=N, NextI = I + 1,
    fl_br_decr(NextI, N, TmpF), F = I * TmpF.
fl_br_decr(I, N, 1).

fl_br_incr(N, F) :- N>0, PrevN = N - 1,
    fl_br_incr(PrevN, TmpF), F = N * TmpF.
fl_br_incr(0, 1).
```

Почнемо із випадку (а). Тут використано:

- змінну `I`, в яку послідовно поміщаються числа 1, 2, 3, ..., n;
- змінну `N`, яка рівна числу n (без задання цієї змінної доведеться вручну вказувати яким повинен бути останній множник);
- змінну `TmpF`, в якій рекурсивно накопичується факторіал числа;
- змінну `F`, в якій міститиметься шукане $n!$; дана змінна необхідна, оскільки решта змінних ще у момент першої перевірки правила будуть у статусі змінних зі значеннями; нам же знадобиться вільна змінна `F` для повернення знайденого $n!$.

Відповідний розділ `goal` для випадку (а) можна записати у вигляді:

```
N=3, fl_fr_incr(1, N, 1, F), write(N, "!=",
    F), readchar(_); 1=1. % 3!=6
```

Тут, в процесі перевірки твердження `fl_fr_incr` у вигляді `fl_fr_incr(1, 3, 1, F)`, відбувається накопичення добудку чисел від `1` (перший аргумент) до `3` (другий аргумент) за допомогою третього аргумента, який попередньо ініціалізується одиницею; результат записується у вільну змінну `F`. Саме рекурсивне правило `fl_fr_incr`, при даному варіанті вхідних даних, викликає саме себе 2 рази. Увесь хід розв'язання представлено у табл. 4.1.

Таблиця 4.1

Хід розв'язання `fl_fr_incr`

викликів	**1**-й аргумент		**2**-й аргумент		**3**-й аргумент		**4**-й аргумент	
			Дані					
0	1		3		1		вільна змінна	
			Перше правило					
–	*до*	*після*	*до*	*після*	*до*	*після*	*до*	*після*
1	1	2	3	без змін	1	1	вільна змінна	без змін
2	2	3	3	без змін	1	2	вільна змінна	без змін
3	3	без змін	3	без змін	2	без змін	вільна змінна	без змін
			Друге правило					
–	*до*	*після*	*до*	*після*	*до*	*після*	*до*	*після*
3	3	без змін	3	без змін	2	без змін	вільна змінна	6
			Результат					
–	*до*	*після*	*до*	*після*	*до*	*після*	*до*	*після*
0	1	без змін	3	без змін	1	без змін	вільна змінна	6

Варто зазначити, що, подібно до наведеного раніше прикладу із вітаннями, твердження `fl_fr_incr` у випадку `N=3` можна переписати у вигляді:

```
fl_fr_incr1(I, N, TmpF, F) :-
    TmpTmpF = TmpF * I, NextI = I + 1,
    fl_fr_incr2(NextI, N, TmpTmpF, F).
fl_fr_incr2(I, N, TmpF, F) :-
    TmpTmpF = TmpF * I, NextI = I + 1,
```

```
    fl_fr_incr3(NextI, N, TmpTmpF, F).
fl_fr_incr3(N, N, TmpF, F) :- F = TmpF * N.
```

У випадку (б) процес множення відбувається в зворотному порядку, що є менш звичним. З іншого боку, предикат `fl_fr_decr`, у порівнянні з `fl_fr_incr`, потребує меншої кількості властивостей. Це стає можливим за рахунок заміни умови `I<N` на `N>0`, через що перший аргумент `I` більше не є потрібним. Загальний хід розв'язання у випадку цільової мети

```
N=3, fl_fr_decr(N, 1, F), write(N, "!=", F),
    readchar(_); 1=1. % 3!=6
```

є аналогічним до випадку твердження `fl_fr_incr` (див. табл. 4.2).

Таблиця 4.2

Хід розв'язання `fl_fr_decr`

викликів	**1**-й аргумент		**2**-й аргумент		**3**-й аргумент	
			Дані			
0	3		1		вільна змінна	
	Перше правило					
–	*до*	*після*	*до*	*після*	*до*	*після*
1	3	2	1	3	вільна змінна	без змін
2	2	1	3	6	вільна змінна	без змін
3	1	0	6	6	вільна змінна	без змін
4	0	без змін	6	без змін	вільна змінна	без змін
	Друге правило					
–	*до*	*після*	*до*	*після*	*до*	*після*
4	0	без змін	6	без змін	вільна змінна	6
			Результат			
–	*до*	*після*	*до*	*після*	*до*	*після*
0	3	без змін	1	без змін	вільна змінна	6

Обчислення факторіалу на ЗХР у випадку (в), в порівнянні з ПХР (а), не потребує використання аргумента `FF`. Таким чином, матимемо наступні аргументи:

- змінну `I`, в яку послідовно поміщаються числа 1, 2, 3, ..., n;
- змінну `N`, яка рівна числу n;
- змінну `F`, в якій рекурсивно накопичується факторіал числа аж до `F`$=n!$.

Відповідний розділ `goal` для випадку (в) можна записати так:

```
N=3, fl_br_decr(1, N, F), write(N, "!=", F),
    readchar(_); 1=1. % 3!=6
```

Тут, в процесі перевірки твердження `fl_br_decr` у вигляді `fl_br_decr(1, N, F)`, відбувається накопичення добудку чисел від `1` (перший аргумент) до `3` (другий аргумент), але у зворотному порядку; результат записується у вільну змінну `F`. Загалом рекурсивне правило `fl_br_decr` викликає саме себе 3 рази. Увесь хід розв'язання представлено у табл. 4.3.

Таблиця 4.3

Хід розв'язання `fl_br_decr`

викликів	**1**-й аргумент		**2**-й аргумент		**3**-й аргумент	
			Дані			
0	1		3		вільна змінна	
			Перше правило			
–	*до*	*після*	*до*	*після*	*до*	*після*
1	1	2	3	без змін	вільна змінна	без змін
2	2	3	3	без змін	вільна змінна	без змін
3	3	4	3	без змін	вільна змінна	без змін
4	4	без змін	3	без змін	вільна змінна	без змін
			Друге правило			
–	*до*	*після*	*до*	*після*	*до*	*після*

продовження табл. 4.3

4	4	без змін	3	без змін	вільна змінна	1
			Перше правило			
–	*до*	*після*	*до*	*після*	*до*	*після*
3	3	без змін	3	без змін	1	3
2	2	без змін	3	без змін	3	6
1	1	без змін	3	без змін	6	6
			Результат			
–	*до*	*після*	*до*	*після*	*до*	*після*
0	1	без змін	3	без змін	вільна змінна	6

У випадку (г) процес множення відбувається у прямому порядку; предикат `fl_fr_incr`, анологічно до (б), потребує меншої кількості властивостей. Загальний хід розв'язання у випадку цільової мети

```
N=3, fl_br_incr(N, F), write(N, "!=", F),
    readchar(_); 1=1. % 3!=6
```

наведено у табл. 4.4, що є аналогічним до випадку твердження `fl_br_decr` (див. табл. 4.3).

Таблиця 4.4

Хід розв'язання `fl_br_incr`

викликів	**1**-й аргумент		**2**-й аргумент	
	Дані			
0	3		вільна змінна	
	Перше правило			
–	*до*	*після*	*до*	*після*
1	3	2	вільна змінна	без змін
2	2	1	вільна змінна	без змін
3	1	0	вільна змінна	без змін
4	0	без змін	вільна змінна	без змін
	Друге правило			
–	*до*	*після*	*до*	*після*
4	0	без змін	вільна змінна	1

продовження табл. 4.4

	Перше правило			
–	*до*	*після*	*до*	*після*
3	1	без змін	1	1
2	2	без змін	1	2
1	3	без змін	2	6
	Результат			
–	*до*	*після*	*до*	*після*
0	3	без змін	вільна змінна	6

Цікавим випадком рекурсії є такий, який передбачає реалізацію кількох рекурсивних правил з однаковим ідентифікатором твердження. Один із найочевидніших прикладів полягає в обчисленні виразу виду:

$$y = \sum_{i=1}^{n} \begin{cases} x^2, \ якщо \ i - парне, \\ x^3, \ якщо \ i - непарне. \end{cases}$$

Відповідний програмний код реалізуємо як для ПХР, так і для ЗХР.

```
predicates

      nondeterm sum_pow_fr(int, int, real, real)
      nondeterm sum_pow_br(int, int, real)

clauses

      sum_pow_fr(I, N, TmpSum, Sum) :-
          I<=N, I/2<>round(I/2),
          TmpTmpSum = TmpSum + I * I, NextI = I +
          1, sum_pow_fr(NextI, N, TmpTmpSum, Sum).
      sum_pow_fr(I, N, TmpSum, Sum) :-
          I<=N,
```

```
        TmpTmpSum = TmpSum + I * I * I, NextI = I
        + 1, sum_pow_fr(NextI, N, TmpTmpSum,Sum).
    sum_pow_fr(I, N, S, S).


    sum_pow_br(I, N, Sum) :-
        I<=N, I/2<>round(I/2),
        NextI = I + 1,
        sum_pow_br(NextI, N, TmpSum),
        Sum = TmpSum + I * I.
    sum_pow_br(I, N, Sum) :-
        I<=N,
        NextI = I + 1,
        sum_pow_br(NextI, N, TmpSum),
        Sum = TmpSum + I * I * I.
    sum_pow_br(I, N, 0).

goal

    sum_pow_fr(1,   3,   0,   Sum),   write(Sum),
        readchar(_); 1=1.%18
    sum_pow_br(1,     3,     Sum),     write(Sum),
        readchar(_); 1=1.%18
```

Як бачимо, у точку зупинки рекурсії можна потрапити лише у момент, коли перший аргумент стає більшим за другий. В інакшому випадку почергово перевіряється то перше твердження, то друге, залежно від виконання умови `A/2<>round(A/2)`. Обидва ці твердження різняться між собою лише наявністю чи відсутністю умови `A/2<>round(A/2)` та відповідним додаванням або `TmpSum + I * I`, або `TmpSum + I * I * I`.

Реалізація ще одного виду рекурсивного правила, яка дозволяє здійснювати розрахунки до досягнення певної умови, можна реалізувати для випадку обчислення

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^k}{k!} + \ldots.$$

Нехай умовою завершення обчислень є $\left| x^k / k! \right| < \varepsilon$. Тоді відповідний код може мати вигляд:

```
predicates

      nondeterm exp_fr(int, real, real, real, real,
          real)
      nondeterm exp_br(int, real, real, real, real)

clauses

      exp_fr(I, Eps, XdivF, X, TmpSum, Sum) :-
          abs(XdivF)>Eps,
          NextI = I + 1, TmpXdivF = XdivF * X / I,
          TmpTmpSum = TmpSum + XdivF,
          exp_fr(NextI, Eps, TmpXdivF, X, TmpTmpSum,
          Sum).
      exp_fr(_, _, _, _, TmpSum, TmpSum).

      exp_br(I, Eps, XdivF, X, Sum) :-
          abs(XdivF)>Eps, !,
          NextI = I + 1, TmpXdivF = XdivF * X / I,
          exp_br(NextI, Eps, TmpXdivF, X, TmpSum),
          Sum = TmpSum + XdivF.
      exp_br(_, _, _, _, 0).
```

Параметри тверджень `exp_fr(I, Eps, XdivF, X, TmpSum, Sum)` і `exp_br(I, Eps, XdivF, X, Sum)` відповідають за наступне:

а) `I` – за знаходження чергового множника в факторіалі;
б) `Eps` – параметр точності розрахунків;
в) `XdivF` – значення чергового доданка розкладу ряду;

г) X – степінь виразу e^x;

д) TmpSum у твердженні exp_fr використовується для накопичування суми ряду;

е) Sum у твердженні exp_br використовується для накопичування суми ряду, а в exp_fr – для копіювання результату з TmpSum.

У результаті перевірки тверджень exp_fr та exp_br при Eps=0.001 та X=0.5 у вигляді

```
exp_fr(1, 0.001, 1, 0.5, 0, Sum), write(Sum),
    readchar(_); 1=1.
exp_br(1, 0.001, 1, 0.5, Sum), write(Sum),
    readchar(_); 1=1.
```

в обох випадках матимемо однаковий розв'язок:

```
1.6484375
```

Контрольні запитання

1. Які є способи реалізації повторюваних обчислень у Prolog?

а) з використанням циклу for;

б) з використанням циклу foreach;

в) з використанням циклу do..while;

г) з використанням рекурсії;

д) з використанням механізму відкату.

2. Що означає абревіатура ПВП у Prolog?

а) player vs player;

б) повторення визначене програмістом;

в) порядок виконання програми;

г) послідовність виконання коду;

д) пауза в повторенні.

3. У яких випадках ВВ є кращим за ВПН?

а) якщо потрібно здійснити повний перебір фактів та

правил;

б) якщо не потрібно здійснювати повний перебір фактів та правил;

в) якщо потрібно згрупувати деякі проміжні результати;

г) при бажанні запобігти групуванню проміжних результатів;

д) таких випадків не існує.

4. У наведеному раніше прикладі, який механізм дозволяє здійснити групування виду: `Andrii passed 1, 2, 3, 5, 1,`?

а) ВПН;

б) ВВ;

в) ПВП;

г) ПК;

д) ПУП.

5. Для чого з переліченого використовують запис виду: `statement. statement :- statement.`?

а) ВПН;

б) ВВ;

в) ПВП;

г) ПК;

д) ПУП.

6. Яка різниця між ПВП та рекурсією?

а) у ПВП не передбачено використання тверджень-фактів;

б) для рекурсії не передбачено використання тверджень-фактів;

в) у ПВП твердження-факт знаходиться перед твердженням-правилом;

г) для рекурсії твердження-факт знаходиться перед твердженням-правилом;

д) немає різниці.

7. Чому краще використовувати рекурсію, ніж

багаторазово перевіряти одноманітні твердження?

а) код програми займатиме суттєво менше дискового простору;

б) рекурсія є інтуїтивно зрозумілішою конструкцією, ніж звичайна перевірка тверджень;

в) код виконуватиметься швидше;

г) код буде значно компактнішим;

д) принципової різниці немає, бо все залежить від стилю («почерку») програміста.

8. Навіщо потрібна точка зупинки рекурсії?

а) без неї відбуватимуться рекурсивні виклики аж до вичерпання пам'яті стеку;

б) вона призначена для задання початкової конфігурації рекурсивного процесу;

в) бо інакше рекурсія перетвориться на цикл;

г) бо потрібно якимось чином лише одноразово застосувати операцію відсікання;

д) щоб задати початкові значення певним змінним.

9. Чим характеризується дане правило `hello(X) :- X > 0, write("hello ", X, " times\n"), X1 = X - 1, hello(X1).`?

а) воно є рекурсивним;

б) повторення відбуваються на ПХР;

в) повторення відбуваються на ЗХР;

г) виведення результату відбувається у порядку зростання `X`;

д) виведення результату відбувається у порядку спадання `X`.

10. Чим характеризується дане правило `hello(X) :- X > 0, X1 = X - 1, hello(X1), write("hello ", X, " times\n").`?

а) воно є рекурсивним;

б) повторення відбуваються на ПХР;

в) повторення відбуваються на ЗХР;
г) виведення результату відбувається у порядку зростання X;
д) виведення результату відбувається у порядку спадання X.

Розділ 5. Списки

5.1. Синтаксис роботи зі списками

У мові Prolog список – це впорядкована послідовність елементів певного конкретного типу, поміщена у квадратні дужки []. Особливістю роботи із послідовностями елементів в Prolog є те, що тут не передбачений доступ до конкретного елемента за допомогою індексу. Натомість робота зі списками відбувається за допомогою рекурсії: формування списку, виведення елементів списку, видалення елементів зі списку тощо.

Оголошення списку відбувається в розділі domain за наступним синтаксисом:

списковий_домен = домен*

Нижче наведено декілька прикладів оголошення спискового домену:

```
lint = integer* % представляє список, який складається з цілочисельних елементів
lreal = real* % представляє список, який складається з дробових елементів
lsymbol = symbol* % представляє список, який складається з символічних імен
llint = lint* % представляє список, який складається зі списків цілочисельних елементів
litem = item* % може представляти список складених доменів чи доменів з альтернативами
```

Наведемо тепер декілька прикладів наповнення списків:

```
X = [1, 2, 3] % список цілих чисел
X = [3.14, 0.83, 2] % список дійсних чисел
```

```
X = [[1], [1, 2], [1, 2, 3]] % список списків цілих чисел
X = [janna, orange, eat] % список символічних імен
X = [book("Markov", "Visual Prolog 7.5", 2016, 625.50), transport(avto, "zaz", 1000000)] % список екземплярів деякого домена з альтернативами
X = [] % порожній список (окремий вид важливих списків)
X = [[], []] % список порожніх списків
X = [[[[]]]] % список списків списків, в якому міститься порожній список
X = [[], [[]]] % список, який складається з двох елементів різних типів, а саме: список і список списків; згідно визначення такий запис є некоректним
```

Запис же для виведення елементів списку у розділі `goal` можна сформувати, наприклад, у вигляді:

```
X = [1, 2, 3, 4, 5], write(X), nl, 1=2, 1=1.
```

Вище наведено найпростіші випадки наповнення та виведення елементів списку. Запишемо на мові Prolog простий приклад програми, приймаючи, що розділ `domain` вже попередньо описаний.

```
predicates

    nondeterm list1(lint)
    nondeterm list2(llint)

clauses

    list1([1, 2, 3, 4, 5]). % або, що те саме, list1(X) :- X = [1, 2, 3, 4, 5].
    list1([5, 4, 3, 2]).
    list1([]).
    list2([[1, 2, 3], [3, 2, 1]]).
```

```
list2([[], [1], [1, 2]]).
```

Наведемо декілька прикладів цільових запитів до `list1` і `list2`.

1. Які є списки цілих?

```
list1(X), write(X), nl, 1=2; 1=1.
```

Результат:

```
[1,2,3,4,5]
[5,4,3,2]
[]
```

2. Які є списки списків цілих з трьох елементів?

```
list2(X), write(X), nl, 1=2; 1=1.
```

Результат:

```
[[1,2,3],[3,2,1]]
[[],[1],[1,2]]
```

3. Знайти третій елемент у списку цілих з 5-ти елементів

```
list1([_,_,X,_,_]), write(X), nl, 1=2; 1=1.
```

Результат:

```
[[1,2,3],[3,2,1]]
[[],[1],[1,2]]
```

Як вже було сказано, це був лише простий приклад роботи зі списками. Якщо є необхідність у розв’язанні

складніших задач, що, наприклад, передбачають динамічне наповнення списків числами з клавіатури, то необхідне використання такої операції, як розділення списку та голову та хвіст (РСГХ). У Prolog дана операція позначається символом `|`. Голова списку – це перший елемент списку. Хвіст – всі решта елементів. Наприклад, у списку `[1, 2, 3, 4, 5]` головою є число `1`, а хвостом – список `[2, 3, 4, 5]`.

Виведення списку із наперед невідомою кількістю елементів теж потребує використання операції РСГХ, інакше, подібно до наведеного вище третього прикладу, довелось би «вручну» вказувати кількість елементів списку. Таким чином, якщо виникне необхідність доповнення списку певними елементами чи виведення елементів списку, ми користуватимемось операцією РСГХ. Її синтаксис наступний:

список1 = [змінна|список2]

або із використанням введених термінів

список = [голова_списку|хвіст_списку]

або із використанням синтаксису мови Prolog

```
Z = [X|Y]
```

Цікаво, що, при використанні операції РСГХ, для добавлення у список нового елемента потрібно задати лише голову списку і його хвіст. І навпаки, щоб вивести голову списку чи його хвіст потрібно задати лише вихідний список. Відповідні вільні змінні заповняться

самим Prolog-ом. Якщо задані і список, і голова, і хвіст, то відповідний вираз перевіряється лише на істинність.

Тобто якщо ми маємо бажання добавити на початок списку [2, 3, 4, 5] елемент 1, то відповідний запис може мати вигляд:

```
X = 1, Y = [2, 3, 4, 5], Z = [X|Y], write(Z).
    %Z = [1, 2, 3, 4, 5]
```

Якщо ж навпаки, необхідно вивести перший елемент списку на екран, то відповідний код можна записати так:

```
Z = [1, 2, 3, 4, 5], Z = [X|Y], write(X).
    %X = 1, Y = [2, 3, 4, 5]
```

Перевірити ж чи X є головою, Y – хвостом і Z – результатом їх об'єднання можна так:

```
X = 1, Y = [2, 3, 4, 5], Z = [1, 2, 3, 4, 5],
    Z = [X|Y], write("Yes"), !; write("No").
    %Yes
```

Іншими словами, якщо ми хочемо вивести на екран, наприклад, перший елемент X списку Z, то потрібно задати цей список, помістити його в ліву сторону від знаку =, а справа розмістити вираз [X|Y], де X та Y мають бути вільними змінними. Якщо ж ми хочемо внести якийсь елемент у список, то потрібно, щоб лівий аргумент Z був вільною змінною, а два аргументи справа (X та Y) – мали певні значення.

Внесення кількох елементів у список чи виведення вмісту списку зазвичай (не зважди) відбувається рекурсивно.

Наведемо ще декілька прикладів застосування операції РСГХ. У деяких випадках корисним є відділити одночасно аж 2 голови. Тоді це здійснюється наступним чином:

```
[1, 2, 3, 4, 5] = [1, 2 | [3, 4, 5]]
    %Z = [1, 2, 3, 4, 5], Z = [X, Y | Z]
```

Можна також штучно (вручну) відокремити кілька голів:

```
[1, 2, 3, 4, 5] = [1|[2|[3|[4|[5|[]]]]]]
    %Z = [1, 2, 3, 4, 5], Z =
    [X1|[X2|[X3|[X4|[X5|[]]]]]]
```

Неможливим є використання операції РСГХ `Z = [X|Y]` у трьох випадках: якщо `X`, `Y`, `Z` або `X`, `Z`, або `Y`, `Z` є вільними змінними.

5.2. Основні алгоритми роботи зі списками

До основних алгоритмів роботи зі списками віднесемо наступні:

1. Наповнення списку елементами:
   а) із заданою кількістю ітерацій у зворотному порядку;
   б) із заданою кількістю ітерацій у прямому порядку;
   в) до виконання певної умови у зворотному порядку;
   г) до виконання певної умови у прямому порядку.
2. Вивід на екран.
3. Пошук елемента/елементів у списку:
   а) по значенню у випадку пошуку першого входження;
   б) по значенню у випадку пошуку всіх входжень;
   в) по позиції.

4. Вставка елемента у список:
   а) в задану позицію;
   б) замість/перед/після елемента/елементом/елемента/елементами із заданим значенням.
5. Видалення елемента зі списку:
   а) із заданої позиції;
   б) із заданим значенням.

Важливо, що оволодіння цими всіма алгоритмами дозволяє у подальшому виконувати більш складні дії над списками, наприклад, пошук елементів, сортування тощо.

Не зменшуючи загальності подальші викладки здійснюватимемо за умови, що елементи списку є типу `integer`.

Спершу наведемо розділи `domains` і `predicates`.

```
domains

    int = integer
    lint = integer*

predicates

    push_fr_a(int, lint, lint)%1а
    push_br_a(int, lint)%1б
    push_fr_b(lint, lint)%1в
    push_br_b(lint)%1г
    print(lint)%2
    search_a(int, int, int, lint)%3а
    search_b(int, int, int, int, lint)%3б
    search_v(int, int, lint, int, int)%3в
    insIn(int, int, int, lint, lint)%4а
    insAt(int, int, lint, lint)%4б
    delEl(int, lint, lint)%5а
    delIn(int, int, lint, lint)%5б
```

Тут кожен коментар до предиката відповідає номеру того чи іншого алгоритму роботи зі списками, які наводяться вище. Призначення властивостей, що відповідають кожному предикату, будуть пояснені нижче по ходу опису алгоритмів роботи зі списками. Також, для економії місця, незначні модифікації, які можна внести у певний рядок коду для отримання схожого результату, будуть представлені у вигляді коментарів. Для скорочення записів коди розділів `clauses` і `goal`, а також текст із вікна результатів виконання програми розділятимемо порожніми рядками.

1. Наповнення списку елементами.

У тих чи інших випадках зручним є наповнення списку елементами у прямому чи зворотному порядку. Це доцільно здійснювати за допомого ЗХР чи ПХР відповідно.

а) із заданою кількістю ітерацій у зворотному порядку.

У даному випадку достатнім є задання трьох властивостей для рекурсивного правила, а саме: кількість ітерацій `N`, список `T`, який поступово наповнюється і результуючий список `L`. Відповідні коди програми у розділах `clauses` і `goal`, а також текст із вікна результатів виконання програми можуть мати вигляд:

```
push_fr_a(N, T, L) :- N>0, !, write("input
    number: "), readint(H), TempT = [H|T],
    PrevN = N - 1, push_fr_a(PrevN,
    TempT,L).
push_fr_a(_, L, L).

push_fr_a(5, [], L), write("Result: ", L),
    1=2, 1=1.
```

```
input number: 11
input number: 22
input number: 33
input number: 44
input number: 55
Result: [55,44,33,22,11]
```

Перевірка твердження `push_fr_a` у вигляді `push_fr_a(5, [], L)` передбачає, що порожній список `[]` наповнюється (доповнюється) 5-ма елементами, введеними з клавіатури, які, в свою чергу, поміщаються у вільну змінну `L`. Оскільки у самому рекурсивному правилі наповнення списку `L` відбувається у порядку введення чисел (згідно ідеї ПХР), а в ТЗР відбувається копіювання результату, то, очікувано, результуючий список наповнився у зворотному порядку.

б) із заданою кількістю ітерацій у прямому порядку.

Тут достатнім є задання двох властивостей для рекурсивного правила, а саме: кількість ітерацій `N` і список `L`, який поступово наповнюється. При використанні ЗХР відповідний код можна записати 2-ма способами (другий спосіб передбачає коментування першого рядка і розкоментування другого).

```
push_br_a(N, L) :- N>0, !, write("input
    number: "), readint(H), PrevN = N - 1,
    push_br_a(PrevN, T), L = [H|T].
%або, що те саме, push_br_a(N, [H|T]) :- N>0, !,
    write("input number: "), readint(H),
    PrevN = N - 1, push_br_a(PrevN, T).
push_br_a(_, []).

push_br_a(5, L), write("Result: ", L), 1=2,
    1=1.
```

```
input number: 11
input number: 22
input number: 33
input number: 44
input number: 55
Result: [11,22,33,44,55]
```

Перевірка твердження `push_br_a` у вигляді `push_br_a(5, L)` передбачає, що список `L` наповнюється 5-ма елементами, введеними з клавіатури. Аналогічно до попереднього випадку, нові елементи у список `L` надходять на початок списку, проте сама черговість приєднання відбувається у зворотному (до введення) порядку (згідно ідеї ЗХР). Остаточно список наповнився у порядку введення чисел з клавіатури.

в) до виконання певної умови у зворотному порядку.

У даному випадку достатнім є задання трьох властивостей для рекурсивного правила, а саме: список `T`, який поступово наповнюється і результуючий список `L`. Умовою завершення введення чисел визначимо хибне виконання правила `readint(H)`, тобто введення з клавіатури символа, відмінного від цілого числа. У випадку ПХР відповідний код представимо у вигляді:

```
push_fr_b(T, L) :- write("input number: "),
    readint(H), !, TmpT = [H|T],
    push_fr_b(TmpT, L).
push_fr_b(L, L).

push_fr_b([], L), write("Result: ", L),
    1=2, 1=1.

input number: 44
input number: -25
```

```
input number: 17
input number: 0
input number: *
Result: [0,17,-25,44]
```

Перевірка твердження `push_fr_b` у вигляді `push_fr_b([], L)` передбачає, що порожній список `[]` наповнюється (доповнюється) введеними з клавіатури елементами, які поміщаються у вільну змінну `L`. Оскільки у самому рекурсивному правилі наповнення списку `L` відбувається у порядку введення чисел з клавіатури, а в ТЗР відбувається копіювання результату, то, очікувано, результуючий список наповнився у зворотному порядку.

г) до виконання певної умови у прямому порядку.

Тут достатнім є задання єдиної властивості для рекурсивного правила, а саме: список `[H|T]`, який поступово наповнюється.

```
push_br_b([H|T]) :- write("input number:
    "), readint(H), !, push_br_b(T).
push_br_b([]).

push_br_b(L), write("Result: ", L), 1=2,
    1=1.
```

```
input number: 44
input number: -25
input number: 17
input number: 0
input number: ю
Result: [44,-25,17,0]
```

Перевірка твердження `push_br_b` у вигляді `push_br_b(L)` передбачає, що список `L` наповнюється

введеними з клавіатури елементами. Аналогічно до попереднього випадку, нові елементи у список L надходять на початок списку, проте сама черговість приєднання відбувається у зворотному (до введення) порядку (згідно ідеї ЗХР). Остаточно список наповнився у порядку введення чисел з клавіатури.

2. Вивід на екран.

Вивід на екран списку здійснюється досить просто. Тут передбачається послідовне «ітераційне» виведення голови списку і передача отриманого хвоста на наступний рекурсивний виклик. При цьому, задля представлення списку у прямому порядку, відділення голови від хвоста та виведення голови відбувається на ПХР (цікаво, що для внесення чисел у список у прямому порядку, навпаки, використовувавсь ЗХР).

```
print([H|T]) :- write(H, ' '), print(T).
print([]).

write("Result:  "),  print([11,  22,  33,  44,
    55]).

Result: 11 22 33 44 55
```

Перевірка твердження `print` у вигляді `print([11, 22, 33, 44, 55])` передбачає, що список `[11, 22, 33, 44, 55]` виведеться на екран у прямому порядку.

3. Пошук елемента/елементів у списку.

а) по значенню у випадку пошуку першого входження.

У даному випадку достатнім є задання чотирьох властивостей для рекурсивного правила, а саме: поточна позиція `TmpPos`, знайдена позиція `Pos` шуканого елемента, шуканий елемент `X`, список `[H|T]`.

```
search_a(TmpPos, Pos, X, [H|T]) :- X<>H, !,
    TmpTmpPos = TmpPos + 1,
    search_a(TmpTmpPos, Pos, X, T).
search_a(Pos, Pos, X, [X|_]).
search_a(_, -1, _, []).

write("Result:  "),  search_a(1,  Position,
    33, [11, 22, 44, 33, 55]).

Result: Position=4
```

Тут рекурсивно відбувається відділення голови від хвоста до тих пір, поки X<>H. Як тільки X=H, то відбувається копіювання значення першої властивості у другу за допомогою першого з фактів. В інакшому випадку, якщо при проходженні всього списку не виявлено співпадань виду X=H, то у другому факті друга властивість набуває значення -1. У результаті виводиться або позиція входження шуканого елемента, або число -1. Перевірка твердження search_a у вигляді search_a(1, Position, 33, [11, 22, 44, 33, 55]) передбачає виведення позиції Position числа 33 у списку [11, 22, 44, 33, 55], вважаючи номер першого елемента списку рівним 1.

б) по значенню у випадку пошуку всіх входжень.

Тут достатнім є задання п’яти властивостей для рекурсивного правила, а саме: поточна позиція Pos, поточна кількість знайдених елементів TmpCount, сумарна кількість знайдених елементів Count, шуканий елемент X, список [H|T].

```
search_b(Pos, TmpCount, Count, X, [H|T]) :-
    X<>H,   !,   TmpPos   =   Pos   +   1,
    search_b(TmpPos, TmpCount, Count, X, T).
```

```
search_b(Pos, TmpCount, Count, X, [H|T]) :-
    write("Position of ", X, " number is ",
    Pos), nl, TmpTmpCount = TmpCount + 1,
    TmpPos = Pos + 1, search_b(TmpPos,
    TmpTmpCount, Count, X, T).
search_b(_, K, K, _, []).

write("Result: \n"), search_b(1, 0, Count,
    33, [11, 33, 22, 33, 44, 33]).

Result:
Position of 33 number is 2
Position of 33 number is 4
Position of 33 number is 6
Count=3
```

Тут рекурсивно відбувається відділення голови від хвоста до тих пір, поки X<>H. Кожен раз, коли X=H, відбувається виведення на екран позиції Pos входження шуканого елемента X, збільшення значення другої властивості, яка відповідає за кількість знайдених входжень елемента X у список і подальше рекурсивне відділення голови від хвоста. Після перебору всіх елементів у списку відбувається копіювання значення другої властивості у третю за допомогою наведеного факту. Перевірка твердження search_b у вигляді search_b(1, 0, Count, 33, [11, 33, 22, 33, 44, 33]) передбачає виведення усіх позицій входження числа 33 у список [11, 33, 22, 33, 44, 33], а також кількості Count таких входжень за умови, що номер першого елемента списку приймається рівним 1, а кількість знайдених входжень числа 33 ініціалізується нулем.

в) по позиції.

У даному випадку достатнім є задання чотирьох властивостей для рекурсивного правила, а саме: позиція Pos шуканого елемента, поточна позиція I, список [H|T], елемент X, який шукається у позиції Pos.

```
search_v(Pos, I, [H|T], X) :- Pos>0, I<Pos,
    !, NextI = I + 1, search_v(Pos, NextI,
    T,      X).      %%%%%%%%%%%%------
    %%%%%%%%%%%%%%%%%%%%%%%%%%%
search_v(Pos,   Pos,   [H|T],   H)   :-   !.
    %%%%%%%%%%%%---
search_v(_,    _,    [],    -99)    :-    !.
    %%%%%%%%%%%%---
search_v(Pos, _, _, -99) :- Pos<0.

write("Result: "), search_v(2, 1, [44, 33,
    22, 22], X).

Result: X=33
```

Тут рекурсивно відбувається відділення голови від хвоста до тих пір, поки I<Pos. Як тільки I=Pos, то відбувається копіювання значення голови у четверту властивість за допомогою другого правила. Якщо ж при проходженні всього списку не виявлено співпадань виду I=Pos, то у першому факті остання властивість набуває значення -99. Другий факт спрацьовує лише у випадку, якщо позиція шуканого елемента є меншою за 1. У результаті виводиться або позиція входження шуканого елемента, або число -99. Перевірка твердження search_v у вигляді search_v(2, 1, [44, 33, 22, 22], X]) передбачає виведення елемента X, який знаходиться в позиції 2 у списку [44, 33, 22, 22], вважаючи номер першого елемента списку рівним 1.

4. Вставка елемента у список.

а) в задану позицію.

У даному випадку достатнім є задання п'яти властивостей для рекурсивного правила, а саме: позиція Pos, в яку вставляється елемент X, поточна позиція I, заданий елемент X, вихідний список [H|TmpT], результуючий список [H|T].

```
insIn(Pos, I, X, [H|TmpT], [H|T]) :- Pos>0,
    I<Pos, !, NextI = I + 1, insIn(Pos,
    NextI, X, TmpT, T).
insIn(Pos, Pos, X, [Y|T], [X|T]).
insIn(_, _, X, [], []).

write("Result: "), insIn(3, 1, 100, [11,
    22, 33, 22], L).

Result: L=[11,22,100,22]
```

Тут рекурсивно відбувається відділення голови від хвоста за допомогою четвертої властивості (згідно ідей ПХР), поки I<Pos. Як тільки I=Pos, то у першому факті відбувається ініціалізація п'ятої властивості фрагментом вихідного списку, який починається із позиції Pos + 1 з одночасним приєднанням елемента X в якості голови. Далі, у процесі ЗХР, у першому правилі відбується приєднання перших Pos - 1 елементів списку. Якщо ж при проходженні всього списку не виявлено співпадань виду I=Pos, то у другому факті остання властивість ініціалізуються порожнім списком. Потім згідно ЗХР відбувається повне копіювання елементів з червертої властивості у п'яту. Перевірка твердження insIn у вигляді insIn(3, 1, 100, [11, 22, 33, 22], L) передбачає виведення списку L, який міститиме увесь список [11, 22, 33, 22] із заміненим

числом у позиції 3 на число 100, вважаючи номер першого елемента списку рівним 1.

б) замість/перед/після елемента/елементом/елемента/елементами із заданим значенням.

Тут достатнім є задання чотирьох властивостей для рекурсивного правила, а саме: значення X, яке необхідно вставити, значення Y, замість/перед/після якого вставлятиметься X, вихідний список [H|TmpT], результуючий список [H|T].

```
insAt(X, Y, [H|TmpT], [H|T]) :- Y<>H, !,
    insAt(X, Y, TmpT, T).
insAt(X, Y, [Y|T], [X|T]).%якщо необхідно
    вставити новий елемент замість заданого числа
%insAt(X, Y, [Y|TmpT], [X|T]) :- insAt(X,
    Y, TmpT, T).%якщо необхідно вставити новий
    елемент замість кожного заданого числа
%insAt(X, Y, [Y|T], [X,Y|T]).%якщо необхідно
    вставити новий елемент перед заданим числом
%insAt(X, Y, [Y|TmpT], [X,Y|T]) :- insAt(X,
    Y, TmpT, T).%якщо необхідно вставити новий
    елемент перед кожним заданим числом
%insAt(X, Y, [Y|T], [Y,X|T]).%якщо необхідно
    вставити новий елемент після заданого числа
%insAt(X, Y, [Y|TmpT], [Y,X|T]) :- insAt(X,
    Y, TmpT, T).%якщо необхідно вставити новий
    елемент після кожного заданого числа
insAt(_, _, [], []).

write("Result: "), insAt(100, 11, [11, 22,
    33, 22], L).

Result: L=[100,22,33,22]
```

Тут рекурсивно відбувається відділення голови від хвоста за допомогою третьої властивості (згідно ідеї ПХР),

поки Y<>H. Як тільки Y=H, то у другому твердженні замість/перед/після елемента/елементом/ елемента із значенням Y вставляється елемент/елементи X / X,Y / Y,X (все залежить від обраного способу вставки). У випадках множинної ставки виклик правила insAt у другому твердженні забезпечує подальші виклики рекурсивних правил. Далі, у процесі ЗХР, відбується приєднання початку списку. Якщо ж при проходженні всього списку не виявлено співпадань виду Y=H, то в останньому факті четверта властивість ініціалізуються порожнім списком. Потім згідно ЗХР відбувається повне копіювання елементів з третьої властивості у четверту. Перевірка твердження insAt у вигляді insAt(100, 11, [11, 22, 33, 22], L) передбачає виведення списку L, який міститиме увесь список [11, 22, 33, 22] із вставленим/вставленими замість/перед/після елемента/елементом 11 числа 100.

5. Видалення елемента зі списку.

   а) із заданої позиції.

У даному випадку достатнім є задання чотирьох властивостей для рекурсивного правила, а саме: поточна позиція I, позиція Pos, з якої видаляється елемент, вихідний список [H|TmpT], результуючий список [H|T].

```
delIn(I, Pos, [H|TmpT], [H|T]) :- I>0,
I<Pos, !, NextI = I + 1, delIn(NextI, Pos,
TmpT, T).
delIn(Pos, Pos, [H|T], T).
delIn(_, _, [], []).

write("Result: "), delIn(1, 2, [44, 33, 22,
33, 11], L).

Result: L=[44,22,33,11]
```

Тут рекурсивно у третій властивості відбувається відділення голови від хвоста (згідно ідей ПХР), поки I<Pos. Як тільки I=Pos, то у першому факті відбувається ініціалізація четвертої властивості фрагментом вихідного списку, який починається із позиції Pos + 1. Далі, у процесі ЗХР, у першому правилі відбується приєднання перших елементів списку аж до позиції Pos - 1. Таким чином не відбувається копіювання лише одного елемента, який знаходився у позиції Pos. Якщо ж при проходженні всього списку не виявлено співпадань виду I=Pos, то у другому факті остання властивість ініціалізуються порожнім списком. Потім згідно ЗХР відбувається повне копіювання елементів з третьої властивості у четверту. Перевірка твердження delIn у вигляді delIn(1, 2, [44, 33, 22, 33, 11], L) передбачає виведення списку L, який міститиме список [44, 33, 22, 33, 11] за виключенням елемента у позиції 2, вважаючи номер першого елемента списку рівним 1.

б) із заданим значенням.

Тут достатнім є задання трьох властивостей для рекурсивного правила, а саме: значення Y, яке необхідно видалити з вихідного списку, вихідний список [H|TmpT], результуючий список [H|T].

```
delEl(Y, [H|TmpT], [H|T]) :- Y<>H, !,
    delEl(Y, TmpT, T).
delEl(Y, [Y|T], T).%del some elemet
%delEl(Y, [Y|TmpT], T) :- delEl(Y, TmpT,
    T). %del all some elemets
delEl(_, [], []).

write("Result: "), delEl(33, [44, 33, 22,
    33, 11], L).
```

```
Result: L=[44,22,33,11]
```

Тут рекурсивно відбувається відділення голови від хвоста за допомогою другої властивості (згідно ідей ПХР), поки `Y<>H`. У випадку реалізації коду, при якому передбачається видалення єдиного елемента зі списку, після виконання умови `Y=H` слідує ініціалізація третьої властивості фрагментом вихідного списку, який складається з елементів, що слідують після `Y`. Далі, у процесі ЗХР, у першому правилі відбується приєднання перших елементів, які передують значенню `Y`. Таким чином не відбувається копіювання лише одного елемента у позиції, в якій знаходиться `Y`. Якщо ж при проходженні всього списку не виявлено співпадань виду `Y=H`, то в останньому факті третя властивість ініціалізуються порожнім списком. Потім згідно ЗХР відбувається повне копіювання елементів з другої властивості у третю. У випадку реалізації коду, при якому передбачається множинне видалення елементів, виконання умови `Y=H` передбачає черговий запуск рекурсивного правила, але без реалізації копіювання елементів зі значенням `Y`. Перевірка твердження `delEl` у вигляді `delEl(33, [44, 33, 22, 33, 11], L)` передбачає виведення списку `L`, який міститиме увесь список `[44, 33, 22, 33, 11]` із видаленим/видаленими елементом/елементами зі значенням `33`.

Контрольні запитання

1. Яка структура даних поміщається у квадратні дужки?

а) масиви;
б) об’єкти;
в) рядки;
г) списки;

д) класи.

2. Що оголошується ось таким чином: `lint = integer*`?

а) масив;

б) об’єкт;

в) вказівник;

г) список;

д) клас.

3. Що позначається символом `|`?

а) ПХР;

б) ЗХР;

в) РСГХ;

г) логічне *або*;

д) побітове *або*.

4. Яким буде результат виконання цільового запиту: `X = 1, Y = [2, 3, 4, 5], Z = [1, 2, 3, 4, 5], Z = [X|Y], write("Yes"), !; write("No").`?

а) `Yes`;

б) `No`;

в) помилка у зв'язку з переповненням стеку;

г) `YesNo`;

д) `NoYes`.

5. Яким буде результат виконання цільового запиту: `Z = [1, 2, 3, 4, 5], Z = [X1|[X2|[X3|[X4|[X5|[]]]]]]`?

а) `Z=[1, 2, 3, 4, 5], X1=1, X2=2, X3=3, X4=4, X5=5`;

б) `Z=[1, 2, 3, 4, 5], X1=[1, 2, 3, 4, 5]`, а решта змінних – вільні;

в) `Z=1, X1=2, X2=3, X3=4, X4=5`, а `X5` – вільна змінна;

г) `Z=[1, 2, 3, 4, 5], X5=[1, 2, 3, 4, 5]`;

д) `Z=[1], X1=[2, 3, 4, 5]`, а решта змінних – вільні.

6. Які з алгоритмів по обробці елементів списку вважають основними?

а) наповнення списку елементами;

б) вивід на екран;

в) пошук елемента/елементів у списку;

г) вставка елемента у список;

д) видалення елемента зі списку.

7. Нехай `T` – порожній список, тоді що означатиме запис для змінної `TempT` у випадку: `H = 1, TempT = [H|T]`?

а) `TempT=[];`

б) `TempT=[1];`

в) `TempT[,1];`

г) `TempT=1;`

д) немає вірної відповіді.

8. Що означає дана послідовність тверджень у випадку перевірки твердження `statement([1, 2, 3, 4, 5]): statement([H|T]) :- write(H, ' '), statement (T). statement ([]).`?

а) нічого не виведеться;

б) виникне помилка переповнення стеку;

в) послідовно виведуться цифри, розділені пробілами;

г) введеться кілька списків різної довжини;

д) виведуться цифри у зворотному порядку, розідлені пробілами.

9. Що можна зрозуміти із запису `[X, Y | Z]`?

а) відбувається обмін значень між змінними `X` та `Y`;

б) із списку `X` відділяється голова `Y`, а результат записується в `Z`;

в) до списку `X` приєднується голова `Y`, а результат записується в `Z`;

г) від голови `Z` відділяється 2 хвости: `X`, `Y`;

д) відбувається відділення/приєднання двох голів.

10. Що може означати перший аргумент, якщо другий – це позиція видалення числа зі списку, вказаного в третьому аргументі: `statement(1, 2, [44, 33, 22, 33, 11],`

L).?

а) кількість елементів, які буде видалено;
б) частота видалення елементів зі списку;
в) максимальна кількість перевірок твердження;
г) позиція початку пошуку елемента для видалення;
д) кількість спроб видалення елемента в позиції 2.

Розділ 6. Рядки

6.1. Вступ

Рядки і списки у Prolog мають багато спільного. Наприклад, для роботи з ними часто реалізовують рекурсивні правила, поелементну обробку організовують саме зліва направо, розроблювані алгоритми є часто аналогічними тощо.

Рядок – це послідовність символів таблиці ASCII, поміщена у лапки. Важливо, що максимальна кількість символів, яку може вмістити рядок, обмежується 4 Гб. Хоча при введенні з клавіатури із використанням твердження `readln` максимальна довжина тексту не повинна перевищувати 250 символів. Самі ці символи можуть бути введені або напряму в їх явному вигляді, або із використанням їх десяткових чи шітнадцяткових кодів.

Цікаво, що робота з рядками зазвичай є простішою за роботу зі списками, оскільки в Prolog вже реалізовано низку стандартних тверджень, які призначені для спрощення роботи з рядками. Наприклад, єдиним викликом твердження можна отримати довжину заданого рядка або об'єднати 2 вже існуючі рядки тощо.

6.2. Стандартні твердження для роботи з рядками

Нижче наведено синтаксиси, призначення (майже кожному з яких поставлено у відповідність певну букву) і приклади використання низки стандартних тверджень. При цьому, для кожного призначення наводяться приклади з результатами перевірки тверджень і відбуваються співставлення із буквами-призначеннями (у коментарях).

1. Твердження `str_len(string, integer)` у вигляді `str_len(X, Y)` призначене для (а) запису у вільну змінну `Y` довжини (кількості символів) заданого рядка `X`. Окрім

цього, якщо X та Y є заданими, то дане твердження має інше побічне призначення: перевіряється (б) чи довжина рядка X рівна числу Y. Наприклад:

```
str_len("Prolog", Y).%Y=6%а
str_len("Prolog", 6).%Yes%б
```

2. Твердження concat(string, string, string) у вигляді concat(X, Y, Z) призначене для (а) послідовного об'єднання рядків X та Y; результат записується у вільну змінну Z. Окрім цього, дане твердження має 3 інші побічні призначення, залежно від підставлених параметрів:

(б) записує у вільну змінну X початок рядка Z за умови, що кінцем рядка Z є вираз Y;

(в) записує у вільну змінну Y кінець рядка Z за умови, що початком рядка Z є вираз X;

(г) перевіряє чи рядок Z рівний послідовному об'єднанню рядків X та Y.

Наприклад:

```
concat("Pr", "olog", Z).%Z="Prolog"%а
concat(X, "olog", "Prolog").%X="Pr"%б
concat(X, "olog", "Lisp").%No%б
concat("Pr", Y, "Prolog").%Y="olog"%в
concat("Pr", Y, "Lisp").%No%в
concat("Pr", "olog", "Prolog").%Yes%г
concat("Pro", "olog", "Prolog").%No%г
```

3. Твердження frontchar(string, char, string) у вигляді frontchar(X, Y, Z) призначене для (а) запису у вільну змінну Y першого символа рядка X, а у вільну змінну Z – решту рядка X. Окрім цього, дане твердження має 2

інші побічні призначення, залежно від підставлених параметрів:

(б) записує у вільну змінну X результат послідовного об'єднання символа Y з рядком Z (подібно до твердження concat);

(в) перевіряє чи рядок X рівний послідовному об'єднанню символа Y та рядка Z.

Наприклад:

```
frontchar("Prolog", Y, Z).%Y='P',Z="rolog"%а
frontchar(X, 'P', "rolog").%X="Prolog"%б
frontchar("Prolog", 'P', "rolog").%Yes%в
```

4. Твердження frontstr(integer, string, string, string) у вигляді frontstr(X, Y, Z, W) має єдине призначення: відділяє від рядка Y підрядок, який складається з перших X його символів; даний підрядок записується у вільну змінну Z, а решта – у вільну змінну W. Наприклад:

```
frontstr(2, "Prolog", Z, W).%Z="Pr",W="olog"
frontstr(8, "Prolog", Z, W).%No
```

5. Твердження для конвертації рядків та символів.

Твердження str_int(string, int) та str_real(string, real) у відповідних виглядах str_int(X, Y) та str_real(X, Y) призначені для:

(а) конвертації цілого/дійсного числа Y в рядок X;

(б) присвоєння у вільну змінну Y цілого/дійсного числа, якщо рядок X складається із коректного запису цього числа.

Окрім цього, дані твердження мають інше побічне призначення: якщо (в) одночасно X та Y є заданими, то перевіряється чи рядок X відповідає (рівний) числу Y. Наприклад:

```
str_int(X, -52).%X=-52%а
str_int("999999999", Y).%Y=999999999%б
str_int("9999999999", Y).%No%б
str_int("1a", Y).%No%б
str_int("12.5", Y).%No%б
str_int("\49", Y).%Y=1%б
str_int("999999", 999999).%Yes%в
str_real(X, 99.9999).%X=99.9999%а
str_real("999999999", Y).%Y=999999999%б
str_real("9999999999", Y).%Y=9999999999%б
str_real("1a", Y).%No%б
str_real("12.5", Y).%Y=12.5%б
str_real("\49\50\46\54", Y).%Y=12.6%б
str_real("99.9999", 99.9999).%Yes%в
```

Пара тверджень `str_char(string, char)` та `char_int(char, int)` у відповідних виглядах `str_char(X, Y)` та `char_int(X, Y)` призначені для:

(а) конвертації символа/коду Y у відповідний рядок/символ X;

(б) конвертації рядка/символу X у відповідний символ/код Y.

Окрім цього, дані твердження мають інше побічне призначення, якщо (в) одночасно X та Y є заданими: перевірка чи рядок/символ X відповідає (рівний) символу/числу Y. Наприклад:

```
str_char(X, 'p').%X="p"%а
str_char("P", Y).%Y='P'%б
```

```
str_char("Prolog", Y).%No%б
str_char("P", 'P').%Yes%в
str_char("P", 'p').%No%в
char_int(X, 123).%X='{'%а
char_int('{', Y).%Y=123%б
char_int('{', 123).%Yes%в
```

Твердження `upper_lower(string, string)` у відповідному вигляді `upper_lower(X, Y)` призначене для:

а) конвертації рядка `Y` у рядок `X`, замінивши усі малі букви на великі;

б) конвертації рядка `X` у рядок `Y`, замінивши усі великі букви на малі.

Окрім цього, дане твердження має інше побічне призначення: перевіряє (в) чи рядок `X` рівний рядку `Y`, але ігноруючи регістр. Наприклад:

```
upper_lower(X, "Prolog v.5").%X=PROLOG V.5%а
upper_lower("Prolog v.5", Y).%Y=prolog v.5%б
upper_lower("Prolog  v.5.2", "Prolog  v.5.2").
    %Yes%в
upper_lower("PrOlOg  v.5.2", "Prolog  v.5.2").
    %Yes%в
upper_lower("Lisp    v.5.2", "Prolog  v.5.2").
    %No%в
```

6. Твердження `isname(string)` у вигляді `isname(X)` має єдине призначення: перевірка належності рядка `X` до ідентифікаторів мови Prolog. Тут ідентифікатор – це послідовність букв, цифр і знаку `_`, яка починається із букви (за виключенням ключових слів, наприклад, domains) або символа `_`. Наприклад:

```
isname("").%No
```

```
isname("Prolog123").%Yes
isname("_").%Yes
isname("12x").%No
```

7. Твердження `fronttoken(string, string, string)` у вигляді `fronttoken(X, Y, Z)` призначене для (а) запису у вільні змінні `Y` та `Z` відповідних атома та залишку відділення атома від початку рядка `X`. Тут атом – це елементарний об'єкт мови Prolog. До таких об'єктів відносяться:

- будь-яка послідовність букв, цифр і (або) символа _, яка починається із малої букви чи знаку _;
- набір символів, який обмежуються лапками;
- спеціальні символи (+, –, *, =, <, > тощо).

Серед неатомів варто виділити пробільний символ та знак табуляції.

Окрім цього, твердження `fronttoken(X, Y, Z)` має 4 інші побічні призначення, залежно від підставлених параметрів:

(б) записує у вільну змінну `X` результат послідовного об'єднання рядків `Y` і `Z` (подібно до твердження `concat`);

(в) записує у вільну змінну `Y` перший атом рядка `X`, а також перевіряє чи рядок `Z` рівний залишку відділення атома від початку рядка `X`;

(г) записує у вільну змінну `Z` залишок відділення атома `Y` від початку рядка `X`;

(д) перевіряє чи `Y` та `Z` є відповідними атомом та залишком відділення атома від початку рядка `X`.

Наприклад:

```
fronttoken("x1^2+y1^2=1", Y, Z).
    %Y="x1", Z="^2+y1^2=1"%a
```

```
fronttoken(" ", Y, Z).
    %No%а
fronttoken(X, "x1+ , ss", "^2+y1^2=1").
    %X=x1+ , ss^2+y1^2=1%б
fronttoken("  x1^2+y1^2=1", Y, "^2+y1^2=1").
    %Y="x1"%в
fronttoken("x1^2+y1^2=1", "x1", Z).
    %Z="^2+y1^2=1"%г
fronttoken("x1^2+y1^2=1","x1","^2+y1^2=1").
    %Yes%д
```

6.3. Основні алгоритми для роботи з рядками

Використання наведених вище тверджень є важливим інструментом, завдяки якому робота з рядками стає суттєво простішою, ніж робота зі списками. До основних алгоритмів роботи з рядками віднесемо наступні:

1. Пошук позиції символа в рядку.
2. Вставка тексту в певну позицію рядка.
3. Видалення фрагмента рядка.
4. Конвертація:
   а) рядка в список символів;
   б) списку символів в рядок.

Важливо, що оволодіння цими всіма алгоритмами дозволяє у подальшому виконувати більш складні дії над рядками, наприклад, обчислювати значення математичних виразів, здійснювати перевірку орфографії у тексті, виконувати автозаміну при невірно введених словах тощо.

Спершу наведемо розділи `domains` і `predicates`.

```
domains

    int = integer
    list_c = char*
```

```
predicates

    search(char, int, int, string)%1
    ins(int, string, string, string)%2
    del(int, int, string, string)%3
    stoc(string, list_c)%4а
    ctos(list_c, string)%4б
```

Тут кожен коментар до предиката відповідає номеру (див. вище) того чи іншого алгоритму роботи з рядками. Призначення властивостей, що відповідають кожному предикату, будуть пояснені нижче по ходу опису алгоритмів роботи з рядками. Для економії місця коди розділів `clauses` і `goal`, а також текст із вікна результатів виконання програми розділятимемо порожніми рядками.

1. Пошук позиції символа в рядку

У даному випадку достатнім є задання чотирьох властивостей для рекурсивного правила, а саме: деякий символ `Symbol`, поточна позиція `I`, знайдена позиція `Pos`, заданий рядок `Str`. Відповідний код програми у розділах `clauses` і `goal`, а також текст із вікна результатів виконання програми можуть мати вигляд:

```
search(Symbol, I, Pos, Str) :- frontchar(Str,
    H, T), Symbol<>H, !, NextI = I + 1,
    search(Symbol, NextI, Pos, T).
search(Symbol, Pos, Pos, Str) :-
    frontchar(Str, Symbol, _), !.
search(_, _, -1, "").

search('o', 1, Pos, "Prolog"), write("Result:
    position=", Pos), 1=2, 1=1.

Result: position=3
```

Тут рекурсивно відбувається відділення від рядка Str першого символа до тих пір, поки не буде знайдено символ, рівний Symbol або поки рядок Str не стане порожнім. У першому випадку перевірка другого правила приводить до присвоєння позиції входження символа Symbol у рядок Str. Інакше, якщо символ Symbol не знайдений, то у єдиному факті третя властивість (яка відповідає за позицію) набуває значення -1. Перевірка твердження search у вигляді search('o', 1, Pos, "Prolog") передбачає, що буде виведена позиція Pos першого входження символа 'o' у рядок "Prolog", вважаючи номер першого елемента списку рівним 1.

2. Вставка тексту в певну позицію рядка

У даному випадку достатнім є задання чотирьох властивостей для рекурсивного правила, а саме: позиція Pos вставки, вихідний текст TmpStr, текст Text для вставки, результуючий рядок Str.

```
ins(Pos, TmpStr, Text, Str) :- Pos > 1,
    frontchar(TmpStr, H, T), !, PrevP = Pos –
    1,   ins(PrevP,   T,   Text,   TmpTmpStr),
    frontchar(Str, H, TmpTmpStr).
ins(_, TmpStr, Text, Str) :- concat(Text,
    TmpStr, Str).

ins(3, "Prog", "ol", Str), write("Result: ",
    Str), 1=2, 1=1.

Result: Prolog
```

Тут рекурсивно відбувається відділення від рядка TmpStr першого символа до тих пір, поки не буде досягнуто символа в позиції Pos. Після цього відбувається ініціалізація четвертої властивості другого правила

рядком, який утворюється об'єднанням символа Text з залишком рядка TmpStr (без перших Pos – 1 символів вихідного рядка). У першому правилі на ЗХР результуючий рядок доповнюється першими Pos – 1 символами.

Цікаво, що наведений рекурсивний алгоритм можна замінити відповідним лінійним із використанням стандартних тверджень у вигляді:

```
ins(Pos, TmpStr, Text, Str) :- Len = Pos – 1,
    frontstr(Len,  TmpStr,  Str1,  Str2),
    concat(Str1,  Text,  Str3),  concat(Str3,
    Str2, Str).
```

Перевірка обох варіантів реалізації твердження ins у вигляді ins(3, "Prog", "ol", X) передбачає, що у змінній X буде розміщений результат вставлення рядка "ol" перед третім символом рядка "Prog".

3. Видалення фрагмента рядка

У даному випадку достатнім є задання чотирьох властивостей для рекурсивного правила, а саме: позиція Pos видалення, довжина Len підрядка для видалення, вихідний текст TmpStr, результуючий рядок Str.

```
del(Pos,  Len,  TmpStr,  Str)  :-  Pos  >  1,
    frontchar(TmpStr, H, T), !, PrevP = Pos -
    1,   del(PrevP,   Len,   T,   TmpTmpStr),
    frontchar(Str, H, TmpTmpStr).
del(_,  Len,  TmpStr,  Str)  :-  frontstr(Len,
    TmpStr, _, Str).

del(4, 3, "ProPrrlog", Str), write("Result: ",
    Str), 1=2, 1=1.

Result: Prolog
```

Тут рекурсивно відбувається відділення від рядка TmpStr першого символа до тих пір, поки не буде досягнуто символа в позиції Pos. Після цього відбувається ініціалізація четвертої властивості другого правила рядком, який утворюється видаленням перших Len символів із залишку рядка TmpStr (без перших Pos – 1 символів вихідного рядка). У першому правилі на ЗХР результуючий рядок доповнюється першими Pos – 1 символами.

Цікаво, що наведений рекурсивний алгоритм можна замінити відповідним лінійним із використанням стандартних тверджень у вигляді:

```
del(Pos, Len, TmpStr, Str) :- PrevP = Pos –
    1, frontstr(PrevP, TmpStr, Str1, Str2),
    frontstr(Len, Str2, Str3, Str4),
    concat(Str1, Str4, Str).
```

Перевірка обох варіантів реалізації твердження del у вигляді del(4, 3, "ProPrrlog", Str) передбачає, що у змінній Str буде розміщений результат видалення з рядка "ProPrrlog" трьох символів, починаючи з позиції 4.

4. Конвертація

   а) рядка в список символів

   У даному випадку достатнім є задання двох властивостей для рекурсивного правила, а саме: рядок Str і результуючий список [H|T].

```
stoc(Str, [H|T]) :- frontchar(Str, H, TmpStr),
    !, stoc(TmpStr, T).
stoc("", []).
```

```
stoc("Prolog v.5.2", L), write("Result: ", L),
    1=2, 1=1.

Result:   ['P','r','o','l','o','g',' ','v','.',
    '5','.','2']
```

Тут на ПХР рекурсивно відбувається відділення від рядка `Str` символів, допоки рядок `Str` не стане порожнім. Тоді у факті ініціалізується порожній список. Далі на ЗХР відбувається наповнення списку символами. Перевірка твердження `stoc` у вигляді `stoc("Prolog v.5.2", L)` передбачає, що буде виведений список `L`, який складається із усіх символів рядка `"Prolog v.5.2"`.

б) списку символів в рядок

У даному випадку достатнім є задання двох властивостей для рекурсивного правила, а саме: результуючий список `[H|T]` і рядок `Str`.

```
ctos([H|T], TmpStr) :- ctos(T, Str),
    frontchar(TmpStr, H, Str).
ctos([], "").

ctos(['P','r','o','l','o','g',' ','v','.','5',
    '.','2'],  Str),  write("Result:  ",  Str),
    1=2, 1=1.

Result: Prolog v.5.2
```

Тут на ПХР рекурсивно відбувається відділення від списку `[H|T]` символів, допоки список `[H|T]` не стане порожнім. Тоді у факті ініціалізується порожній рядок. Далі на ЗХР відбувається посимвольне наповнення рядка. Перевірка твердження `ctos` у вигляді `ctos(['P','r','o','l','o','g',' ','v','.','5','.','`

2'], Str) передбачає, що буде виведений рядок Str, який складається із усіх символів списку ['P','r', 'o','l','o','g',' ','v','.','5','.','2'].

Контрольні запитання

1. Що таке рядок?

а) послідовність символів, яка закінчується крапкою;

б) послідовність символів, яка починається з малої букви або символу підкреслення;

в) послідовність сиволів, поміщена в лапки;

г) структура даних, яка нагадує union в C++;

д) структура даних, яка нагадує struct в C++.

2. Скільки різних призначень може мати твердження concat?

а) 1;

б) 2;

в) 3;

г) 4;

д) 5.

3. Оберіть вірне висловлення щодо тверджень frontstr та frontchar

а) Обидва працюють з рядками;

б) Перше твердження працює лише з рядками, а друге – лише з символами;

в) Обидва твердження працюють лише з символами;

г) Обидва твердження працюють однаково;

д) Обидва твердження працюють з шрифтами.

4. Що з переліченого стосується твердження char_int?

а) Воно призначене для конвертації коду в символ;

б) Воно призначене для конвертації символа в код;

в) Воно призначене для конвертації рядка символів в код;

г) Воно призначене для конвертації коду в рядок

символів;

д) Перевіряє чи код символа відповідає числу.

5. Для чого призначене твердження `fronttoken`?

а) Для порівняння атомів двох рядків;

б) Для порівняння залишків відділення атомів двох рядків;

в) Для заміни символів;

г) Для приєднання атома до рядка;

д) Для відділення атома від рядка.

6. Яке твердження є найбільш корисним при пошуку символа в рядку?

а) `str_len`;

б) `concat`;

в) `str_int`;

г) `frontchar`;

д) `fronttoken`.

7. Які твердження, серед перерахованих, є найбільш корисними при вставці символа в рядок?

а) `str_len`;

б) `concat`;

в) `str_int`;

г) `frontchar`;

д) `fronttoken`.

8. Які 2 твердження, серед перерахованих, є найбільш корисними при видаленні символа з рядка?

а) `str_len`;

б) `concat`;

в) `str_int`;

г) `frontchar`;

д) `fronttoken`.

9. Яке твердження є найбільш корисним при конвертації радка у список символів?

а) `str_len`;

б) `concat`;

в) `str_int`;

г) `frontchar`;

д) `fronttoken`.

10. Яке твердження є найбільш корисним при конвертації списка символів у радок?

а) `str_len`;

б) `concat`;

в) `str_int`;

г) `frontchar`;

д) `fronttoken`.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Bramer M. Logic Programming with Prolog. 2nd ed. London : Springer, 2013. 253 p.

2. Bratko I. Prolog Programming for Artificial Intelligence. 4th ed. Toronto : Pearson Education Canada, 2011. 696 p.

3. Tate B. Programmer Passport: Prolog. 1st ed. Raleigh : Pragmatic Bookshelf, 2022. 120 p.

4. Buscaroli R., Chesani F., Giuliani G., Loreti D., Mello P. A Prolog application for reasoning on maths puzzles with diagrams. *Journal of Experimental & Theoretical Artificial Intelligence*. 2022. Vol. 35 (7). P. 1079–1099.

5. Dietz E., Philipp T., Schramm G., Zindel A. A Logic Programming Approach to Global Logistics in a Co-Design Environment. *Electronic Proceedings in Theoretical Computer Science*. 2023. Vol. 385. P. 227–240. DOI: 10.4204/EPTCS.385.23.

6. Dovier A., Formisano A., Gupta G., Hermenegildo M., Pontelli E., Rocha R. Parallel Logic Programming: A Sequel. *Theory and Practice of Logic Programming*. 2022. Vol. 22 (6). P. 905–973.

7. Costa E. Visual Prolog 7.3 for Tyros. New York : Springer-Verlag, 2010. 270 p.

8. Fernando C.N.P., Stuart M. S. Prolog and Natural-Language Analysis. Brookline : Microtome Publishing, 2002. 252 c.

9. Fernandes P. A. Space Syntax with Logic Programming: An Application to a Modern Estate. *Urban Science*. 2023. Vol. 7 (3). 22 p.

10. Schrijvers T., van den Berg B., Riguzzi F. Automatic Differentiation in Prolog. *Theory and Practice of Logic Programming*. 2023. Vol. 23(4). P. 900–917.

11. Toni F., Potyka N., Ulbricht M., Totis P. Understanding

ProbLog as Probabilistic Argumentation. *Electronic Proceedings in Theoretical Computer Science*. 2023. Vol. 385. P. 183–189. DOI: 10.4204/EPTCS.385.18.

12. Visual Prolog: A Versatile Programming Language. URL: https://www.visual-prolog.com/ (Last accessed: 05.06.2024).

13. Warren S. D., Dahl V., Eiter T., Hermenegildo V. M., Kowalski R., Rossi F. Prolog: The Next 50 Years. Lecture Notes in Computer Science Series. Cham : Springer, 2023. 394 p.

14. Singh Y. Handbook on Prolog Language: Experiments based on Artificial Intelligence & introduction to Neural Networks. London : LAP LAMBERT Academic Publishing, 2019. 72 p.

15. Zaiats V. M., Zaiats M. M. Logical and functional programming. Tutorial. Kamianets-Podilskyi : Ruta, 2016. 400 p.

16. Zheng X. Building Conventional “Experts” With a Dialogue Logic Programming Language. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*. 2023. Vol. 31. P. 1784–1796.

17. Дейнега Л. Ю., Камінська Ж. К., Левада І. В., Сердюк С. М. Практичне програмування мовою Visual Prolog : навч. посіб. Запоріжжя : Національний університет «Запорізька політехніка», 2016. 236 с.

18. Заяць В. М., Заяць М. М. Логічне і функціональне програмування. Системний підхід : підручник. 2-ге вид., випр. та допов. Рівне : Національний університет водного господарства та природокористування, 2018. 422 с.

19. Месюра В. І., Лисак Н. В., Суприган О. І. Математичні основи логічного програмування : навч. посіб. Вінниця : Вінницький національний технічний університет, 2013. 94 с.

20. Новожилова М. В., Петрова О. О. Використання мови логічного програмування Visual Prolog для розробки експертних систем : навч. посіб. Харків : Харківський національний університет міського господарства ім. О.М. Бекетова, 2019. 89 с.

21. Темнікова О. Л. Математична логіка та теорія алгоритмів: конспект лекцій : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2021. 177 с.

22. Шумейко О. О., Кнуренко В. М. Visual Prolog. Опануй на прикладах : навч. посіб. Дніпро : Біла К.О., 2014. 404 с.

Додатки

Додаток А
Список визначень професійної термінології

Prolog – мова логічного програмування загального призначення, яка заснована на логіці предикатів диз'юнкторів Хорна. Назва походить від PROgramming in LOGic (ПРОграмування мовою ЛОГіки).

`TextMode` – це значення `UI Strategy`, яке задається з метою створення саме консольного додатку.

Анонімна змінна – змінна, яка не має імені, та коли не є важливим її значення.

Арність – кількість властивостей (термів домену), які задано для терма предикату.

Атом – це послідовність букв, цифр і символів підкреслення, починаючи з малої літери або спеціальних символів чи будь-яких символів, взятих у лапки.

Відсікання (`!`) – операція, яка призначеня для видалення усіх створених раніше точок розгалуження.

Вільна змінна – змінна, яка ще не набула значення.

Властивості твердження – атрибути або характеристики твердження, які розміщуються у дужках після індентифікатора твердження.

Декларативні мови програмування – парадигма програмування, відповідно до якої програма описує, який результат необхідно отримати, замість описання послідовності отримання цього результату.

Детермінований режим – режим виконання програми, при якому відомо, що певний предикат може повертати лише одне рішення.

Домени (`domains`) – типи даних у Prolog.

Домени з альтернативами – домени, які мають декілька можливих значень.

Змінна – послідовність символів, яка починається з великої букви та може представляти певне значення.

Змінна зі значенням – змінна, яка набула значення.

Ідентифікатор твердження (ідентифікатор відношення) – унікальний ідентифікатор, який ототожнюється з конкретним твердженням.

Імперативні мови програмування – парадигма програмування (стиль написання вихідного коду комп'ютерної програми), згідно з якою описується процес (детальний алгоритм) отримання результатів як послідовність інструкцій зміни стану програми.

Інтелектуальні системи (системи штучного інтелекту) – програмно-апаратні комплекси, які призначені для виконання якихось цілей таким чином, щоб імітувати мисленнєву діяльність людини.

Істинний (хибний) результат – результат виконання деякого твердження у тілі іншого твердження із логічним значенням `Yes` (`No`).

Лінійні (або, що те саме, послідовні) обчислення – алгоритмічна конструкція, при якій усі твердження деякого правила розділені *логічним і*.

Логічне програмування – це підхід до програмування, в якому основний наголос робиться на логічне слідування і декларативний характер опису програми.

Механізм відкату – механізм, який дозволяє Prolog «відміняти» попередні обчислення та шукати інші варіанти вирішення.

Обчислення з розгалуженням – алгоритмічна конструкція, при якій перевірка істинності твердження допускає використання кількох альтернативних віток.

Остаточно істинний (остаточно хибний) результат – результат виконання усього тіла деякого твердження із логічним значенням `Yes` (`No`).

Перевірка (виклик) твердження – перевірка чи при даному варіанті вхідних даних те чи інше твердження набуває істинного значення; у позитивному випадку можливе повернення певного результату.

Повторювані обчислення – вид алгоритмічних конструкцій, при яких передбачається багаторазова перевірка одного і того ж тіла твердження при, можливо, різних даних.

Предикати (`predicates`) – символьні імена відношень, які встановлюють зв'язки між доменними типами; це є шаблони, за якими будуються твердження-факти і твердження-правила.

Прямий/зворотній хід рекурсії – два режими рекурсії, при яких обчислення здійснюються рекурсивно в прямому/зворотньому напрямках.

Рекурсія – це, подібно до інших мов програмування, така програмна конструкція, при якій у тілі певного твердження викликається твердження з тим самим іменем.

Рядок – це послідовність символів таблиці ASCII, поміщена у лапки.

Складені домени – домени, які об'єднують у собі декілька інших доменних типів під єдиною назвою.

Список – це впорядкована послідовність елементів певного конкретного типу, поміщена у квадратні дужки.

Твердження (`clauses`) – це логічні вирази, кожне з яких складається з голови та, можливо, тіла у вигляді: голова правила якщо тіло правила.

Твердження-правило – це твердження, істинність якого перевіряється у результаті співставлення усіх наявних тверджень про це правило.

Твердження-факт – це твердження-«аксіома» з порожнім тілом.

Терм – синтаксична одиниця мови Prolog, до якої відносять атоми, числа, змінні та структури.

Терм домену – це назва (ідентифікатор) доменного типу, яка починається з малої букви.

Точка відкату/розгалуження – місце у твердженні, в яке виконання програми може повернутися для пошуку альтернативних рішень, що призводили б до отримання від даного твердження остаточно істинного результату.

Уніфікаційні підпрограми Prolog – підпрограми, які без участі програміста керують процесом набуття значень змінних і відкату.

Функтор – це терм, назва якого задається програмістом при оголошенні складеного домену або домену з альтернативами.

Функціональне програмування – це парадигма програмування, заснована на використанні функцій як основного будівельного блоку програми.

Цільова мета (мета; цільовий запит) (`goal`) – вираз, який застосовується до існуючих тверджень з метою отримати ті чи інші умовиводи.

Штучне хибне твердження – це твердження, яке завжди є хибним.

Додаток Б

Список скорочень

ВВ – відкат із відсіканням.
ВПН – відкат після невдачі.
ЗХР – зворотній хід рекурсії.
ПВП – повторення визначене програмістом.
ПХР – прямий хід рекурсії.
ТЗР – точка зупинки рекурсії.

Додаток В

Інші позначення

Під змінною з префіксом `Next` (перекладається як наступний), наприклад, `NextI`, позначаємо `I + 1`.

Під змінною з префіксом `Prev` (перекладається як попередній), наприклад, `PrevN`, позначаємо `N - 1`.

Під змінною з префіксом `Tmp` (слово temp позначає щось тимчасове), наприклад, `TmpSum`, позначаємо проміжне (тимчасове) значення змінної `Sum` (сума).

Додаток Г

Домовленості

Для зручності написання та розуміння написаного коду доцільно притримуватись низки домовленостей.

Для забезпечення «хорошого тону» при програмуванні притримуватимемось наступних правил:

- для кожного предиката декларувати спочатку перелік тверджень-фактів, а потім – тверджень-правил;
- кожне твердженні-правило та твердження-факт декларувати у різних рядках;
- твердження декларувати у тій же послідовності, що й предикати;
- в усіх фрагментах коду, де можливо поставити пробільний символ без виникнення помилки, ставити пробіл, за виключенням випадку порівняння операндів (щоб відрізняти від присвоєння).

Для скорочення записів замість термінів твердження-факт і твердження-правило інколи вживатимуться слова факт, правило або твердження, залежно від контексту. У низці випадків скорочення виду `s = symbol` суттєво спрощує процес написання коду і підвищує його читабельність.

Додаток Д

Пояснення слів, наведених у посібнику латиницею

№	Англійською	Українською
1.	address	адреса
2.	all	всі
3.	are	є
4.	display	показувати
5.	above	вище
6.	greetings	вітання
7.	apple	яблуко
8.	area	площа
9.	arrow	стрілка
10.	author	автор
11.	birds	птахи
12.	cond (condition)	умова
13.	cycle	цикл
14.	door	двері
15.	head (H)	голова
16.	hello	привіт
17.	item	річ
18.	kluch (key)	ключ
19.	laboratory work	лабораторні(на) роботи(та)
20.	likes	подобається
21.	list	список
22.	music	музика
23.	number	число
24.	orange	апельсин
25.	owner/owns	власник/володіти
26.	passed	здав
27.	person	особа
28.	price	ціна

29.	realty	нерухомість
30.	continue	продовження
31.	rule	правило
32.	search	шукати
33.	see/seen	бачити/побачене
34.	sol (salt)	соль (сіль)
35.	statem (statement)	твердження
36.	tail (T)	хвіст
37.	times	разів
38.	title	назва (книги)
39.	triangle	трикутник
40.	type (transport type)	вид транспорту
41.	year	рік
42.	bomb shelter	бомбосховище
43.	choose an option	виберіть варіант
44.	run	запуск
45.	underground	підземелля
46.	start	початок
47.	recursion (recursive)	рекурсія (рекурсивне)
48.	position (Pos)	позиція
49.	forward	прямий
50.	backward	зворотній

Додаток Е

«Розшифровування» термів деяких тверджень

№	Твердження	Переклад	Призначення
1.	`fl_fr_incr`	factorial forward increment	обчислення факторіалу на прямому ході рекурсії у прямому порядку множення.
2.	`fl_fr_decr`	factorial forward decrement	обчислення факторіалу на прямому ході рекурсії у зворотному порядку множення.
3.	`fl_br_decr`	factorial backward decrement	обчислення факторіалу на зворотному ході рекурсії у зворотному порядку множення.
4.	`fl_br_incr`	factorial backward increment	обчислення факторіалу на зворотному ході рекурсії у прямому порядку множення.
5.	`sum_pow_fr`	sum power forward	обчислення суми чисел, піднесених до певних степенів, на прямому ході рекурсії.
6.	`sum_pow_br`	sum power backward	обчислення суми чисел, піднесених до певних степенів, на зворотному ході рекурсії.
7.	`exp_fr`	exponent forward	обчислення експоненти деякого числа на прямому ході рекурсії.
8.	`exp_br`	exponent backward	обчислення експоненти деякого числа на

			зворотному ході рекурсії.
9.	`push_fr`	push forward	наповнення списку елементами на прямому ході рекурсії.
10.	`push_br`	push backward	наповнення списку елементами на зворотному ході рекурсії.
11.	`insIn`	insert in	вставка елемента в певну позицію у списку.
12.	`insAt`	insert at	вставка елемента біля певної позиції у списку.
13.	`delEl`	delete element	видалення елемента/елементів із певним значенням зі списку.
14.	`delIn`	delete in	видалення елемента з певної позиції у списку.
15.	`stoc`	string to char	конвертація рядка у список символів.
16.	`ctos`	char to string	конвертація списку символів у рядок.

Додаток Ж

Стандартні позначення та твердження

№	Твердження чи операція	Призначення
1.	<	менше
2.	>	більше
3.	=	злиття (тобто присвоєння); порівняння
4.	<>	недорівнює
5.	.	розділювач між цілою та дробовою частиною числа; ознака завершення правила чи цільової мети
6.	, (and)	логічне і
7.	; (or)	логічне або
8.	not()	твердження логічного не
9.	:- (if)	зворотна імплікація
10.	+	додавання
11.	-	віднімання
12.	*	множення
13.	/	ділення
14.	[]	список
15.	|	операція розділення списку на голову та хвіст
16.	'c'	символ
17.	%коментар	однорядковий коментар
18.	/*коментар*/	багаторядковий коментар
19.	"текст"	текст
20.	'\n', '\t'	використовується для перенесенні тексту на новий рядок
21.	_	анонімна змінна
22.	!	знак відсікання

23.	`fail`	штучне хибне твердження
24.	`nl`	твердження, за допомогою якого відбувається перехід на новий рядок у консольному вікні
25.	`readln`	використовується для зчитування рядка з консолі
26.	`write`	використовується для виведення даних на консоль
27.	`readint`	призначене для зчитування цілих чисел з консолі
28.	`readchar`	дозволяє зчитувати один символ з консолі
29.	`str_len`	визначення довжини (кількості символів) заданого рядка
30.	`concat`	об'єднання двох рядків
31.	`frontchar`	відділення першого символа рядка; решта рядка поміщається у вільну змінну
32.	`frontstr`	відділення певної кількості символів з початку рядка; решта рядка поміщається у вільну змінну
33.	`str_int`	конвертація цілого числа в рядок та навпаки
34.	`str_real`	конвертація дійсного числа в рядок та навпаки
35.	`str_char`	конвертація символа у рядок та навпаки
36.	`char_int`	конвертація коду у символ та навпаки
37.	`upper_lower`	конвертація наявного рядка у новий рядок, в якому замінені усі малі букви на великі або навпаки
38.	`isname`	перевірка належності деякого рядка

		до ідентифікаторів мови Prolog
39.	`fronttoken`	запис у вільні змінні атома та залишку відділення атома від початку деякого рядка

Додаток И

Стандартні доменні типи

```
integer
real
char
string
file
symbol
```

Навчальне видання

Бойчура Михайло Володимирович
Сидор Андрій Іванович

ЛОГІЧНЕ ПРОГРАМУВАННЯ НА МОВІ PROLOG

Навчальний посібник

Друкується в авторській редакції

Технічний редактор Галина Сімчук

Видавець і виготовлювач
Національний університет
водного господарства та природокористування
вул. Соборна, 11, м. Рівне, 33028.

Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції РВ № 31 від 20.04.2005 р.