Міністерство освіти і науки України
Рівненський державний гуманітарний університет

Сяський В.А.

ЛОГІЧНЕ ПРОГРАМУВАННЯ

З ПРИКЛАДАМИ ЗАСТОСУВАННЯ

Навчальний посібник

Рівне – 2017

ББК 32.973.2 – 018 я 73
С 99
УДК 004 : 510.755 (075.8)

Рекомендовано до друку Вченою радою Рівненського державного гуманітарного університету, протокол № 1 від 26 січня 2017 року.

Рецензенти:

Турбал Ю.В., доктор технічних наук, професор, Національний університет водного господарства та природокористування, м. Рівне;

Власюк А.П., доктор технічних наук, професор, Міжнародний економіко-гуманітарний університет імені академіка Степана Дем'янчука, м. Рівне.

Сяський В.А. Логічне програмування з прикладами застосування: навчальний посібник / Сяський В.А. – Рівне: РДГУ, 2017. – 96 с.

У посібнику висвітлені визначальні принципи логічного програмування, яке базується на парадигмі декларування даних та мети задачі у форматі предикатів і тверджень. В якості інструментального засобу програмування використана мова PDC Prolog. На основі базових елементів мови та відповідних синтаксичних і семантичних правил розглянуті способи структурування даних, а також різноманітні операції по їх обробці. Для кожного окремого випадку наведені приклади програмної реалізації зі зразками цільових запитів і очікуваних результатів. Посібник розрахований на студентів різних спеціальностей, які ґрунтовно вивчають програмування і в подальшому займатимуться розробкою інтелектуальних інформаційних систем.

ББК 32.973.2 – 018 я 73
С 99
УДК 004 : 510.755 (075.8)

ЗМІСТ

ПЕРЕДМОВА

Практично всі сучасні електронні прилади містять компоненти, які реалізують різноманітні технології штучного інтелекту, зокрема: керування, розпізнавання, прийняття рішень, прогнозування тощо. Особливо актуальним є використання систем штучного інтелекту в стратегічних галузях та для керування складними системами (військова галузь і оборона, національна безпека, ядерні технології, космічна галузь, транспорт та ін.). Для стратегічних глобальних задач використовуються потужні інтелектуальні системи, що базуються на унікальних комп'ютерних комплексах і відповідному програмному забезпеченні. Проте для нескладних задач інтелектуальні системи можна проектувати та реалізовувати на звичайних ЕОМ при застосуванні традиційних мов програмування. Для розробки систем штучного інтелекту часто використовують декларативні мови, до яких відноситься предикатна мова логічного програмування Пролог. Вивчення цієї мови разом з іншими мовами програмування беззаперечно є актуальним при підготовці фахівців з прикладної математики, інформатики, комп'ютерних наук, програмної інженерії.

В основу навчального посібника покладено курс лекцій, який на протязі останніх років читався автором на факультеті математики та інформатики РДГУ для студентів спеціальностей „Прикладна математика", „Інформатика", „Комп'ютерні науки". У посібнику висвітлені визначальні принципи логічного програмування, яке базується на парадигмі декларування всіх даних задачі та цільових запитів мети у форматі тверджень різних предикатів. В якості інструментального засобу програмування пропонується один із діалектів мови Пролог – PDC Prolog, для якої компілятор та інтегроване середовище розроблене компанією Borland Inc. Посібник містить детальні описи основних елементів мови PDC Prolog: домени, предикати, твердження-факти, твердження-правила, динамічні бази даних, цільові запити мети тощо. На основі базових елементів мови та відповідних синтаксичних і семантичних правил представлені різноманітні способи структурування даних у вигляді складених доменів, доменів з альтернативами, списків, рядків символів, динамічних баз даних, файлів та описані різноманітні операції по їх обробці. Особлива увага приділена реалізації повторюваних обчислень на основі механізму відкату і рекурсії. Для всіх ключових елементів мови, а також для основних операцій по обробці даних у посібнику наведені приклади реалізації у вигляді окремих фрагментів коду програм зі зразками цільових запитів і очікуваних результатів. Кожен із прикладів ілюструє можливості мови PDC Prolog та особливості її застосування для вирішення конкретних прикладних задач.

ВСТУП

Сучасний рівень інформаційних технологій є результатом багатоетапного розвитку як апаратних ресурсів ЕОМ (розрядність шини, процесор, пам'ять), так і програмного забезпечення. За більш ніж 70-річну історію практичного застосування електронно-обчислювальної техніки було розроблено і впроваджено багато різноманітних мов програмування: від перших мов машинних кодів до сучасних засобів візуального, мережевого чи web-програмування. Частина мов мали нетривале, вузько окреслене застосування. Інші ж мови, навпаки, – неодноразово трансформувалися, породжуючи все нові діалекти, широко використовуються вже не одне десятиліття до теперішнього часу.

Враховуючи таке багатоманіття мов програмування, їх традиційно класифікують за різними категоріями **(**рівень абстракції, області застосування, підтримувані парадигми програмування). Парадигма програмування – це система ідей і понять, які визначають стиль написання комп'ютерних програм, а також спосіб мислення програміста. Також парадигма програмування – це базові принципи і концепції, що визначають схему обчислень, які виконує комп'ютер. Таким чином, парадигма програмування визначає, як програміст розглядає (уявляє, розуміє) роботу програми. Зрозуміло, що одні і тіж мови програмування можуть підтримувати різні, але не антагоністичні парадигми. Переважна більшість мов програмування може бути віднесена до однієї із двох великих груп мов, які підтримують дві антагоністичні парадигми: **імперативного** програмування та **декларативного** програмування.

Групу **імперативних** мов складають ті мови, програми на яких мають чітко виражений наказовий характер: „**візьми це** (ввід даних), **зроби це** (обробка даних), **віддай це** (вивід даних)**!**". До таких мов відносять мови структурного і процедурного програмування (Fortran, Pascal, C), об'єктно-орієнтовані мови (Object Pascal, Delphi, C++, Java, Pyton), різноманітні системи керування базами даних (Clipper, SQL, Oracle). Ці мови набули масового поширення і популярності завдяки не лише їх функціональним можливостям, а й відносно легкому сприйняттю їх програм людиною (за умови, що ця людина розуміє такі мови).

До **декларативних** мов відносять ті мови, програми на яких не мають чітко вираженого наказового характеру, тобто не є строго визначеною послідовністю інструкцій чи команд, а швидше є сукупністю **декларацій** (описів, оголошень) **властивостей** усіх елементів задачі та **відношень** між ними. Програми таких мов можна охарактеризувати висловом: „**що саме і у якій комбінації слід використати із задекларованого, щоб досягти заданої мети?**". До декларативних мов серед інших відносять мови, які часто використовуються в якості інструменту реалізації систем штучного інтелекту, зокрема, мова

функційного програмування Lisp (розроблена у 1958 році) та предикатна мова логічного програмування Prolog (розроблена у 1972 році).

У галузі досліджень зі штучного інтелекту давно сформувався самостійний напрямок, який одержав назву експертні системи. Мета досліджень з експертних систем полягає в розробці програм, які при вирішенні прикладних задач дають результати, що не поступаються за якістю і ефективністю рішенням, отриманим людиною-експертом у відповідній галузі. Для розв'язування багатьох інтелектуальних задач потрібно здійснювати логічний аналіз предметної області. Автоматичне логічне міркування можливе при наявності формальної мови, на якій можна формулювати твердження і робити вірні логічні висновки. Саме таким інструментальним засобом проектування та реалізації систем штучного інтелекту є мова Prolog (українською – Пролог).

Мова Пролог, що базується на декларативних засадах, дозволила впровадити нові методи програмування, які дозволили згладити відмінності між кроками вирішення задачі і самим перетворенням даних. Програми на Пролозі описують властивості усіх об'єктів задачі, відношення між цими об'єктами та результати, які дадуть ті чи інші дії. При цьому програміст надає самій системі – спеціальним внутрішнім уніфікаційним підпрограмам компілятора (УПП) – розібратися в сукупності способів вирішення задачі.

Прикладом, що ілюструє ефективність декларативного програмування, є класична експертна система „Хімічна лабораторія". Нехай є деякий набір базових хімічних речовин. Також відомий опис можливих хімічних реакцій, причому із різною кількістю вхідних та вихідних реагентів. Потрібно одержати деяку шукану речовину та встановити усі можливі способи її отримання. Очевидно, що пошук рішення має починатися з аналізу набору базових речовин. Якщо результат негативний, то аналізуються окремі хімічні реакції. Якщо і це не дає результат, то аналізуються усі двоетапні, триетапні і т.д. ланцюги реакцій. Якщо на деякому кроці бажана речовина буде отримана, то це буде одним із шуканих розв'язків. Якщо ж перебір усіх можливих варіантів комбінування хімічних реакцій не дасть шуканої речовини, то очевидним є негативний результат задачі.

Для тих, хто звик користуватися імперативними мовами, декларативні мови можуть видатися позбавленими строгості. Проте часто для вирішення конкретної задачі програма на мові Пролог потребує значно менше тверджень, ніж потрібно операторів при використанні імперативних мов. Основна частина програми імперативною мовою призначена для керування ходом обчислень, що обробляють дані. На мові Пролог, навпаки, велика частина керування ходом виконання програми міститься в базових конструкціях мови і автоматично реалізується УПП. Необхідно лише навчитися точно формулювати логічні співвідношення, що описують задачу, а програма сама виконає всі необхідні дії.

РОЗДІЛ 1. ОСНОВИ ЛОГІЧНОГО ПРОГРАМУВАННЯ

1.1. Елементи логіки предикатів та мова Prolog

Для полегшення розуміння особливостей логічного програмування розглянемо простий приклад відношень між об'єктами двох різних типів, які початково декларуються звичайною мовою людського спілкування. Для цього сформулюємо декілька тверджень, які описують відношення вподобання між двома об'єктами виду „*хто* любить *що*":

1) „*Софія* любить *морозиво*";
2) „*Коля* любить *яблука*";
3) „*Коля* любить *морозиво*";
4) „*Оля* любить *тістечка*";
5) „*Андрій* любить *цукерки*";
6) „*Андрій* любить *яблука*";
7) „*Віта* любить *те саме*, що любить *Оля*";
8) „*Таня* любить *те саме*, що любить *Софія*, і любить *Коля*";
9) „*Петро* любить *те саме*, що любить *Оля*, або любить *Андрій*";
10) „*Оксана* любить *те саме*, що любить *Петро*, і не любить *Коля*".

Як видно, частина тверджень (1-6) відрізняється від решти (7-10). Істинність тверджень першої групи є незмінною тобто константною. Такі твердження називають **твердженнями-фактами.** Твердження другої групи не мають постійних значень істинності, їх істинність залежить від значення параметрів цих тверджень. Такі твердження називають **твердженнями-правилами.** Незалежно від різновиду тверджень (факт чи правило) кожне із них обов'язково має особливе слово – **терм** (ідентифікатор), що позначає суть самого відношення. Термом усіх відношень є слово „любить". Це відношення пов'язує два об'єкти: **активний суб'єкт** – це той, хто любить; **пасивний об'єкт** – це те, що люблять. Отож, таке відношення має виражений односторонній характер.

Фактично на основі цих тверджень можна скласти таку таблицю вподобань:

хто любить	любить *що*
Софія	*морозиво*
Коля	*яблука, морозиво*
Оля	*тістечка*
Андрій	*цукерки, яблука*
Віта	*тістечка*
Таня	*морозиво*
Петро	*тістечка, цукерки, яблука*
Оксана	*тістечка, цукерки*

Переформулюємо всі твердження, скориставшись новим форматом тверджень виду відношення(*суб'єкт, об'єкт*):

1) любить(*Софія, морозиво*);
2) любить(*Коля, яблука*);
3) любить(*Коля, морозиво*);
4) любить(*Оля, тістечка*);
5) любить(*Андрій, цукерки*);
6) любить(*Андрій, яблука*);
7) любить(*Віта, щось*), **якщо** любить(*Оля, щось*);
8) любить(*Таня, щось*), **якщо** любить(*Софія, щось*) **і** любить(*Коля, щось*);
9) любить(*Петро, щось*), **якщо** любить(*Оля, щось*) **або** любить(*Андрій, щось*);
10) любить(*Оксана, щось*), **якщо** любить(*Петро, щось*) **і** **не** любить(*Коля, щось*).

Нові твердження мають точно такий самий зміст, що і початкові. Проте усі вони приведені до єдиного спільного формату. У твердженнях-правилах в якості параметра-об'єкта використано займенник *щось*. Саме займенники вказують на предмет, але не називають його. Отже, слово *щось* може позначати що завгодно, по відношенню до чого має місце відношення любить. У мовах програмування аналогом займенника є змінна. Адже змінні можуть набувати будь-яких значень певного типу, а отже позначати що завгодно із цього типу.

А тепер ще раз перепишемо ці твердження, але вже дотримуючись окремих синтаксичних правил мови Prolog:

- усі терми будуються за правилом для ідентифікаторів з букв англійського алфавіту (***A, B, …, Z, a, b, …, z***), символа підкреслення (_), десяткових цифр (***0, 1, …, 9***);
- ідентифікатори констант починаються з малої букви (***a, b, …, z***);
- ідентифікатори змінних починаються з великої букви (***A, B, …, Z***) або символа підкреслення (_);
- кожне окреме твердження завершується крапкою (**.**);
- умовний сполучник **якщо** позначається службовим словом ***if*** або з'єднанням двох символів двокрапка і мінус (***:-***);
- сполучник **і**, що реалізує операцію логічного множення (кон'юнкції), позначається службовим словом ***and*** або символом кома (**,**);
- сполучник **або**, що реалізує операцію логічного додавання (диз'юнкції), позначається службовим словом ***or*** або символом крапка з комою (***;***);
- частка **не**, що реалізує операцію логічного заперечення, позначається службовим словом ***not***.

Не здійснюючи точного перекладу на англійську мову всіх елементів, а лише терму любить, нові твердження можна записати так:

clauses
likes(sofiya, morozyvo).
likes(kolya, yabluka).
likes(kolya, morozyvo).
likes(olya, tistechka).
likes(andriy, cukerky).
likes(andriy, yabluka).
likes(vita, X) :- likes(olya, X).
likes(tanya, X) :- likes(sofiya, X), likes(kolya, X).
likes(petro, X) :- likes(olya, X); likes(andriy, X).
likes(oksana, X) :- likes(petro, X), not(likes(kolya, X)).

Тут ***clauses*** – службове слово мови Prolog, заголовок розділу оголошення тверджень.

Тепер сформулюємо декілька запитань. При цьому спочатку запишемо кожне із них звичайною мовою людського спілкування (українською), а потім – мовою програмування Prolog. Відповіді на питання подаватимуться значеннями змінних або логічними значеннями істинності чи хибності (*Yes, No*):

1) „Що любить Коля?"
likes(kolya, X).

X=yabluka; X=morozyvo *(2 розв'язки)*

2) „Хто любить морозиво?"
likes(X, morozyvo).

X=sofiya; X=kolya; X=tanya *(3 розв'язки)*

3) „Чи любить Петро яблука?"
likes(petro, yabluka).

Yes

4) „Чи любить Оксана яблука?"
likes(oksana, yabluka).

No

5) „Чи любить Віта щось?"
likes(vita, _).

Yes

6) „Чи хтось любить молоко?"

likes(_, moloko).

No

7) „Чи взагалі хтось щось любить?”

likes(_, _).

Yes

У питаннях 5-7 в якості параметра використовується особлива змінна, яка називається **анонімною змінною** і позначається символом підкреслення (_). Така змінна передбачає довільне значення, що забезпечує істинність твердження.

З точки зору логічного програмування всі запитання називають **цільовими запитами** або просто **метою**. Як видно з прикладів цільових запитів, структура самих запитів аналогічна структурі базових тверджень відношення (*суб’єкт, об’єкт*). Відмінність у різних запитах реалізовувалась шляхом використання константних значень або змінних, або анонімної змінної в якості того чи іншого параметра. При цьому зміст усіх базових тверджень не змінювався. Можна говорити, що ці твердження утворюють так звану статичну базу даних.

Усі базові твердження і запити мають однакову структуру, а точніше, запити – це такі самі твердження. Тому всі твердження разом (базові та запити) відносяться до єдиного спільного для них предиката. **Предикат** у мові Prolog – це шаблон тверджень. Він визначає структуру подібних між собою тверджень або представляє сукупність таких тверджень. Подібність тверджень тут розуміється в сенсі спільного відношення та однакового набору формальних параметрів-аргументів (кількість, їх типи та порядок).

У розглянутому прикладі термом відношення є *likes*; відношення має два параметри-аргументи однакового базового типу, який у мові Prolog має назву **символічні імена** (ідентифікатори) і є стандартним доменом ***symbol***. Згідно із логікою задачі перший параметр умовно вважається активним по відношенню до другого. Виходячи з цього предикат для твердження *likes* може бути оголошений так:

predicates

likes(symbol, symbol)

Тут ***predicates*** – службове слово мови Prolog, заголовок розділу оголошення предикатів.

1.2. Базові елементи мови PDC Prolog. Структура програми

Prolog, як і всі мови програмування, відноситься до так званих формальних мов. Такі мови передбачають строге визначення за допомогою синтаксичних та семантичних правил для кожного із елементів, у тому числі і для програми в цілому.

- **Алфавіт** мови Prolog складається з трьох груп символів:

➢ **букви** – це великі та малі англійські букви, символ підкреслення ***A, B, C, …, Z, a, b, c, ..., z, _*** ;

➢ **цифри** – це десяткові цифри ***0, 1, 2, …, 9*** ;

➢ **знаки** – це символи **+ - * / < > = . , : ; ( ) [ ] ‘ “ $ | !**

- **Програма** в мові Prolog складається із декількох розділів, у яких декларуються різні за змістом об’єкти і відношення. Частина з розділів є обов’язковими, а окремі використовуються лише за необхідності. Порядок розділів не принциповий – головне, щоб були оголошені усі нестандартні (користувацькі) компоненти програми. Не обмежуючи загальності, розглянемо обов’язкові розділи та ті з необов’язкових, що часто використовуються.

- **Розділ оголошення доменів (доменних типів).** Він є необов’язковим і використовується лише тоді, коли в програмі застосовуються нестандартні типи доменнів. Цей розділ починається службовим словом ***domains.*** Усі користувацькі домени декларуються згідно із синтаксичним правилом:

терм_домену = опис_домену

Опис домену передбачає оголошення за певними правилами на основі або стандартних, або інших оголошених доменів. Якщо використовується декілька однакових за типом, але з різними іменами доменів, то їх терми перелічуються через кому (**,**). Порядок декларації доменів може бути довільним. Головне, щоб вони були здійснені в межах одного розділу оголошення доменів. Наприклад:

```
domains
    cili = integer                  /* тотожний домену integer */
    spysok_real = real*             /* список дійсних чисел */
    knyga = kn(avtor, nazva, rik)   /* складений домен із трьох */
    avtor, nazva = string           /* елементів різних доменів, */
    rik = integer                   /* що оголошуються пізніше */
```

У мові Prolog використовується розгалужена система стандартних скалярних та складених доменів. Серед них виділимо шість доменів, що мають загальне використання.

домен	терм	розмір у пам'яті	діапазон значень, приклади
цілі числа із знаком	***integer***	2 байти	-32768 .. 32767 -32768 = $8000 32767 = $7ffff
дійсні числа	***real***	8 байти	≈1.2E-308 .. ≈1.8E+308 123.456 = 123456E-03 123456 = 1.23456e+05
символи	***char***	1 байт	'A' = '\65' '#' = '\35' '7' = '\55'
рядки символів	***string***	≤64 кБ	"A#7" = "\65\35\55" "" – порожній рядок
символічні імена	***symbol***	рядки	"A#7", "Prolog"
		ідентифікатори ≤250 символів	abc, ABC123, _123
файли	***file***	логічні файлові імена	screeen, keyboard, printer, stdin, stdout, com1, f1, f2

- цілі числа із знаком можна подавати десятковими або шістнадцятковими цифрами. Числа поза межами діапазону значень не вважаються цілими, а є дійсними;
- дійсні числа подвоєної точності можна подавати з фіксованою або з плаваючою десятковою крапкою. До дійсних відносять усі великі за абсолютною величиною цілі числа, що не потрапляють у діапазон допустимих значень;
- символ ASCII, обмежений парою апострофів (**'**), можна подавати явно або через цілочисельний код;
- рядки символів, обмежені парою лапок (**"**), можна подавати явно або через цілочисельні коди;
- символічні імена об'єднують два класи об'єктів: довільна послідовність символів, обмежена парою лапок (**"**), – рядки символів; ідентифікатори – так звані імена мови Prolog;
- файли – це логічні файлові імена, які представляють у програмі стандартні файлові пристрої вводу-виводу, периферійні файлові пристрої та дискові файли.

- **Розділ оголошення предикатів.** Він є обов'язковим, і в ньому декларуються предикати для усіх тверджень статичної бази даних програми (СБД). Цей розділ починається службовим словом ***predicates***. Усі предикати декларуються згідно із синтаксичним правилом:

терм_предиката [(домен [, домен [, домен, ...]])]

Кількість, порядок та тип доменів визначає набір формальних параметрів-аргументів предиката. У кожній парі квадратних дужок ***[ ]*** позначені необов'язкові елементи. Отже, предикати можуть мати довільну кількість параметрів, навіть можуть бути без параметрів. Кількість параметрів визначає так звану **арність** (ступінь) відношення. Наприклад:

```
predicates
    go                          /* предикат без параметрів */
    kolir(string)               /* предикат арності 1 */
    likes(symbol, symbol)       /* предикат арності 2 */
    suma(real, real, real)      /* предикат арності 3 */
```

- **Розділ оголошення предикатів динамічної бази даних (ДБД).** Він є необов'язковим і використовується лише тоді, коли в програмі виконується обробка тверджень ДБД. Цей розділ починається службовим словом ***database*.** Предикати ДБД декларуються за таким самим синтаксичним правилом, як і предикати у розділі ***predicates***. Хоча компілятор мови Prolog розпізнає різні за змістом, але з однаковими термами елементи програми за їх контекстом входження у твердження, проте доцільно для предикатів СБД і ДБД використовувати унікальні ідентифікатори. Наприклад:

```
database
    d_go                        /* предикат без параметрів */
    d_kolir(string)             /* предикат арності 1 */
    d_likes(symbol, symbol)     /* предикат арності 2 */
    d_suma(real, real, real)    /* предикат арності 3 */
```

- **Розділ оголошення тверджень програми.** Він є обов'язковим, і в ньому декларуються усі твердження для кожного із предикатів СБД програми. Цей розділ починається службовим словом ***clauses***. Для кожного твердження має бути оголошений свій предикат. Один предикат може представляти декілька подібних тверджень. У свою чергу для кожного предиката має бути хоча б одне твердження. Порядок декларації тверджень (факти чи правила) може бути довільним, але у багатьох випадках їх послідовність визначається алгоритмом розв'язку задачі. Головне, щоб усі твердження одного предиката були згруповані в одному блоці. Наприклад:

```
clauses
    go.                         /* твердження-факт */
    go :- write("Hello!"), nl.  /* твердження-правило */
    likes(olya, tistechka).     /* твердження-факт */
    likes(vita, X) :- likes(olya, X).   /* твердження-правило */
    d_suma(X, Y, Z) :- Z = X + Y.       /* твердження-правило */
```

- **Розділ цільового запиту або мети.** Він є необов'язковим і використовується лише тоді, коли програма виконується у режимі так

званої **внутрішньої мети**. Цей розділ починається службовим словом ***goal.*** У розділі мети формулюється твердження цільового запиту. У випадку внутрішньої мети спеціальні уніфікаційні підпрограми компілятора (УПП) „знаходять" перший по порядку із можливих розв'язків. При цьому вільні змінні, що використовуються у запиті, набувають певних значень. Значення таких змінних слід виводити за допомогою стандартних тверджень виводу ***write.*** Якщо ж розділ цільової мети ***goal*** відсутній, то програма виконується у режимі так званої **зовнішньої мети**. Тоді твердження цільового запиту формулюються в окремому діалоговому вікні. У випадку зовнішньої мети УПП „обчислюють" усі можливі варіанти розв'язків, і значення усіх змінних, що набуваються в процесі обчислення, виводяться автоматично. Наприклад:

```
goal                                    /* внутрішня мета: */
    likes(X, Y), write(X, "любить", Y), nl.    /* "хто що любить?" */
```

У результаті буде встановлено перший по порядку розв'язок:

```
    sofiya любить morozyvo.
```

Зауваження. Для ефективного ілюстрування прикладів цільових запитів у випадку зовнішньої мети домовимося перед самим твердженням мети вказувати слово ***GOAL*** з великих букв та відділяти символом двокрапка (***:***). Тоді варіант запиту у випадку зовнішньої мети може мати вигляд:

```
GOAL:                                   /* зовнішня мета: */
likes(X, yabluka)                       /* "хто любить яблука?" */
-------------------------
X=kolya
X=andriy
X=petro
```

У результаті було встановлено всі три розв'язки. Тут і в подальшому для зручності відокремлюватимемо твердження зовнішньої мети від результату горизонтальною лінією.

1.3. Обчислення мети

Цільовий запит, що сформульований у внутрішній чи зовнішній меті, обчислюється УПП шляхом співставлення твердження самої мети із твердженнями з розділу ***clauses***. Звичайно, що твердження мети може мати змінні. У такому випадку ці змінні є **вільними змінними**. Поняття вільної змінної означає, що ця змінна не має жодного значення, навіть нейтрального для даного типу. Тоді УПП у процесі обчислення нададуть усім вільним змінним тих значень, які забезпечать істинність мети.

У мові Prolog не використовується попереднє оголошення типу змінних. Їх тип визначається у процесі обчислення за контекстом застосування. Наприклад:

```
A = 3,                  /* A – integer, бо 3 – integer */
B = A,                  /* B – integer, бо A – вже integer */
likes(C, morozyvo),     /* C – symbol через оголошення likes */
readreal(D).            /* D – real, бо readreal повертає real */
```

Процес обчислення мети, що базується на співставленні тверджень, здійснюється так:

- УПП „вибирають” перше за порядком твердження СБД, терм якого відповідає терму цільового запиту, а на черговому за порядком альтернативному твердженні „ставлять” так звану **точку відкату**;
- виконується співставлення фактичних параметрів мети та твердження СБД:
  - якщо чергова пара параметрів є константами і вони співпадають, то здійснюється перехід до наступної пари параметрів і т.д. Якщо ж деяка пара відповідних параметрів не є тотожньою, то поточне твердження „відкидається” і здійснюється **відкат** (повернення) в останню точку відкату до наступного твердження СБД, терм якого відповідає терму цільового запиту. При цьому на черговому за порядком альтернативному твердженні ставиться нова точка відкату;
  - якщо чергова пара параметрів передбачає співставлення константи і вільної змінної або змінної зі значенням і вільної змінної, то вільна змінна набуває відповідного значення, і здійснюється перехід до наступної пари параметрів і т.д.;
  - якщо чергова пара параметрів передбачає співставлення константи та змінної зі значенням або двох змінних зі значеннями, то виконується порівняння обох значень. При рівності здійснюється перехід до наступної пари параметрів, а при нерівності відбувається відкат в останню точку відкату до іншого твердження, яке відповідає цільовому запиту. При такому відкаті всі змінні, які отримали значення на останньому етапі співставлень, знову звільняються;

- процес співставлення параметрів завершується, коли або співпадуть усі пари фактичних параметрів, або після чергового неспівпадання параметрів вже вичерпані альтернативні твердження СБД для виконання відкату. У першому випадку цільовий запит є успішним і всі вільні змінні мети отримали ті значення, які визначають черговий варіант розв'язку. У другому випадку цільовий запит є хибним: розв'язок не встановлено, всі змінні мети залишаються вільними.

Якщо цільовий запит є складеним твердженням, тобто передбачає обчислення деякого правила, то можна говорити про комбінацію нових підцілей. Кожна така підціль – це новий запит, який обчислюється за таким самим алгоритмом. У кожній із підцілей є свої внутрішні вільні змінні. Звичайно, що формально одноіменні змінні на різних рівнях вкладення підцілей фактично є різними і не конфліктують. При багаторівневому вкладенні підцілей відкати для вибору альтернативних тверджень виконується від найнижчого рівня вкладень до вищих.

Зауваження. Використання змінних, які зв'язуються певними операціями, потребує уваги і контролю за коректністю програми. Зокрема це стосується тверджень, що реалізують відношення з операцією еквівалентності, яка позначається символом рівності (=). Операція еквівалентності = може мати різний зміст у залежності від того, які змінні чи константи вона зв'язує:

- якщо у твердженні операція еквівалентності = зв'язує вільну змінну та константу або вільну змінну та іншу змінну зі значенням, то операція = розуміється як операція присвоєння деякого значення вільній змінній. Наприклад:

```
GOAL:
X=3, X=Y                  /* Y – вільна змінна, а X – зі значенням */
-------------------------  /* напрям присвоєння – довільний */
X=3, Y=3
```

- якщо у твердженні операція еквівалентності = зв'язує змінну зі значенням та константу або дві змінні зі значеннями, то операція = розуміється як операція порівняння на рівність з істинним або хибним результатом. Наприклад:

```
GOAL:
X=3, Y=5, Y=X             /* X, Y – змінні з різними значеннями */
-------------------------
No
```

1.4. Складені правила

Основу програми на мові Prolog складають твердження. Ті твердження, що описують деяку задачу (формалізують її), утворюють так звану статичну базу даних (СБД). Цільові запити мети – це теж твердження, тільки у формі запитань. Усі твердження можуть бути або у вигляді фактів, або у вигляді правил.

Правила у мові Prolog – це різновид тверджень, істинність яких не має постійного значення і залежить від фактичних параметрів-аргументів. Правила можна розуміти як логічнозначні функції з довільною кількістю аргументів різних доменів. Загальна структура твердження-правила:

голова_правила :- тіло_правила.

Голова та тіло правила – це твердження. Істинність голови визначається істинністю тіла правила. Отже, обчислення голови передбачає обчислення тіла правила. Фактичні параметри голови передаються в тіло і навпаки – після обчислення тіла повертаються в голову. Тіло правила складається із тверджень, що пов'язуються за допомогою логічних операцій кон'юнкції, диз'юнкції, заперечення.

- **Кон'юнкція тверджень** передбачає послідовне обчислення таких тверджень. Результат усієї кон'юнкції залежить від наявності хоча б одного хибного твердження:
  - якщо всі підцілі істинні, то остаточний результат правила з кон'юнкцією є істинним. При цьому всі вільні змінні отримують такі значення, які забезпечують істинність результату;
  - якщо чергова підціль хибна, то процес обчислень обривається і всі наступні підцілі ігноруються. У цьому випадку остаточний результат правила з кон'юнкцією є хибним. Всі змінні, що отримали якісь значення при обчисленнях попередніх підцілей, звільняються. Хибність поточної підцілі встановлюється лише після того, як УПП виконали всі можливі відкати з метою вибору альтернативних тверджень для її обчислення. Більш того, відкати виконуються і для попередніх підцілей на даному рівні вкладення тверджень, якщо вони повертали в якості фактичних параметрів змінні із наченнями, що потім передавалися в наступні підцілі.

Якщо всі твердження складеного правила з кон'юнкцією мають істинне значення, то послідовне обчислення цих тверджень можна вважати аналогом лінійної послідовності операторів, яку в імперативних мовах прийнято називати складеним оператором або операторо-блоком:

мова Pascal	мови C/C++	мова Prolog
begin *operator_1;* *operator_2;* *...* *operator_n* *end*	*{* *operator_1;* *operator_2;* *...* *operator_n;* *}*	*rule :- clause_1, clause_2, ..., clause_n.*

- **Диз'юнкція тверджень** передбачає або послідовне обчислення всіх тверджень, якщо вони хибні, або обчислення до першого за порядком істинного твердження. Результат усієї диз'юнкції залежить від наявності хоча б одного істинного твердження:
  - якщо всі підцілі хибні, то остаточний результат правила з диз'юнкцією є хибним. Хибність кожної підцілі встановлюється лише після того, як УПП виконали всі можливі відкати з метою вибору альтернативних тверджень для їх обчислення;
  - якщо чергова підціль істинна, то процес обчислень обривається і всі наступні підцілі ігноруються. У цьому випадку остаточний результат правила з диз'юнкцією є істинним. При цьому всі вільні змінні отримують такі значення, які забезпечують істинність результату.

Оскільки для складеного правила з диз'юнкцією достатньо істинності одного з альтернативних тверджень, то операцію диз'юнкції можна використовувати для реалізації алгоритмічних обчислень із розгалуженнями. При цьому кожній з альтернатив повинна передувати її **умова застосування** – булевий вираз. Це свого роду логічний фільтр, який набуває істинності лише для однієї з альтернативних віток і унеможливлює виконання інших. Складені правила з диз'юнкцією альтернативних тверджень можна вважати аналогом умовних операторів, операторів вибору варіанту чи операторів-перемикачів у імперативних мовах:

мова Pascal	мови C/C++	мова Prolog
if bool_expr *then operator_1* *else operator_2*	*if (bool_expr)* *operator_1;* *else operator_2;*	*rule_if :- bool_expr, clause_1;* *not(bool_expr), clause_2.*

В якості прикладу пропонується твердження-правило, яке в залежності від знаку цілого числа виводить одне із повідомлень „додатнє", „від'ємне", „нуль":

```
predicates
    znak(integer)
clauses
    znak(X) :- X > 0, write("додатнє"), nl;
               X < 0, write("від'ємне"), nl;
               X = 0, write("нуль"), nl.
```

Зауваження. Обчислення правил із диз'юнкцією здійснюється дещо відмінним чином у різних випадках цільової мети програми. Якщо мета внутрішня, то УПП шукають лише один перший за порядком із альтернативних розв'язків. У випадку, коли операція диз'юнкції поєднує декілька істинних тверджень, виконається лише перше. Якшо ж мета зовнішня, то УПП шукають усі можливі варіанти розв'язків. У випадку, коли операція диз'юнкції поєднує декілька істинних тверджень, послідовно виконаються всі альтернативи. Тому для однозначності обчислень із розгалуженнями доцільно застосовувати для кожної з альтернативних віток свою умову застосування.

У мові Prolog є сособливе стандартне твердження, яке гарантовано забезпечує навіть у випадку зовнішньої мети виконання лише одного з декількох істинних тверджень, що поєднані операцією диз'юнкції. Це твердження „руйнує" **точки відкату**, які при виборі чергового із тверджень, що відповідає меті, автоматично встановлюються на кожній наступній альтернативній вітці обчислення розв'язку. Оскільки точок відкату немає, то УПП не можуть розпочати обчислення чергового альтернативного варіанту розв'язку. Це твердження називається **відсіканням** і позначається словом ***cut*** або еквівалентним йому символом – знаком оклику (***!***). Відсікання ***!*** традиційно використовується відразу після умови застосування відповідного альтернативого твердження.

Таким чином, алгоритмічна конструкція ***if then else*** на мові Prolog може бути реалізована правилом:

rule_if :- bool_expr,!, clause_1; clause_2.

Звичайною мовою людського спілкування це можна тлумачити так: „якщо умова *bool_expr* істинна, то виконується **виключно** твердження *clause_1*, інакше – виконується твердження *clause_2*".

Якщо диз'юнкція передбачає багатовіткове розгалуження, то в усіх альтернативних вітках окрім останньої доцільно використовувати свої умови застосування з відсіканням ***!***, а остання альтернатива може бути взагалі без умови. Попередній приклад правила, що визначає знак цілого числа, можна записати так:

```
znak(X) :- X > 0,! write("додатнє"), nl;
           X < 0,! write("від'ємне"), nl;
                   write("нуль"), nl.
```

- **Заперечення *not*** змінює результат істинності тверджень на протилежний. При цьому використання заперечення потребує дотримання ряду вимог:
  - заперечення ***not* не можна** застосовувати до тверджень із фактичними параметрами, що є вільними змінними. Змінна, що є параметром твердження, може отримати значення лише у випадку його істинності. Оскільки будь-яка мета передбачає обчислення з остаточним істинним результатом, то твердження внутрішньої підцілі, до якого застосовується заперечення, мало б обчислюватись із остаточним хибним результатом. А це суперечить загальній парадигмі логічного програмування. Наприклад:

    GOAL:
    not(likes(X, yabluka)) */* ПОМИЛКА!!! */*
    ---------- */* не можна встановити, */*
    error */* хто не любить яблука */*

  - заперечення ***not* можна** застосовувати до тверджень із попередньо визначеними фактичними параметрами. У цьому випадку перевіряється істинність чи хибність такого твердження при зафіксованих значеннях параметрів. Наприклад:

    GOAL:
    not(likes(sofiya, yabluka)) */* проте можна перевірити, */*
    ---------- */* чи Софія не любить яблука */*
    Yes

Зауваження. Щоб у останньому прикладі цільовий запит дійсно мав наведений у коментарі зміст „чи Софія не любить яблука?”, необхідно розуміти **відсутність хоча б одного твердження** виду „*хтось конкретний* любить *щось конкретне*” **еквівалентною твердженню** „*хтось конкретний* не любить *щось конкретне*”. Точно відомо, що „*Софія* любить *морозиво*”, а про відношення Софії до усього іншого нічого не відомо. Тому в силу прийнятої еквівалентності це можна тлумачити як „*Софія* не любить *усе інше, зокрема і яблука*”.

- **Композиція логічних операцій** дозволяє конструювати правила довільної складності. Проте у мові Prolog є свої особливості композиції операцій кон’юнкції та диз’юнкції. Оскільки відсутня операція підвищення пріоритету тверджень ***( )***, то властивість дистрибутивності або розподільний закон у записі

  (clause_1 or clause_2) and clause_3

не дасть бажаного результату виду

  (clause_1 and clause_3) or (clause_2 and clause_3).

Вирішити таку проблему можна шляхом використання додаткового правила, яке окремо реалізує диз’юнкцію перших двох тверджень:

rule_or :- clause_1; clause_2.

rule :- rule_or, clause_3.

Додаткові твердження, що підвищують пріоритет операцій, дозволять конструювати як завгодно складні композиції тверджень. Якщо знову проводити аналогію з імперативними мовами програмування, то такі твердження подібні до використання підпрограм-процедур чи підпрограм-функцій.

Розглянемо приклад цілісної програми, що реалізує задачу побудови родинного дерева. В якості базових тверджень про родинні зв'язки використаємо наступні твердження-факти:

чоловіча_стать(ім'я);

жіноча_стать(ім'я);

нащадок(ім'я нащадка, ім'я предка).

Потрібно побудувати похідні твердження-правила, що реалізують наступні відношення:

тато(ім'я, ім'я); мама(ім'я, ім'я);
дід(ім'я, ім'я); баба(ім'я, ім'я);
брат(ім'я, ім'я); сестра(ім'я, ім'я);
дядя(ім'я, ім'я); тьотя(ім'я, ім'я);
кузен(ім'я, ім'я); кузина(ім'я, ім'я).

Програмна реалізація може мати вигляд:

```
domains
    s=symbol
predicates
    chol(s)              gin(s)
    nashch(s, s)         tato(s, s)        mama(s, s)        batki(s, s)
    did(s, s)            baba(s, s)        brat(s, s)        sestra(s, s)
    dyadya(s, s)         tyotya(s, s)      kuzen(s, s)       kuzyna(s, s)
clauses
    chol(did_t).         chol(did_m).      ...      chol(kuzen_4).
    gin(baba_t).         gin(baba_m).      ...      gin(kuzyna_4).
    nashch(tato, did_t).          nashch(tato, baba_t).
    nashch(mama, did_m).          nashch(mama, baba_m).
                ...       ...     ...
                ...       ...     nashch(kuzyna_4, tyotya_m).
    tato(X, Y) :- chol(X), nashch(Y, X).
    mama(X, Y) :- gin(X), nashch(Y, X).
    batki(X, Y) :- tato(X, Y); mama(X, Y).
    brat(X, Y) :- chol(X), tato(T, X), tato(T, Y), mama(M, X), mama(M, Y),
                  X<>Y.
    sestra(X, Y):-gin(X), tato(T, X), tato(T, Y), mama(M, X), mama(M, Y),
                  X<>Y.
    did(X, Y) :- tato(X, Z), batki(Z, Y).
```

```
baba(X, Y) :- mama(X, Z), batki(Z, Y).
dyadya(X, Y) :- brat(X, Z), batki(Z, Y).
tyotya(X, Y) :- sestra(X, Z), batki(Z, Y).
kuzen(X, Y) :- chol(X), tato(Z, X), dyadya(Z, Y);
               chol(X), mama(Z, X), tyotya(Z, Y).
kuzyna(X, Y) :- gin(X), tato(Z, X), dyadya(Z, Y);
                gin(X), mama(Z, X), tyotya(Z, Y).
```

1.5. Структури – складені домени

У багатьох випадках програмування передбачає обробку структур даних, які мають складену будову. При цьому структурованість може передбачати об'єднання в одне ціле декількох елементів даних різних за типом. Наприклад: книга – це сукупність деяких елементів даних (автор – рядок символів, назва – рядок символів, рік видання – ціле число, ціна – дійсне число тощо); студент – це сукупність інших елементів даних (ім'я, рік народження, форма навчання (державна, платна), середня успішність тощо); автомобіль – це сукупність ще інших елементів даних (марка, модель, рік випуску, об'єм двигуна, державний реєстраційний номер, колір тощо).

В інших випадках обробки неструктурованої інформації, що є іменами деяких образів, виникає потреба додаткового уточнення одноіменних, але різних за змістом, образів. Наприклад: англійське слово *orange* може позначати або фрукт, або напій, або колір.

У мові Prolog для об'єднання декількох різнотипних компонентів у одну структуру або для уточнення атомарних елементів даних використовують особливі користувацькі домени, що називають **структурами** або **складеними доменами**. Синтаксичне правило оголошення складеного домену наступне:

терм_домену = функтор [(домен [, домен [, домен, ...]])]

Як і раніше, у кожній парі квадратних дужок ***[ ]*** позначені необов'язкові елементи. Отже, структура може містити довільну кількість елементів різних доменних типів. Терм домену – це ідентифікатор нового домену, який в подальшому буде використовуватись для оголошення типу параметрів предикатів.

Функтор – особливий терм, який фактично є іменем структури (подібно до мов C/C++). Він виконує або об'єднуючу функцію, коли складений домен має декілька компонентів, або уточнюючу функцію, коли домен має одну компоненту. Функтор може співпадати з іменем самого домену. УПП розрізнятимуть такі одноіменні об'єкти за контекстом їх використання. Якщо функтор не має параметрів, то говорять про порожню структуру. Така структура однаково має функтор, і цей функтор може бути фактичним параметром тверджень.

Синтаксичні правила оголошення складених доменів і предикатів мають багато спільного. Тому можна говорити про подібність, а точніше – про спільність цих об'єктів. Адже структура – це теж відношення між декількома елементами – відношення об'єднання або уточнення. Часто замість терму предиката говорять про функтор предиката.

Складені домени можуть бути використані у твердженнях, для яких звичайно мають бути оголошені предикати. Наприклад:

```
domains
    knyga = knyga(avtor, nazva, rik, cina)
    student = st(imya, rik, forma, usp)
    avtor, nazva, imya = string
    rik = integer
    cina, usp, stypendiya = real
    forma = char
    napiy = n(symbol)
    frukt = fr(symbol)
    colir = col(symbol)
predicates
    vlasnyk(symbol, knyga)
    stypendiya(student, real)
    likes_1(symbol, napiy)
    likes_2(symbol, frukt)
    likes_3(symbol, colir)
clauses
    vlasnyk(kolya, knyga("Народ", "Колобок", 2016, 1234.5)).
    vlasnyk(sofiya, knyga("Дж.К.Ролінг","Гаррі Поттер", 2010, 100)).
    stypendiya(st("Коля", 1997, 'd', 4.25), 1234.5).
    stypendiya(st("Софія", 1997, 'p', 4.5), 0).
    likes_1(kolya, n(orange)).
    likes_1(sofiya, n(coffee)).
    likes_2(kolya, fr(orange)).
    likes_2(sofiya, fr(banana)).
    likes_3(kolya, col(orange)).
    likes_3(sofiya, col(green)).
```

У розглянутому прикладі складений домен *knyga* має одноіменний функтор *knyga*, а складені домени *student*, *napiy*, *frukt*, *colir* мають відмінні функтори *st, n, fr, col*. Для відношення вподобання різних речей (напій, фрукт, колір) використовуються різні предикати *likes_1, likes_2, likes_3*. Предикат і твердження *stypendiya* мають другий параметр, що формально позначений таким самим термом *stypendiya*. УПП розрізнятимуть їх за контекстом використання.

Для задекларованих предикатів і тверджень можна сформулювати такі цільові запити:

GOAL:
vlasnyk(X, knyga(_, ”Колобок”, _, _)) / хто має книгу „Колобок”? */*

X = kolya

vlasnyk(sofiya, X) / які книги має Софія? */*

X = knyga(“Дж.К.Ролінг”, “Гаррі Поттер”, 2010, 100)

stypendiya(st(X, _, _, _), Y), Y > 0 / хто має стипендію, */*
--- */* який розмір стипедії? */*
X = “Коля”, Y = 1234.5

likes_1(kolya, X) / який напій любить Коля? */*

X= n(orange)

likes_2(X, fr(banana)) / хто любить банани? */*

X= sofiya

likes_3(X, col(blue)) / хто любить синій колір? */*

No

Як видно з прикладів, вільні змінні в запитах можуть представляти або цілу структуру, або окремі її елементи. Це залежить від того, чи використовується функтор структури на місці відповідного параметра твердження. Взагалі функтор виконує або об’єднуючу, або розділяючу функцію в залежності від того, який зміст мають його параметри:

- якщо функтор застосовують до заданої структури, а його параметри є вільними змінними, то він виконує функцію розділення структури на складові:
  GOAL:
  X = knyga(“Дж.К.Ролінг”, “Гаррі Поттер”, 2010, 100),
  knyga(A, N, R, C) = X

  A = “Дж.К.Ролінг”, N = “Гаррі Поттер”, R = 2010, C = 100

- якщо функтор застосовують до набору фактичних параметрів, які є константами або змінними із значеннями, то він виконує функцію об’єднання їх в єдину складену структуру:
  GOAL:
  A = “Дж.К.Ролінг”, N = “Гаррі Поттер”, R = 2010, C = 100,

X = knyga(A, N, R, C)

X = knyga("Дж.К.Ролінг", "Гаррі Поттер", 2010, 100)

1.6. Структури з альтернативами

У попередніх прикладах мала місце алгоритмічна проблема, що вимагала використання трьох різних предикатів для тверджень про спільне відношення вподобання різних речей: *likes_1* – це відношення лише для напоїв, *likes_2* – для фруктів, *likes_3* – для кольорів. Використання трьох різних предикатів частково вирішує цю проблему, але потребує збільшення програми і ускладнює цільові запити:

GOAL: */* що взагалі любить Коля? */*

likes_1(kolya, X); likes_2(kolya, X); likes_3(kolya, X)

X = n(orange))

X = fr(orange))

X = col(orange))

Як видно із прикладу, проблема вирішена шляхом використання диз'юнкції різних тверджень. Ця ж операція також дозволить уніфікувати декілька подібних тверджень в одне спільне, якщо використовувати особливий різновид складених доменів – **домени з альтернативами**.

Оголошення доменів з альтернативами здійснюється відповідно до синтаксичного правила:

терм_домену = функтор_1 [(домен [, домен [, домен, ...]])];
функтор_2 [(домен [, домен [, домен, ...]])];
...
функтор_n [(домен [, домен [, домен, ...]])]

Кожна з альтернатив – це довільна структура зі своїм функтором. Їх кількість і порядок можуть бути довільними. Якщо знову проводити аналогію з мовами C/C++, то структури з альтернативами подібні до об'єднань (union). У твердженнях тих предикатів, які пов'язують відношеннями складені домени з альтернативами, можуть використовуватися довільні з альтернатив.

В якості прикладу розглянемо твердження про власників різних речей, які можуть бути одним із:

- *книга(автор, назва, рік_видання);*
- *транспорт(вид, модель, рік_випуску, колір);*
- *колекція(вид, кількість_предметів).*

Відповідні декларації можуть мати вигляд:

```
domains
    rich = knyga(avtor, nazva, rik);
           transp(vyd, model, rik, colir);
           colekc(vyd, kilkist)
    avtor, nazva, model = string
    vyd, colir = symbol
    rik, kilkist = integer
predicates
    vlasnyk(symbol, rich)
clauses
    vlasnyk(kolya, knyga("Народ", "Колобок", 2016)).
    vlasnyk(kolya, transp(velo, "Спорт-Трек", 2016, silver)).
    vlasnyk(sofiya, knyga("Дж.К.Ролінг", "Гаррі Поттер", 2010)).
    vlasnyk(sofiya, transp(avto, "Fiat 500", 2012, red)).
    vlasnyk(kolya, colekc(znachky, 500)).
    vlasnyk(sofiya, colekc(marky, 1000)).
```

Для задекларованих предикатів і тверджень можна сформулювати такі цільові запити:

```
GOAL:
vlasnyk(kolya, X))                    /* чим володіє Коля? */
-------------------------
X = knyga("Народ", "Колобок", 2016)
X = transp(velo, "Спорт-Трек", 2016, silver)
X = colekc(znachky, 500)

vlasnyk(X, knyga(A, N, R)))           /* хто має книги і які саме? */
-------------------------
X = kolya, A = "Народ", N = "Колобок", R = 2016
X = sofiya, A = "Дж.К.Ролінг", N = "Гаррі Поттер", R = 2010
```

РОЗДІЛ 2. ОБЧИСЛЕННЯ З ПОВТОРЕННЯМ

У структурному програмуванні основними способами реалізації обчислень із повторенням є ітерація та рекурсія. Ітерація в більшості випадків реалізується за допомогою операторів циклів. Такий процес обчислень можна охарактеризувати висловом: „**виконувати для всіх значень, що задовольняють умові**”. Це означає, що оператори циклу передбачають зміну значень деяких змінних (параметрів циклу), які визначають умову продовження повторень.

У мові Prolog цикли в традиційному розумінні застосовувати неможливо, оскільки операція переприсвоєння значення змінної взагалі є неуспішною. Адже отримати значення може лише вільна змінна. Якщо ж змінна вже має значення, то спроба переприсвоєння розуміється як порівняння старого значення із іншим відмінним значенням. А це завжди хибне твердження.

У логічному програмуванні повторювані обчислення виконуються за допомогою правил, які використовують механізм **відкату** або **рекурсію**.

2.1. Відкат як спосіб реалізації обчислень з повторенням

Відкат є основою ітеративних схем повторюваних обчислень. УПП у поточний момент обчислень „аналізують” деяке твердження СБД, а на наступному альтернативному твердженні, що відповідає меті, встановлюється точка відкату. Якщо якимось чином генерується хибний результат, то всі змінні, що отримали значення в процесі попередніх обчислень, звільняються, і виконується відкат до останньої точки відкату. Таким чином шукатиметься новий альтернативний розв’язок. Отже, для реалізації обчислень з повторенням потрібно забезпечити формування якимось чином хибного результату.

Управління відкатом в основному здійснюється за допомогою умови повторення і твердження відсікання ***!***. Можна виділити три основні методи керування повтореннями з відкатом: **відкат після невдачі (ВПН); відкат з відсіканням (ВВ); повторення, що визначене користувачем (ПВК).**

- **Метод ВПН** застосовується тоді, коли потрібно реалізувати повний перебір усіх тверджень, що відповідають цільовому запиту. Це так званий безумовний повний перебір. У випадку зовнішньої мети він працює автоматично. А у випадку внутрішньої мети після того твердження, яке має повторюватися, через операцію кон’юнкції використовується довільне гарантовано хибне твердження. Це може бути щось на зразок ***1 = 2*** або стандартне хибне твердження ***fail***. Наприклад:

rule_vpn :- clause, 1=2; 1=1.

У наведеному прикладі тіло правила реалізовано у вигляді диз'юнкції тверджень: перше із них передбачає повторюване виконання деякого твердження *clause* за методом ВПН, а друге – це гарантовано істинне твердження *1 = 1*, яке забезпечить остаточну істинність усього правила *rule_vpn.*

- **Метод ВВ** застосовується тоді, коли потрібно реалізувати неповний перебір усіх тверджень, що відповідають цільовому запиту. Це так званий умовний перебір, що завершується при істинності деякої умови. Завершення повторень реалізується відсіканням ***!***, якому має передувати умова зупинки. Наприклад:

rule_vv :- clause, if_stop, !.

Хибність умови *if_stop* забезпечує повторюване виконання твердження *clause* за методом ВПН. Як тільки умова стає істинною, відсікання ***!*** „знищує" всі точки відкату і завершує повторення.

Метод ВВ дозволяє реалізовувати багаторівневі вкладені повторювані обчислення. Зокрема це стосується задач пошуку всіх можливих варіантів розв'язку, що задовольняють деякій умові. В якості прикладу пропонується задача пошуку всіх власників книг, які є фанатами творчості деякого автора. Нехай фанатом вважається той, хто має не менше 3-ох різних книг одного автора. У цьому випадку зовнішнє правило передбачає повний перебір усіх власників, що декларуються окремими твердженнями предиката арності 1, – це ніби цикл з параметром **for**. Внутрішнє правило фактично реалізує метод ВВ. У ньому умова зупинки – це кон'юнкція тверджень про володіння трьома книгами одного автора. Програмна реалізація може мати вигляд:

```
domains
    knyga = kn(avtor, nazva, rik)
    avtor, nazva = string
    rik = integer
predicates
    htos(symbol)
    vlasnyk(symbol, knyga)
    vnutr_pravylo(symbol, avtor)
    zovn_pravylo(symbol, avtor)
clauses
    htos(kolya). htos(sofiya). htos(petro). htos(olya).
    vlasnyk(olya, kn("Дж.К.Ролінг", "Гаррі Поттер.
                    Таємна кімната", 2011)).
    vlasnyk(kolya, kn("Народ", "Колобок", 2016)).
    vlasnyk(kolya, kn("Дж.К.Ролінг", "Гаррі Поттер.
                    Філософський камінь", 2010)).
    vlasnyk(kolya, kn("Дж.К.Ролінг", "Гаррі Поттер.
                    Келих вогню", 2012)).
```

```
    vlasnyk(sofiya, kn("Дж.К.Ролінг", "Гаррі Поттер.
                    Філософський камінь", 2010)).
    vlasnyk(sofiya, kn("Дж.К.Ролінг", "Гаррі Поттер.
                    Таємна кімната", 2011)).
    vlasnyk(sofiya, kn("Дж.К.Ролінг", "Гаррі Поттер.
                    Келих вогню", 2012)).
    vlasnyk(petro, kn("Дж.К.Ролінг", "Гаррі Поттер.
                    Таємна кімната", 2011)).
    vlasnyk(petro, kn("Народ", "Колобок", 2016)).
    vlasnyk(petro, kn("Народ", "Рукавичка", 2015)).
    vlasnyk(petro, kn("Народ", "Івасик-Телесик", 2014)).
    vlasnyk(petro, kn("Народ", "Курочка Ряба", 2013)).
    vnutr_pravylo(X, A) :- vlasnyk(X, kn(A, N1, _ )),
                           vlasnyk(X, kn(A, N2, _ )),
                           vlasnyk(X, kn(A, N3, _ )), N1<N2, N2<N3, !.
    zovn_pravylo(X, A) :- htos(X), vnutr_pravylo(X, A).
goal
    zovn_pravylo(X, A), write(X, " – фанат автора ", A), nl, 1=2; 1=1.
------------------------
sofiya – фанат автора Дж.К.Ролінг
petro – фанат автора Народ
```

Зауваження. *olya* не є фанатом, оскільки має лише одну книгу; *kolya* хоч має три різних книги, але не одного автора; *sofiya* дійсно має три різних книги одного автора; *petro* має аж чотири книги – народні казки. Якщо у правилі *vnutr_pravylo* після умови *N1<N2*, *N2<N3*, яка передбачає впорядкованість по зростанню назв трьох книг і гарантує їх взаємну відмінність, не використати відсікання ***!***, то виконуватиметься повний перебір усіх можливих комбінацій по три різних книги з наявних чотирьох книг одного автора. Це привело б до алгоритмічно неточного результату із чотирьох тотожніх розв'язків *petro – фанат автора Народ.*

- **Метод ПВК** застосовується тоді, коли потрібно реалізувати процес обчислень із повторенням без використання повного чи часткового перебору деяких тверджень програми. Для цього потрібно додаткове правило, яке б гарантувало встановлення точки відкату на альтернативній вітці. Це правило має строго визначену структуру – два варіанти реалізації у такому порядку:

➢ 1-ше твердження – це твердження-факт;
➢ 2-ге твердження – це твердження-правило, тіло якого містить лише виклик самого себе.

Терм такого правила може бути довільним. Наприклад, три варіанти абсолютно однакових за змістом тверджень:

```
    predicates
```

```
    pvk
    a
    repeat
clauses
    pvk.        pvk :- pvk.
    a.          a :- a.
    repeat.     repeat :- repeat.
```

Таке додаткове правило входить у склад іншого правила, яке буде основним і реалізує повторення. Структура цього правила теж є строго визначеною. Тіло правила обов'язково починається з додаткового правила *pvk*; далі слідують твердження, що мають повторюватись; потім – умова зупинки, яка при хибному значенні забезпечить черговий крок повторень за методом ВПН; завершується тіло правила відсіканням ***!*** як в методі ВВ:

```
    rule_pvk :- pvk, clause, if_stop, !.
```

Виклик в якості мети правила *rule_pvk*, передбачає послідовне обчислення підцілей, які утворюють його тіло. Для виконання підцілі *pvk* спочатку застосовується перший варіант її реалізації – твердження-факт, а на другому варіанті – твердженні-правилі – ставиться точка відкату. Далі обчислюється основна повторювана підціль *clause*. Потім перевіряється умова зупинки *if_stop*. У випадку її хибності виконується відкат у останню точку відкату – на другий варіант *pvk*. Обчислення нової підцілі – правила *pvk* – передбачає обчислення його тіла, що містить лише виклик цього ж самого твердження. Для виконання підцілі *pvk* знову спочатку застосовується перший варіант реалізації – твердження-факт, а на другому варіанті – твердженні-правилі – ставиться точка відкату і т.д. Весь процес обчислень повторюється до тих пір, поки умова зупинки *if_stop* не стане істинною. Тоді виконується відсікання ***!***, всі точки відкату „руйнуються" і повторення завершуються.

Розглянемо наступний приклад: з клавіатури послідовно вводяться довільні цілі числа. Для кожного з них потрібно вивести повідомлення про знак числа (відповідне правило *znak(integer)* вже розглядалося раніше, коли вивчалися особливості реалізації розгалужених обчислень). Умова зупинки – введення від'ємного числа **-32768**.

```
predicates
    pvk
    go
    znak(integer)
clauses
    pvk.        pvk :- pvk.
    go :- pvk, write("Введіть ціле : "), readint(X), znak(X), X = -32768, !.
    znak(X) :- X > 0, !, write("додатнє"), nl;
               X < 0, !, write("від'ємне"), nl;
                         write("нуль"), nl.
```

2.2. Рекурсія

Усі розглянуті методи управління повтореннями з відкатом не дозволяють реалізувати обчислення, які передбачають накопичення деякого результату (сумування, обробка рядків, списків, файлів тощо). Це пов'язано з тим, що у мові Prolog не можна переприсвоювати значення змінних у поточній області видимості ідентифікаторів. Виходом із даної ситуації є використання рекурсивних правил. Тоді поточне значення деякої змінної передається в якості фактичного параметра чергового виклику твердження, а це вже інша область видимості ідентифікаторів.

Подібно до правил методу ПВК рекурсивні правила у своєму тілі обов'язково містять звертання до цього ж самого твердження. Однак на відміну від правила ПВК, для якого в якості першого варіанту реалізації використовується твердження-факт, а в якості другого варіанту – твердження-правило, рекурсивні правила містять взаємний виклик у першому варіанті реалізації.

Прикладом простого рекурсивного правила, яке багатократно друкує на екрані повідомлення *"Привіт"* із переходом на початок нового рядка, може бути наступне:

```
predicates
    recurs_rule_1
clauses
    recurs_rule_1 :- write("Привіт"), nl, recurs_rule_1.
```

Тут у тілі правила взаємному виклику передують твердження *write("Привіт"), nl*, які реалізують вивід повідомлення – це дія на так званому **прямому ході рекурсії (ПХР)**. Якщо тепер виконати цільовий запит *recurs_rule_1*, то це приведе до багатократного повторення такої дії. Кожен новий рекурсивний виклик твердження *recurs_rule_1* приводить до виділення частини оперативної пам'яті, яка називається програмним стеком (розмір стеку не перевищує 64 кБ). Як тільки стек буде вичерпаний, обчислення перериваються аварійно з помилкою виконання програми ***stack owerflow*** (переповнення стеку).

Якщо у тілі правила поміняти місцями твердження повторюваної дії та рекурсивний виклик, наприклад,

```
predicates
    recurs_rule_2
clauses
    recurs_rule_2 :- recurs_rule_2, write("Привіт"), nl.
```

то цільовий запит *recurs_rule_2* не забезпечить повторюваного виводу повідомлення. У пам'яті буде запущено багато однакових процесів, але в жодному із них до тверджень *write("Привіт"), nl* черга так і не дійде через помилку виконання програми ***stack owerflow***.

Для коректного завершення процесу обчислень з рекурсивним повторенням у тілі правила перед взаємним викликом має

використовуватися особлива умова виконання чергового кроку рекурсії (у попередніх прикладах така умова відсутня). Істинність умови дозволяє виконати черговий виклик, а хибність перериває цей процес – це так звана **точка зупинки рекурсії (ТЗР)**. При цьому має бути альтернативний варіант реалізації рекурсивного правила, який має виконуватися у ТЗР.

Після ТЗР починається обернений процес – завершення обчислень усіх тверджень, які були викликані на прямому ході рекурсії, – це так званий **зворотний хід рекурсії (ЗХР)**. На ЗХР теж можуть обчислюватися твердження, які реалізують інший процес обчислень з повторенням. При цьому повторення на ПХР і на ЗХР мають взаємно протилежні напрями: першому кроку на ПХР відповідає останній крок на ЗХР і навпаки.

Розглянемо більш складніший приклад рекурсивного правила з параметром:

```
predicates
    recurs_rule_3(integer)
clauses
    recurs_rule_3(N) :- N > 0, !, N1=N-1,
                        write("Привіт. Ще буде ", N1, " раз(u/ів)"), nl,
                        recurs_rule_3(N1);
                        write("Стоп"), nl.
GOAL:
recurs_rule_3(5)
-------------------------
Привіт. Ще буде 4 раз(u/ів)
Привіт. Ще буде 3 раз(u/ів)
Привіт. Ще буде 2 раз(u/ів)
Привіт. Ще буде 1 раз(u/ів)
Привіт. Ще буде 0 раз(u/ів)
Стоп
```

Поміняємо тепер місцями основні повторювані твердження та рекурсивний виклик у тілі правила:

```
predicates
    recurs_rule_4(integer)
clauses
    recurs_rule_4(N) :- N > 0, !, N1=N-1,
                        recurs_rule_4(N1),
                        write("Привіт. Ще буде ", N1, " раз(u/ів)"), nl;
                        write("Стоп"), nl.
GOAL:
recurs_rule_4(5)
-------------------------
Стоп
Привіт. Ще буде 0 раз(u/ів)
```

Привіт. Ще буде 1 раз(и/ів)
Привіт. Ще буде 2 раз(и/ів)
Привіт. Ще буде 3 раз(и/ів)
Привіт. Ще буде 4 раз(и/ів)

Як видно, обмін місцями основних повторюваних тверджень та рекурсивного виклику у тілі правила приводить до зміни порядку повторюваних дій. У випадку правила *recurs_rule_3* повторення основної дії виконується на ПХР до ТЗР у тому ж порядку, в якому змінюється параметр рекурсії *N*. А на ЗХР ніяких дій не виконується, окрім завершення чергового виклику твердження. У випадку правила *recurs_rule_4* на ПХР лише зменшується параметр *N*, а повторення основної дії виконується на ЗХР вже після ТЗР. При цьому порядок повторюваних дій обернений до порядку зміни параметра рекурсії.

В основі рекурсії використовується той самий механізм відкату, що і при ітераційних повтореннях. При кожному рекурсивному виклику УПП спочатку „аналізують" перший варіант правила з рекурсивним звертанням, а на альтернативному варіанті правила, який відповідає ТЗР, „ставиться" точка відкату. Використання відсікання ***!*** відразу після умови продовження рекурсії у першому варіанті правила гарантує, що реалізація правила для ТЗР спрацює лише один раз у випадку хибності цієї умови.

Враховуючи все вище зазначене, загальну структуру рекурсивних правил можна визначити так:

рекурсивне_правило :- умова_продовження_рекурсії
[, повторювані_дії_ПХР] ,
рекурсивне_правило
[, повторювані_дії_ЗХР] ;
умова_зупинки_рекурсії [, дія_ТЗР] .

Тут, як і раніше, у квадратних дужках ***[ ]*** позначені необов'язкові елементи. Проте необов'язковість ***повторюваних_дій_ПХР*** зовсім не означає, що на ПХР не має ніяких тверджень, які змінюють параметр рекурсії. Якщо цього не забезпечити, то ***умова_продовження_рекурсії*** ніколи не змінить свого значення на хибне, і рекурсивні повторення будуть нескінченними.

Замість ***умови_зупинки_рекурсії*** можна використати відсікання ***!*** після ***умови_продовження_рекурсії***:

рекурсивне_правило :- умова_продовження_рекурсії, !
[, повторювані_дії_ПХР] ,
рекурсивне_правило
[, повторювані_дії_ЗХР] ;
дія_ТЗР .

Рекурсивні правила дозволяють реалізовувати повторювані обчислення з накопиченням деякого результату. Замість операції переприсвоєння, що реалізує заміну значення деякої змінної, у рекурсивних правилах використовують механізм передачі параметрів. Те

значення, що передається у твердження в якості фактичного параметра під деяким іменем, у тілі твердження формально матиме вже інше ім'я, яке визначене при оголошенні. Тобто у тілі правила це вже нова змінна із переданим їй значенням.

У розглянутих прикладах рекурсивних правил *recurs_rule_3* та *recurs_rule_4* параметр *N* у голові – це формальне ім'я того значення, що фактично передається у цільовому запиті. Обчислення мети передбачає обчислення тверджень, що утворюють тіло відповідного правила. При цьому в програмному стеку виділяється пам'ять під нову локальну змінну *N1*, значення якої обчислюється через значення параметра *N.* Далі нове фактичне значення з іменем *N1* передається у черговий рекурсивний виклик. *N1* – це фактичний параметр, який на наступному рівні обчислень формально знову має ім'я *N* і т.д. Очевидно, чим більша кількість параметрів у рекурсивному твердженні і локальних змінних у його тілі, тим швидше вичерпується стек, а отже глибина рекурсії зменшується.

Накопичення результату можна реалізувати як на ПХР, так і на ЗХР. В залежності від того на якому етапі рекурсії здійснюються основні повторювані обчислення, рекурсивні твердження матимуть різну кількість параметрів:

- **якщо основні повторювані обчислення реалізується на ПХР**, то на кожен наступний рівень має передаватись чергове значення накопичуваного результату. Окрім параметра, через який передається нове значення результату, має використовуватись ще один параметр такого самого доменного типу. Додатковий параметр при кожному виклику є вільною змінною з одним і тим самим іменем, а отже не передбачає створення у пам'яті нових локальних копій. У ТЗР цей параметр отримує своє значення від того параметра, через який щоразу передавалося нове значення накопичуваного результату. Оскільки додатковий параметр – це одна і таж змінна для усіх викликів, то на ЗХР при „згортанні" рекурсії викликів остаточне значення накопиченого результату повертається через цей параметр у точку виклику;
- **якщо основні повторювані обчислення реалізується на ЗХР**, то і накопичення результату проводиться після ТЗР. На ПХР виконуються лише рекурсивні виклики твердження. Для кожного чергового виклику правила в якості фактичного параметра використовується своя локальна змінна для проміжного значення накопичуваного результату. Кожна з таких змінних є вільною аж до ТЗР. У ТЗР остання із створених змінних отримує деяке початкове значення. На ЗХР попередньо створена локальна змінна, яка є параметром передостаннього виклику, отримує змінене значення на основі значення останньої локальної змінної з ТЗР. Змінна, яка є параметром передпередостаннього виклику, отримує нове значення на основі значення передостанньої локальної змінної і т.д. Коли завершується перший з викликів – власне цільовий запит, то вільна змінна, що є його параметром, отримує остаточне значення

накопичуваного результату. Отже, кількість параметрів для накопичення результату є меншою – достатньо одного. Усі локально створені тимчасові об'єкти зберігаються в пам'яті аж до ЗХР і знищуються при „згортанні" рекурсії в порядку, оберненому до порядку їх створення. Таким чином максимальне використання програмного стеку припадає саме на ТЗР.

У якості прикладу пропонується два варіанти рекурсивних правил для обчислення факторіалу деякого невід'ємного цілого числа. У випадку повторюваних обчислень на ПХР із зменшенням параметра рекурсії факторіал числа N фактично визначається за формулою $N!=N*(N-1)*...*2*1$. Якщо ж повторювані обчислення виконуються на ЗХР, то факторіал числа N визначається за рекурентною схемою $N!=\begin{cases} N*(N-1)!, & N>0 \\ 1, & N=0 \end{cases}$.

```
predicates
    fact_pr(integer, real, real)
    fact_zv(integer, real)
clauses
    fact_pr(N, F, FF) :- N>0, !, N1=N-1, F1=F*N, fact_pr(N1, F1, FF).
    fact_pr(0, F, FF) :- FF=F.          /* або   fact_pr(0, F, F). */

    fact_zv(N, F) :- N>0, !, N1=N-1, fact_zv(N1, F1), F=F1*N.
    fact_zv(0, F) :- F=1.               /* або   fact_zv(0, 1). */
```

Цільові запити у випадку зовнішньої мети можуть бути наступними (у коментарі відображено послідовність повторюваних операцій множення):

```
GOAL:
fact_pr(5, 1, F)
-------------------------
F = 120                             /* 5! = 1*5*4*3*2*1 = 120 */

fact_zv(5, F)
-------------------------
F = 120                             /* 5! = 1*1*2*3*4*5 = 120 */
```

Зауваження:

- рекурсивні правила з повторюваними обчисленнями на ЗХР хоч і мають меншу кількість параметрів, проте значно більше використовують програмний стек. У стеку зберігаються усі створені локальні змінні для кожного виклику. Тому глибина рекурсії для таких правил значно менша;

- у мові Prolog при звертанні до будь-якого твердження в якості фактичного параметра не можна передавати вираз. На противагу цьому у імперативних мовах програмування виклик підпрограм, фактичними параметрами яких є вираз, є абсолютно коректним. Це пов'язано з тим, що в імперативних мовах основними елементами інформації або даних є **значення** того чи іншого типу, а у мові Prolog такими базовими елементами є **образи** і **твердження**, що пов'язують образи. Образом може бути терм (ідентифікатор) або зображення – конкретне значення певного доменного типу. Вираз має значення, але вираз не є образом, тому помилкою є твердження *fact_zv(N-1, F1)*;
- при виклику тверджень значення фактичних параметрів передаються виключно через терми змінних, констант або зображення, наприклад *N1=N-1, fact_zv(N1, F1).*

РОЗДІЛ 3. СПИСКИ

У більшості мов програмування використовуються розгорнуті ієрархії типів даних. Усі типи поділяють на скалярні і складені. До складених типів традиційно відносять:

- **регулярні типи**, що є особливим чином організованим набором елементів одного і того ж базового типу (масиви, рядки, множини);
- **комбіновані типи**, що є сукупністю елементів різних базових типів (записи, записи з варіантами, структури, об'єднання, об'єкти, класи, складені домени з альтернативами);
- **послідовності**, що є структурами даних з послідовним доступом до елементів одного і того ж базового типу (списки, файли).

У мові Prolog реалізується декілька механізмів структурування даних, зокрема складені домени і домени з альтернативами є прикладом комбінованих типів. В якості одного із стандартних регулярних доменів використовуються рядки символів ***string***. Також Prolog дозволяє обробляти дискові файли як файли послідовного і прямого доступу.

Особливістю систематики доменів мови Prolog є те, що у ній відсутні такі регулярні складені типи даних як масиви. Це пов'язано з тим, що Prolog оперує образами, а масив не є цілісним образом. У цій мові використовуються структури даних, які мають виражений рекурсивний характер. Рекурсивні структури даних передбачають виокремлення одного першого за порядком елемента („голови"), а залишок („хвіст") знову є структурою даних такого самого типу. Таке виділення елементів із структури даних можна проводити багатократно аж до отримання порожнього залишку.

Однією з найбільш поширених і часто використовуваних у логічному програмуванні структур даних є **списки**. На відміну від списків у імперативних мовах програмування (Pascal, C/C++), де такі об'єкти є динамічними структурами даних і реалізуються виключно через вказівники, у мові Prolog списки є статичними структурами даних та представляються своїми іменами.

Списки – це фіксована послідовність елементів одного і того ж базового типу. Подібно до масивів структура списків не може змінюватись, однак на відміну від масивів у списках також не можуть змінюватися окремі елементи і відсутня безпосередня індексація елементів. У списках використовується послідовний доступ до елементів, але при цьому початковий список розділяється на один перший елемент і залишковий список, тобто утворюється два нових об'єкти.

3.1. Оголошення спискового домену

Оголошення спискових доменів здійснюється відповідно до синтаксичного правила:

терм_спискового_домену = терм_базового_домену*

При оголошенні нового домену, що є списком, використовується особлива операція побудови спискового домену, яка позначається символом зірочка (*****). Ця операція є унарною постфіксною, тобто застосовується до одного операнда зліва – терму базового домену. Наприклад:

```
domains
    list_i = integer*                   /* список цілих чисел */
    list_r = real*                      /* список дійсних чисел */
    list_str = string*                  /* список рядків */
    list_list_i = list_i*               /* список списків цілих чисел */
    list_kn = knyga*                    /* список складених доменів */
```

3.2. Задання списків

Список як фіксована структура даних задається послідовним переліком через символ кома (**,**) усіх його елементів у парі квадратних дужок ***[ ]***. У випадку списків квадратні дужки ***[ ]*** є обов'язковими елементами. Наприклад:

```
[1, 2, 3]                           /* список із трьох цілих */
[3.14, 0]                           /* список із двох дійсних */
["ABC123", "Коля", ""]              /* список із трьох рядків */
[[1], [2, 3], [4, 5, 6]]            /* список із трьох списків */
[knyga("Народ", "Колобок", 2016)]   /* список з однієї структури */
[ ]                                 /* порожній список */
[ [ [ ] ] ]                 /* список з одного елемента, */
                            /* що є списком з одного елемента, */
                            /* що є порожнім списком */
```

3.3. Операція розділення списку на голову і хвіст (РСГХ)

Для розділення списку на окремі елементи з метою їх подальшої роздільної обробки можна використати точно визначену кількість ідентифікаторів вільних змінних. Окремі із цих змінних можуть бути анонімними. Наприклад:

```
GOAL:
L = [1, 2, 3, 4, 5], L = [A, B, C, D, E]
-------------------------
L=[1, 2, 3, 4, 5], A=1, B=2, C=3, D=4, E=5
```

L = [1, 2, 3, 4, 5], L = [_, _, X, _, _]

L = [1, 2, 3, 4, 5], X=3

F = 1, G = 2, H = 3, I = 4, J = 5, L = [F, G, H, I, J]

F = 1, G = 2, H = 3, I = 4, J = 5, L=[1, 2, 3, 4, 5]

Оскільки тут мета є зовнішньою, то автоматично виводяться значення всіх змінних, що використовуються у твердженнях цільових запитів. У першому прикладі має місце розділення списку на окремі елементи, в другому реалізується доступ до третього за порядком елемента, а в третьому виконується з'єднання послідовності окремих елементів у один список. Такий спосіб доступу до елементів є незручним, оскільки вимагає використання точно такої ж кількості вільних змінних, скільки є елементів у самому списку. А це може бути або невідома наперед кількість або дуже велике число.

Для виокремлення одного першого за порядком елемета у мові Prolog використовується особлива бінарна операція, що називається **розділенням списку на голову і хвіст (РСГХ)** і позначається символом вертикальна риска або вертикальний слеш (**|**). Префіксний операнд – це або один перший за порядком елемент, або перелік декількох початкових елементів списку – **голова списку**; постфіксний операнд – залишок списку без першого елемента або певної кількості початкових елементів – **хвіст списку**. Таким чином синтаксичне правило для операції РСГХ наступне:

список = [голова | хвіст] .

Часто використовуються такі позначення: *L* – список (від англ. **list**), *H* – голова (від англ. **head**), *T* – хвіст (від англ. **tail**). Тоді справедливим буде запис

L = [H | T].

Тут ліва і права частини рівності є еквівалентними позначеннями одного і того ж списку.

Оскільки операція РСГХ відділяє в якості голови перший елемент списку, то її не можна застосовувувати до порожніх списків. Хвіст за своєю природою – це теж список того самого базового типу. У свою чергу хвіст може бути порожнім списком. У загальному випадку операцією РСГХ можна відділити довільну кількість перших елементів, якщо їх достатньо у списку. Наприклад:

GOAL:

L = [1, 2, 3], L = [A | [B | [C | T]]]

L = [1, 2, 3], A=1, B=2, C=3, T=[ ]

[1, 2, 3, 4, 5] = [A, B, C | T]

A=1, B=2, C=3, T=[4, 5]

L = [1, 2], L = [A, B, C | T]

No

Операція РСГХ може виконувати не тільки функцію розділення списку, а й приєднання нових елементів у якості голови до поточного списку:

GOAL:
T = [2, 3, 4], H = 1, L = [H | T]

T = [2, 3, 4], H = 1, L=[1, 2, 3, 4]

L = [1, 2, 3 | [4, 5]]

L=[1, 2, 3, 4, 5]

3.4. Обробка списків

Обробка списків часто передбачає виконання наступних дій:

- формування списку;
- вивід (перегляд);
- пошук елементів (за значенням, за позицією);
- вставка нового елемента (у задану позицію, перед заданим елементом, після заданого елемента);
- видалення елемента (із заданої позиції, із заданим значенням);
- конкатенація – з'єднання списків;
- розділення списків на частини (по заданій позиції, по елементу із заданим значенням, сепарація за умовою);
- сортування списків;
- компоновка даних у список.

Формування списку

Список можна формувати або за принципом стеку, або за принципом черги. У випадку стеку елементи у списку матимуть оберений порядок до порядку їх введення, а у випадку черги порядок елементів у списку співпадатиме з порядком їх введення. Зміна порядку елементів при формуванні списку можлива за рахунок застосування операції РСГХ на прямому чи зворотному ході рекурсії.

Формування списку-стеку можливе при використанні операції РСГХ на ПХР. Якщо ж РСГХ використовується на ЗХР, то отримаємо список-чергу. Наприклад:

```
domains
    list_i=integer*
predicates
    stvor_stek_1(integer, list_i, list_i)
    stvor_cherga_1(integer, list_i)
clauses
    stvor_stek_1(N, L, LL) :- N>0, !
                              write("Введіть число : "), readint(X),
                              L1=[X|L], N1=N-1,
                              stvor_stek_1(N1, L1, LL);
                              LL=L, write("Ввід даних завершено"), nl.

    stvor_cherga_1(N, L) :- N>0, !,
                            write("Введіть число : "), readint(X),
                            N1=N-1, stvor_cherga_1(N1, L1), L=[X|L1].
    stvor_cherga_1(0, [ ]) :- write("Ввід даних завершено"), nl.
```

Для правила *stvor_stek_1* дія ТЗР, що передбачає копіювання накопиченого списку у результуючу вільну змінну *LL=L*, реалізована у вигляді альтернативної вітки диз'юнкції. А для правила *stvor_cherga_1* дія ТЗР, що передбачає початкову ініціалізацію результату порожнім списком, реалізована окремим твердженням *stvor_cherga_1(0, [ ])*.

Цільові запити по формуванню списків у випадку зовнішньої мети можуть бути такими:

```
GOAL:
stvor_stek_1(3, [ ], L)
------------------------
Введіть число : 4
Введіть число : 5
Введіть число : 6
Ввід даних завершено
L=[6, 5, 4]

stvor_cherga_1(3, L)
------------------------
Введіть число : 4
Введіть число : 5
Введіть число : 6
Ввід даних завершено
L=[4, 5, 6]
```

Зауваження. Для правила *stvor_cherga_1* по формуванню списку-черги, основну повторювану дію – операцію РСГХ, що приєднує новий елемент до поточного списку *L=[X|L1]*, можна перемістити в голову правила на місце параметра, через який повертається результат

формування списку. Ця дія буде виконуватися на ЗХР після ТЗР. Такий підхід є виключенням із загального правила, що забороняє використовувати вирази в якості параметрів тверджень. Операція РСГХ – це не є вираз у традиційному розумінні, а лише спосіб задання списку. Формально запис *[X|L]* є образом цілого списку, в якому головою є елемент *X*, а хвостом – список *L*. Новий варіант правила *stvor_cherga_1* є абсолютно еквівалентним попередньому:

```
stvor_cherga_1(N, [X|L]) :- N>0, !,
                            write("Введіть число : "), readint(X),
                            N1=N-1, stvor_cherga_1(N1, L);
                            L=[ ], write("Ввід даних завершено"), nl.
```

Тут дія ТЗР, що передбачає початкову ініціалізацію результату порожнім списком *L=[ ]*, реалізована у вигляді альтернативної вітки диз'юнкції, а не окремим твердженням.

У розглянутих прикладах програм в якості умови продовження рекурсивних повторень є додатня кількість елементів, які потрібно ще ввести у список, *N>0*. Формування списку можна здійснювати і при іншій умові, наприклад, поки коректно вводяться цілі числа. Таким чином, умовою зупинки є некоректний ввід даних при використанні стандартного твердження для читання цілих чисел ***readint***. У цьому випадку відповідні правила можуть мати меншу кількість параметрів, наприклад:

```
predicates
    stvor_stek_2(list_i, list_i)
    stvor_cherga_2(list_i)
clauses
    stvor_stek_2(L, LL) :- write("Введіть ціле число : "),
                           readint(X), !, stvor_stek_2([X|L], LL).
    stvor_stek_2( L, L ) :- write("Ввід даних завершено"), nl.

    stvor_cherga_2([X|L]) :- write("Введіть ціле число : "),
                             readint(X), !, stvor_cherga_2(L);
                             L=[ ], write("Ввід даних завершено"), nl.
```

Тут теж використано передачу в якості параметра списку із операцією РСГХ *[X|L]*, а також різні варанти реалізації дії ТЗР. Зокрема, для правила *stvor_stek_2* дія ТЗР окрім виводу повідомлення передбачає копіювання значення сформованого списка в результуючу вільну змінну безпосередньо в голові правила *stvor_stek_2(L, L)*.

Цільові запити по формуванню списків у випадку зовнішньої мети можуть бути такими:

```
GOAL:
stvor_stek_2([ ], L)
------------------------
Введіть число : 4
```

Введіть число : 5
Введіть число : 6
Введіть число : 7.5 */* число 7.5 не є цілим */*
Ввід даних завершено
L=[6, 5, 4]

stvor_cherga_2(L)

Введіть число : 4
Введіть число : 5
Введіть число : 6
Введіть число : 123456789 */* число 123456789 не є цілим */*
Ввід даних завершено
L=[4, 5, 6]

Перегляд списку

Для перегляду списку знову використовується операція РСГХ. Очевидно, що вона має бути на ПХР, а операція виводу значення елемента може бути як на ПХР, так і на ЗХР. У першому випадку елементи виводитимуться в тому ж порядку, що є у списку, а в другому випадку порядок виводу буде обернений. В обох випадках умовою зупинки є розділення списку на елементи до порожнього хвоста. Наприклад:

```
predicates
    print_pr(list_i)
    print_zv(list_i)
clauses
    print_pr([H|T]) :- !, write(H), nl, print_pr(T); write("кінець"), nl.
    print_zv([H|T]) :- !, print_zv(T), write(H), nl; write("кінець"), nl.

GOAL:
print_pr([1, 2, 3])
---
1
2
3
кінець                          /* дія ТЗР */

print_zv([1, 2, 3])
---
кінець                          /* дія ТЗР */
3
2
1
```

Пошук елемента за значенням

Результатом пошуку має бути або номер позиції **першого за порядком входження елемента із заданим значенням**, або повідомлення про його відсутність. Очевидно, що потрібно використовувати додатковий параметр – лічильник поточної позиції. Пошук, як і більшість інших операцій, доцільно реалізувати на ПХР. Отже, правило матиме чотири параметри: номер поточної позиції у списку, результуюча позиція шуканого елемента, задане значення елемента, власне сам список. Умов зупинки рекурсії буде дві:

- чергова голова співпадає із заданим значенням – позитивний результат. У цьому випадку окрім друку відповідного повідомлення в якості результату повертається номер позиції входження;
- поточний список порожній – негативний результат. У цьому випадку окрім друку відповідного повідомлення в якості результату позиції входження повертається нуль.

Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    poshuk_znach(integer, integer, integer, list_i)
clauses
    poshuk_znach(J, P, X, [H|T]) :- H<>X, !,
                                    J1=J+1, poshuk_znach(J1, P, X, T).
    poshuk_znach(J, J, X, [X|T]) :- write("елемент ", X, " є у списку"),nl.
    poshuk_znach(_, 0, X, [ ]) :- write("елемент ", X, " відсутній"), nl.

GOAL:
poshuk_znach(1, P, 5, [4, 5, 6, 5])
-------------------------
елемент 5 є у списку
P=2

poshuk_znach(1, P, 3, [4, 5, 6, 5])
-------------------------
елемент 3 відсутній
P=0
```

Іншим різновидом пошуку за значенням є пошук **усіх входжень елементів із заданим значенням.** Оскільки потрібно перевірити усі елементи списку, то така операція пошуку матиме лише одну ТЗР – досягнення кінця списку. Проте у цьому випадку буде два варіанти реалізації правила із рекурсивним викликом:

- черговий елемент не співпадає із заданим значенням – перехід на наступний елемент без зміни кількості входжень;
- черговий елемент співпадає із заданим значенням – перехід на наступний елемент, і кількість входжень збільшується на 1.

Відповідне правило матиме п’ять параметрів: номер поточної позиції в списку, поточне значення кількості знайдених елементів, остаточна кількість знайдених елементів, задане значення шуканих елементів, власне сам список. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    poshuk_usih(integer, integer, integer, integer, list_i)
clauses
    poshuk_usih(J, K, KK, X, [H|T]) :- X<>H, !, J1=J+1,
                                        poshuk_usih(J1, K, KK, X, T).
    poshuk_usih(J, K, KK, X, [X|T]) :- !,
                                       write("елемент ", X," є в позиції ", J), nl,
                                       J1=J+1, K1=K+1,
                                       poshuk_usih(J1, K1, KK, X, T).
    poshuk_usih(_, K, K, X, [ ]) :- K=0, !,
                                    write("елемент ", X, " відсутній"), nl;
                                    write("всього ", K, " елемент(и/ів)"), nl.

    GOAL:
    poshuk_usih(1, 0, K, 4, [1, 4, 2, 4, 3, 4])
    -------------------------
    елемент є в позиції 2
    елемент є в позиції 4
    елемент є в позиції 6
    всього 3 елемент(и/ів)
    K=3

    poshuk_usih(1, 0, K, 5, [1, 4, 2, 4, 3, 4])
    -------------------------
    елемент 5 відсутній
    K=0
```

Пошук елемента за позицією входження

У цьому випадку встановлюється значення елемента в заданій позиції списку. Якщо ж позиція задана некоректно, тобто перевищує довжину списку або менша від **1**, то виводиться повідомлення про негативний результат. Відповідне правило матиме чотири параметри: номер поточного елемента, задана позиція в списку, шукане значення елемента, власне сам список. Умов зупинки рекурсії буде дві:

- номер поточної позиції в списку співпадає із заданою позицією – позитивний результат. У цьому випадку, окрім друку відповідного повідомлення, в якості результату повертається шукане значення елемента у цій позиції;

- усі інші випадки, коли задана позиція не відповідає реальній довжині списку, – негативний результат. У цьому випадку, окрім друку відповідного повідомлення, в якості результату має повертатися якесь значення базового типу. Зокрема, це може бути значення анонімної змінної _, що спричинить вивід повідомлення компілятора.

Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    poshuk_poz(integer, integer, integer, list_i)
clauses
    poshuk_poz(J, P, X, [_|T]) :- J<P, !, J1=J+1, poshuk_poz(J1, P, X, T).
    poshuk_poz(P, P, H, [H|T]) :- !, write("елемент є у списку"), nl.
    poshuk_poz(_, _, _, _) :- write("елемент відсутній"), nl.

GOAL:
poshuk_poz(1, 3, X, [4, 5, 6, 7])
-------------------------
елемент є у списку
X=6

poshuk_poz(1, 10, X, [4, 5, 6, 7])
-------------------------
елемент відсутній
X=_

poshuk_poz(1, -10, X, [4, 5, 6, 7])
-------------------------
елемент відсутній
X=_
```

Вставка нового елемента в задану позицію

У цьому випадку формується новий список із новим елементом. Якщо позиція вставки перевищує довжину списку або менша від **1**, то для однозначності дій проводиться вставка відповідно в кінець або на початок списку. Відповідне правило матиме п'ять параметрів: номер поточного елемента, задана позиція вставки, значення нового елемента, початковий список, результуючий список.

На ПХР перші елементи вхідного списку аж до заданої позиції вставки відділяються операцією РСГХ і тимчасово розміщуються в програмному стеку. У ТЗР на звільнене місце операцією РСГХ приєднується новий елемент, і отриманий список стає основою нового результуючого списку. На ЗХР копії раніше відділених елементів повернуться зі стеку в результуючий список із збереженням початкового порядку. На перший погляд умов зупинки рекурсії має бути дві: номер поточної позиції в списку співпадає із позицією вставки; позиція вставки

не відповідає реальній довжині списку. Проте за попередньою домовленістю в обох випадках мають виконуються однакові дії: новий елемент приєднується операцією РСГХ в якості голови до поточного залишку списку, і отриманий список стає основою нового результуючого списку.

Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    vstavka_poz(integer, integer, integer, list_i, list_i)
clauses
    vstavka_poz(J, P, X, [H|T], [H|T1]) :- J<P, !, J1=J+1,
                                           vstavka_poz(J1, P, X, T, T1).
    vstavka_poz(_, _, X, T, [X|T]).

GOAL:
vstavka_poz(1, 3, 6, [4, 5, 7, 8], L)
-------------------------
L=[4, 5, 6, 7, 8]

vstavka_poz(1, 10, 6, [4, 5, 7, 8], L)
-------------------------
L=[4, 5, 7, 8, 6]

vstavka_poz(1, -10, 6, [4, 5, 7, 8], L)
-------------------------
L=[6, 4, 5, 7, 8]

vstavka_poz(1, 3, 6, [4, 5, 7, 8], [4, 5, 6, 7, 8])
-------------------------
Yes

vstavka_poz(1, 3, X, [4, 5, 7, 8], [4, 5, 6, 7, 8])
-------------------------
X=6
```

Зауваження. Два останні приклади цільових запитів ілюструють можливість використання одного і того ж твердження для виконання інших дій: перевірка, чи дійсно в результаті вставки нового елемента **6** в **третю** позицію початкового списка **[4, 5, 7, 8]** отримали новий список **[4, 5, 6, 7, 8]**; визначення того елемента, в результаті вставки якого в початковий список **[4, 5, 7, 8]** у **третю** позицію отримали новий список **[4, 5, 6, 7, 8]**.

Вставка нового елемента перед елементом чи після елемента із заданим значенням

Звичайно розуміється вставка відносно першого за порядком входження елемента з вказаним значенням. Як і в попередньому випадку формується новий список із новим елементом. Фактично обидві операції дуже подібні, оскільки передбачають пошук першого за порядком входження елемента із заданим значенням. Звичайно, що у випадку відсутності такого елемента операція вставки буде неуспішною. Відповідне правило матиме чотири параметри: задане значення елемента, відносно якого здійснюється вставка, значення нового елемента, початковий список, результуючий список.

По аналогії з попередньою дією ті елементи, що були на ПХР виокремлені операцією РСГХ з вхідного списку, на ЗХР будуть копіюватися через стек у результуючий список із збереженням початкового порядку. Основою результуючого списку в ТЗР є хвостова частина вхідного списку, до якої долучається новий елемент або перед, або після того елемента, що виявиться першим після пошуку за вказаним значенням. Умов зупинки рекурсії буде дві:

- чергова голова в списку співпадає із заданим значенням – позитивний результат. У цьому випадку виконується вставка нового елемента або перед, або після цієї голови;
- вхідний список розділений на елементи аж до порожнього хвоста, тобто елемент із заданим значенням, відносно якого має проводитися вставка, відсутній в списку, – негативний результат. У цьому випадку результуючий список співпадатиме із початковим.

Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    vstavka_pered(integer, integer, list_i, list_i)
    vstavka_pislya(integer, integer, list_i, list_i)
clauses
    vstavka_pered(X, Y, [X|T], [Y, X|T]) :- !.
    vstavka_pered(X, Y, [H|T], [H|T1]) :- X<>H, !,
                                          vstavka_pered(X, Y, T, T1).
    vstavka_pered(_, _, [ ], [ ]).

    vstavka_pislya(X, Y, [X|T], [X, Y|T]) :- !.
    vstavka_pislya(X, Y, [H|T], [H|T1]) :- X<>H, !,
                                          vstavka_pislya(X, Y, T, T1).
    vstavka_pislya(_, _, [ ], [ ]).
```

GOAL:
vstavka_pered(7, 6, [4, 5, 7, 8], L)

L=[4, 5, 6, 7, 8]

vstavka_pislya(5, 6, [4, 5, 7, 8], L)

L=[4, 5, 6, 7, 8]

vstavka_pislya(3, 6, [4, 5, 7, 8], L)

L=[4, 5, 7, 8]

vstavka_pered(7, 6, L, [4, 5, 6, 7, 8])

L=[4, 5, 7, 8]

vstavka_pislya(5, Y, [4, 5, 7, 8], [4, 5, 6, 7, 8])

Y=6

Зауваження. Два останні приклади цільових запитів ілюструють можливість використання одного і того ж твердження для виконання інших дій: встановлення початкового списку, з якого отримали новий список **[4, 5, 6, 7, 8]** в результаті вставки нового елемента **6** перед елементом **7**; визначення того елемента, в результаті вставки якого в початковий список **[4, 5, 7, 8]** після елемента **5** отримали новий список **[4, 5, 6, 7, 8]**. Для реалізації таких дій потрібно застосувати особливий порядок тверджень у розділі ***clauses***: першим має бути твердження для ТЗР у випадку позитивного результату, далі слідує рекурсивне твердження-правило, останнім є твердження для ТЗР у випадку негативного результату. Інший порядок тверджень не дасть бажаного результату в загальному випадку довільних запитів мети, а лише для окремих часткових випадків.

Видалення елемента із заданої позиції

Правило для реалізації такої дії багато в чому подібне до операції вставки. Знову на ПХР відділяється певна кількість початкових елементів, поки не досягається задана позиція. У ТЗР оперцією РСГХ вилучається перший елемент. На ЗХР копії раніше відділених елементів повернуться зі стеку в результуючий список із збереженням початкового порядку. Відповідне правило матиме чотири параметри: номер поточного елемента, задана позиція видалення, початковий список, результуючий список. Умов зупинки рекурсії є дві:

- лічильник поточної позиції в списку співпадає із заданим значенням – позитивний результат. Тоді чергова голова вилучається із списку, а хвіст стає основою формування результуючого списку. У цьому випадку новий список матиме на один елемент менше, ніж початковий;
- позиція видалення не відповідає реальній довжині списку – негативний результат. У цьому випадку початковий список буде розділений на елементи аж до порожнього хвоста. Тоді результуючий список співпадатиме із початковим.

Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    vydal_poz(integer, nteger, list_i, list_i)
clauses
    vydal_poz(P, P, [H|T], T) :- !.
    vydal_poz(J, P, [H|T], [H|T1]) :- !, J1=J+1, vydal_poz(J1, P, T, T1).
    vydal_poz(_, _, [ ], [ ]).

GOAL:
vydal_poz(1, 3, [4, 5, 6, 7, 8], L)
-------------------------
L=[4, 5, 7, 8]

vydal_poz(1, 10, [4, 5, 6, 7, 8], L)
-------------------------
L=[4, 5, 6, 7, 8]

vydal_poz(1, -10, [4, 5, 6, 7, 8], L)
-------------------------
L=[4, 5, 6, 7, 8]

vydal_poz(1, P, [4, 5, 6, 7, 8], [4, 5, 7, 8])
-------------------------
P=3

vydal_poz(1, P, [4, 5, 6, 7, 8], [4, 5, 6, 7, 8])
-------------------------
P=_

vydal_poz(1, 3, [4, 5, 6, 7, 8], [4, 5, 6, 8])
-------------------------
No
```

Зауваження. Подібно до попередніх прикладів програмної реалізації завдяки особливому порядку тверджень у розділі ***clauses*** можна застосовувати одне і те ж твердження для виконання принципово різних

дій, що пов’язані з операцією видалення елемента. Три останні приклади ілюструють наступні дії: встановлення позиції видалення елемента із початкового списку **[4, 5, 6, 7, 8]**, якщо в результаті отримали новий список **[4, 5, 7, 8]**; якщо при видаленні із початкового списку **[4, 5, 6, 7, 8]** отримали такий самий результуючий список **[4, 5, 6, 7, 8]**, то це може бути **довільна позиція поза межами реальної довжини списку**; не можна при видаленні **третього** за порядком елемента із початкового списку **[4, 5, 6, 7, 8]** отримати результуючий список **[4, 5, 6, 8]**.

Видалення першого серед елементів із заданим значенням

Відповідне правило матиме три параметри: задане значення елемента, початковий список, результуючий список. Програмна реалізація цієї операції подібна до розглянутих вище:

```
predicates
    vydal_znach(integer, list_i, list_i)
clauses
    vydal_znach(X, [X|T], T) :- !.
    vydal_znach(X, [H|T], [H|T1]) :- !, vydal_znach(X, T, T1).
    vydal_znach(_, [ ], [ ]).

GOAL:
vydal_znach (6, [4, 5, 6, 7, 8], L)
-------------------------
L=[4, 5, 7, 8]

vydal_znach (3, [4, 5, 6, 7, 8], L)
-------------------------
L=[4, 5, 6, 7, 8]

vydal_znach(X, [4, 5, 6, 7, 8], [4, 5, 7, 8])
-------------------------
X=6

vydal_znach(X, [4, 5, 6, 7, 8], [4, 5, 6, 7, 8])
-------------------------
X=_

vydal_znach(6, L, [4, 5, 7, 8])
-------------------------
L=[6, 4, 5, 7, 8]
```

Видалення всіх елементів із заданим значенням

Програмна реалізація цієї операції подібна до розглянутої вище, але матиме два варіанти правила із рекурсивними викликами та один варіант правила для ТЗР:

```
predicates
    vydal_usih(integer, list_i, list_i)
clauses
    vydal_usih(X, [X|T], T1) :- !, vydal_usih(X, T, T1).
    vydal_usih(X, [H|T], [H|T1]) :- !, vydal_usih(X, T, T1).
    vydal_usih(_, [ ], [ ]).

GOAL:
vydal_usih(4, [1, 4, 2, 4, 3, 4], L)
------------------------
L=[1, 2, 3]

vydal_usih(5, [1, 4, 2, 4, 3, 4], L)
------------------------
L=[1, 4, 2, 4, 3, 4]

vydal_usih(X, [1, 4, 2, 4, 3, 4], [1, 2, 3])
------------------------
X=4

vydal_usih(X, [1, 4, 2, 4, 3, 5], [1, 2, 3])
------------------------
No
```

Конкатенація (з'єднання) списків

Операція конкатенації двох списків передбачає приєднання другого списку в кінець першого. При цьому формується новий результуючий список. Порядок дій наступний: на ПХР операцією РСГХ перший із списків розділяється на елементи аж до порожнього; у ТЗР другий із списків копіюється в основу результуючого списку; на ЗХР копії раніше відділених елементів першого списку із стеку повертаються в результуючий список із збереженням початкового порядку. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    concat_list(list_i, list_i, list_i)
clauses
    concat_list([ ], L, L) :- !.
    concat_list([H|T1], L2, [H|T]) :- !, concat_list(T1, L2, T).

GOAL:
```

concat_list([1, 2, 3], [4, 5, 6], L)

L=[1, 2, 3, 4, 5, 6]

concat_list(L, [4, 5, 6], [1, 2, 3, 4, 5, 6])

L=[1, 2, 3]

concat_list([1, 2, 3], L, [1, 2, 3, 4, 5, 6])

L=[4, 5, 6]

Розділення списку на частини по заданій позиції

Операція розділення списку по заданій позиції передбачає формування двох результуючих списків: у перший попадають усі початкові елементи до заданої позиції включно, а в другий – усі решта елементи. Якщо задана позиція поділу менша від **1**, то перший результуючий список – порожній, а другий співпадає із початковим списком. Якщо ж позиція поділу перевищує реальну довжину початкового списку, то все навпаки: перший результуючий список співпадає із початковим списком, а другий – порожній.

Порядок дій наступний: на ПХР за допомогою операції РСГХ із початкового списку відділяється задана кількість елементів і розміщується у стеку; у ТЗР залишок вхідного списку копіюється в другий результуючий список, а основою першого результуючого списку стає порожній список; на ЗХР копії раніше відділених елементів початкового списку із стеку повертаються в перший результуючий список із збереженням початкового порядку. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    rozdil_poz(integer, list_i, list_i, list_i)
clauses
    rozdil_poz(P, [H|T], [H|T1], L2) :- P>0, !, P1=P-1,
                                        rozdil_poz(P1, T, T1, L2).
    rozdil_poz(_, L, [ ], L).

GOAL:
rozdil_poz(3, [1, 2, 3, 4, 5, 6], L1, L2)
---
L1=[1, 2, 3], L2=[4, 5, 6]

rozdil_poz(10, [1, 2, 3, 4, 5, 6], L1, L2)
---
L1=[1, 2, 3, 4, 5, 6], L2=[ ]
```

rozdil_poz(-10, [1, 2, 3, 4, 5, 6], L1, L2)

L1=[ ], L2=[1, 2, 3, 4, 5, 6]

rozdil_poz(3, L, [1, 2, 3], [4, 5, 6])

L=[1, 2, 3, 4, 5, 6]

Розділення списку на частини по першому за порядком елементу із заданим значенням

Нехай за домовленістю елемент із заданим значенням – це буде перший елемент другого результуючого списку. Якщо заданий елемент у початковому списку відсутній, то весь цей список утворить перший результуючий список, а другий буде порожнім. Твердження, що реалізує таку дію, подібне до попереднього правила. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    rozdil_znach(integer, list_i, list_i, list_i)
clauses
    rozdil_znach(X, [X|T], [ ], [X|T]) :- !.
    rozdil_znach(X, [H|T], [H|T1], L2) :- !, rozdil_znach(X, T, T1, L2).
    rozdil_znach(_, [ ], [ ], [ ]).
```

GOAL:
rozdil_znach(6, [4, 5, 6, 7], L1, L2)

L1=[4, 5], L2=[6, 7]

rozdil_znach(4, [4, 5, 6, 7], L1, L2)

L1=[ ], L2=[4, 5, 6, 7]

rozdil_znach(2, [4, 5, 6, 7], L1, L2)

L1=[4, 5, 6, 7], L2=[ ]

rozdil_znach(X, L, [4, 5], [6, 7])

X=6, L=[4, 5, 6, 7]

Розділення списку за ознакою

Розділення за ознакою передбачає формування декількох результуючих списків, ялементи яких задовольняють різним умовам із збереженням початкового взаємного порядку. В якості прикладу можна

розглянути поділ початкового списку цілих чисел на три окремих списки із від'ємних, нульових та додатніх елементів. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    rozdil_um(list_i, list_i, list_i, list_i)
clauses
    rozdil_um([H|T], [H|T1], L2, L3) :- H<0!, rozdil_um(T, T1, L2, L3).
    rozdil_um([H|T], L1, [H|T2], L3) :- H=0!, rozdil_um(T, L1, T2, L3).
    rozdil_um([H|T], L1, L2, [H|T3]) :- H>0!, rozdil_um(T, L1, L2, T3).
    rozdil_um([ ], [ ], [ ], [ ]).

GOAL:
rozdil_um([5, -2, 0, 1, 0, -3, 0, 4, -1], L1, L2, L3)
------------------------
L1=[-2, -3, -1], L2=[0, 0, 0], L3=[5, 1, 4]

rozdil_um(L, [-2, -3, -1], [0, 0, 0], [5, 1, 4])
-------------------------
L=[-2, -3, -1, 0, 0, 0, 5, 1, 4]          /* чому такий результат? */
```

Сортування списків

Як відомо, сортування структур даних може здійснюватися за різними визначальними принципами. Традиційно використовують наступні способи: **включення (вставка), вибір, обмін, злиття**. Подібно до розглянутих раніше операцій результатом сортування буде новий список, елементи якого задовольняють умові впорядкованості, наприклад по неспаданню.

Сортування за принципом включення (вставки)

Відсортований список формується повторенням операції **вставки** чергового нового елемента з вхідного списку на відповідне йому місце в результуючому списку. Сортування здійснюється двома вкладеними рекурсивними правилами:

- внутрішнє правило реалізує вставку нового елемента в результуючий список на відповідне йому місце;
- зовнішнє правило реалізує розділення на елементи вхідного списку і повторюване виконання внутрішнього правила.

Розглянемо два варіанти зовнішнього правила з повторенням основної дії – внутрішнього правила – як на ПХР, так і на ЗХР. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    vkl_vn(integer, list_i, list_i)
    vkl_zov_1(list_i, list_i, list_i)        /* повторення на ПХР */
    vkl_zov_2(list_i, list_i)                /* повторення на ЗХР */
```

```
clauses
    vkl_vn(X, [H|T], [H|T1]) :- X>H, !, vkl_vn(X, T, T1).
    vkl_vn(X, L, [X|L]).

    vkl_zov_1([H|T], L, LL) :- !, vkl_vn(H, L, L1), vkl_zov_1(T, L1, LL).
    vkl_zov_1([ ], L, L).

    vkl_zov_2([H|T], L) :- !, vkl_zov_2(T, L1), vkl_vn(H, L1, L).
    vkl_zov_2([ ], [ ]).

GOAL:
vkl_zov_1([4, 2, 6, 3, 1, 5], [ ], L)
-------------------------
vkl_zov_1([4|[2, 6, 3, 1, 5]], [ ], L)      /* тут відображена */
vkl_zov_1([2|[6, 3, 1, 5]], [4], L)         /* послідовність підцілей */
vkl_zov_1([6|[3, 1, 5]], [2, 4], L)         /* і процес формування */
vkl_zov_1([3|[1, 5]], [2, 4, 6], L)         /* результуючого списку */
vkl_zov_1([1|[5]], [2, 3, 4, 6], L)         /* на ПХР, */
vkl_zov_1([5|[ ]], [1, 2, 3, 4, 6], L)
vkl_zov_1([ ], [1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6])          /* дія ТЗР */
-------------------------
L=[1, 2, 3, 4, 5, 6]                        /* остаточний результат */

vkl_zov_2([4, 2, 6, 3, 1, 5], L)
-------------------------
vkl_zov_2([ ], [ ])                         /* дія ТЗР, */
vkl_zov_2([5], [5])                         /* далі відображена */
vkl_zov_2([1, 5], [1, 5])                   /* послідовність дій */
vkl_zov_2([3, 1, 5], [1, 3, 5])             /* по формуванню */
vkl_zov_2([6, 3, 1, 5], [1, 3, 5, 6])       /* результуючого списку */
vkl_zov_2([2, 6, 3, 1, 5], [1, 2, 3, 5, 6]) /* на ЗХР */
vkl_zov_2([4, 2, 6, 3, 1, 5], [1, 2, 3, 4, 5, 6])
-------------------------
L=[1, 2, 3, 4, 5, 6]                        /* остаточний результат */
```

Сортування за принципом вибору

У вхідному списку **вибирається** або найменший, або найбільший елемент. Цей елемент вилучається із початкового списку і вставляється відповідно або в кінець, або на початок результуючого списку. Такий процес повторюється, поки вхідний список не стане порожнім. Оскільки основною операцією є РСГХ, то доцільно **вибирати** саме найбільший елемент у невпорядкованому списку. Тоді його можна легко приєдати в якості голови до результуючого списку. Очевидно, що кожен наступний обраний найбільший елемент не перевищуватиме попереднього.

Такий спосіб сортування потребує трьох рекурсивних правил:

- одне внутрішнє правило реалізує пошук найбільшого елемента в залишку вхідного списка;
- друге внутрішнє правило реалізує видалення першого за порядком елемента зі знайденим максимальним значенням у залишку вхідного списка. Така операція вже розглядалася раніше, це правило *vydal_znach(integer, list_i, list_i)*;
- зовнішнє правило реалізує повторюване виконання двох внутрішніх правил і формування результуючого списку.

У даному випадку зовнішнє правило виконує повторення основних дій лише на ПХР. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    poshuk_max(integer, integer, list_i)
    vybir_zov(list_i, list_i, list_i)
clauses
    poshuk_max(M, Max, [H|T]) :- H>M, !, poshuk_max(H, Max, T).
    poshuk_max(M, Max, [H|T]) :- !, poshuk_max(M, Max, T).
    poshuk_max(M, M, [ ]).

    vybir_zov([H|T], L, LL) :- !, poshuk_max(H, M, T),
                                  vydal_znach(M, [H|T], T1),
                                  vybir_zov(T1, [M|L], LL]).
    vybir_zov([ ], L, L).

    GOAL:
    vybir_zov([4, 2, 6, 3, 1, 5], [ ], L)
    -------------------------
    vybir_zov([4, 2, 6, 3, 1, 5], [ ], L)        /* тут відображена*/
    vybir_zov([4, 2, 3, 1, 5], [6], L)           /* послідовність підцілей */
    vybir_zov([4, 2, 3, 1], [5, 6], L)           /* і процес формування */
    vybir_zov([2, 3, 1], [4, 5, 6], L)           /* результуючого списку */
    vybir_zov([2, 1], [3, 4, 5, 6], L)           /* на ПХР, */
    vybir_zov([1], [2, 3, 4, 5, 6], L)
    vybir_zov([ ], [1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6])        /* дія ТЗР */
    -------------------------
    L=[1, 2, 3, 4, 5, 6]                         /* остаточний результат */
```

Сортування за принципом обміну

Принцип обміну передбачає порівняння і можливі **обміни** виключно між сусідніми елементами. Ці дії повторюються в русі вздовж струкутури даних:

- якщо порівняння і можливі обміни виконувати в русі від початку до кінця списку, то найбільший елемент „виштовхнеться” в кінець структури даних, – це так званий метод **камінця**;
- якщо порівняння і можливі обміни виконувати в русі від кінця до початку списку, то найменший елемент „виштовхнеться” на початок структури даних, – це так званий метод **бульбашки**.

Повторення того чи іншого варіанту дій дасть впорядкований результуючий список.

Звичайно, що після кожного етапу з перетвореного вхідного списку має вилучатися або перший найменший, або останній найбільший елемент і добавлятися відповідно або в кінець, або на початок результуючого списку. Операцією РСГХ легко відділити найменший елемент з початку перетвореного вхідного списку, але потрібно мати додаткове правило для вставки його в кінець результуючого списку. Аналогічно, легко вставити новий найбільший елемент на початок результуючого списку, але потрібно мати додаткове правило для видалення його з кінця перетвореного вхідного списку.

Обидва способи сортування потребуватимуть трьох основних рекурсивних правил:

- одне з внутрішніх правил реалізує один етап або методу бульбашки, або методу камінця;
- у випадку методу камінця друге внутрішнє правило реалізує ідентифікацію і видалення останнього елемента у списку. А у випадку методу бульбашки друге внутрішнє правило реалізує вставку нового елемента в кінець результуючого списку;
- зовнішнє правило реалізує повторюване виконання двох внутрішніх правил і формування результуючого списку.

Порівняння і обміни доцільно реалізувати окремим правилом, оскільки в основі цієї операції лежить диз’юнкція. Алгоритм методу камінця передбачає рух по списку від початку до кінця, тому відповідне правило реалізує повторення основних дій на ПХР. Алгоритм методу бульбашки передбачає рух по списку від кінця до початку, тому відповідне правило реалізує повторення основних дій на ЗХР. Для обох методів зовнішнє правило передбачає повторення основних дій лише на ПХР, але воно відрізнятиметься для кожного з них. Програмна реалізація і цільові запити можуть мати вигляд:

```
predicates
    obmin(integer, integer, integer, integer)
    kamin_vn(list_i, list_i)
    vydal_kinets(integer, list_i, list_i)
    kamin_zov(list_i, list_i, list_i)
    bulb_vn(list_i, list_i)
    vstavka_kinets(integer, list_i, list_i)
    bulb_zov(list_i, list_i, list_i)
```

```
clauses
    obmin(X, Y, A, B) :- X<=Y, !, A=X, B=Y; A=Y, B=X.

    kamin_vn([X], [X]) :- !.
    kamin_vn([X, Y|T], [A|T1]) :- !, obmin(X, Y, A, B), kamin_vn([B|T], T1).

    vydal_kinets(X, [X], [ ]) :- !.
    vydal_kinets(X, [H|T], [H|T1]) :- !, vydal_ kinets(X, T, T1).

    kamin_zov([H|T], L, LL) :- !, kamin_vn([H|T], L1),
                                  vydal_kinets(X, L1, L2),
                                  kamin_zov(L2, [X|L], LL).
    kamin_zov([ ], L, L).

    bulb_vn([X], [X]) :- !.
    bulb_vn([X|T], [A, B|T1]) :- !, bulb_vn(T, [Y|T1]), obmin(X, Y, A, B).

    vstavka_kinets(X, [H|T], [H|T1]) :- !, vstavka_kinets(X, T, T1).
    vstavka_kinets(X, [ ], [X]).

    bulb_zov([H|T], L, LL) :- !, bulb_vn([H|T], [X|T1]),
                                  vstavka_kinets(X, L, L1),
                                  bulb_zov(T1, L1, LL).
    bulb_zov([ ], L, L).

GOAL:
kamin_zov([4, 2, 6, 3, 1, 5], [ ], L)
-------------------------
kamin_zov([4, 2, 6, 3, 1, 5], [ ], L)       /* тут відображена */
kamin_zov([2, 4, 3, 1, 5], [6], L)          /* послідовність підцілей, */
kamin_zov([2, 3, 1, 4], [5, 6], L)          /* перетворення вхідного */
kamin_zov([2, 1, 3], [4, 5, 6], L)          /* списку та формування */
kamin_zov([1, 2], [3, 4, 5, 6], L)          /* результуючого списку */
kamin_zov([1], [2, 3, 4, 5, 6], L)
kamin_zov([ ], [1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6])        /* дія ТЗР */
-------------------------
L=[1, 2, 3, 4, 5, 6]                        /* остаточний результат */

bulb_zov([4, 2, 6, 3, 1, 5], [ ], L)
-------------------------
bulb_zov([4, 2, 6, 3, 1, 5], [ ], L)        /* тут відображена */
bulb_zov([4, 2, 6, 3, 5], [1], L)           /* послідовність підцілей, */
bulb_zov([4, 3, 6, 5], [1, 2], L)           /* перетворення вхідного */
bulb_zov([4, 5, 6], [1, 2, 3], L)           /* списку та формування */
```

```
bulb_zov[5, 6], [1, 2, 3, 4], L)          /* результуючого списку */
bulb_zov([6], [1, 2, 3, 4, 5], L)
bulb_zov ([ ], [1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6])          /* дія ТЗР */
-------------------------
L=[1, 2, 3, 4, 5, 6]                    /* остаточний результат */
```

Компоновка даних у список

Часто виникає потреба оформити у список дані, що задані в програмі твердженнями СБД. Наприклад, сформувати список власників книг або список самих книг тощо. Якщо за умовою задачі таких тверджень багато, то компоновка їх у список за допомогою розглянутої вище операції формування списку потребуватиме значних затрат часу на введення даних з клавіатури.

Для компоновки даних у список у мові Prolog використовується стандартне твердження ***findall***, декларація якого наступна:

findall(Variable_name, Predicate_expression, List_name)

Параметри твердження мають такий зміст:

- перший параметр ***Variable_name*** – ідентифікатор змінної, що буде позначати той із параметрів предикатного виразу ***Predicate_expression***, за яким формується список;
- другий параметр ***Predicate_expression*** – предикатний вираз тверджень СБД, у якому обов'язково присутня на відповідному місці в якості параметра змінна ***Variable_name;***
- третій параметр ***List_name*** – ідентифікатор змінної результуючого списку. Ця змінна відноситься до спискового домену, який обов'язково має бути оголошеним у розділі ***domains***.

У якості прикладу розглянемо задачу формування списку всіх книг, що є у власності деяких людей.

```
domains
    knyga = kn(avtor, nazva, rik)
    avtor, nazva = string
    rik = integer
    list_kn=knyga*          list_vl=symbol*          list_na=nazva*
predicates
    vlasnyk(symbol, knyga)
clauses
    vlasnyk(kolya, kn("Народ", "Колобок", 2016)).
    vlasnyk(kolya, kn("Дж.К.Ролінг", "Гарі Потер.
                    Філософський камінь", 2010)).
    vlasnyk(sofiya, kn("Дж.К.Ролінг", "Гарі Потер.
                    Таємна кімната", 2011)).
    vlasnyk(petro, kn("Дж.К.Ролінг", "Гарі Потер.
                    Келих вогню", 2012)).
```

GOAL:
findall(X, vlasnyk(X, _), L)

L=[kolya, kolya, sofiya, petro]

findall(X, vlasnyk(_, X), L)

L=[kn(“Народ”, “Колобок”, 2016)),
kn(“Дж.К.Ролінг”, “Гарі Потер. Філософський камінь”, 2010)),
kn(“Дж.К.Ролінг”, “Гарі Потер. Таємна кімната”, 2011)),
kn(“Дж.К.Ролінг”, “Гарі Потер. Келих вогню”, 2012))]

findall(X, vlasnyk(_, kn(_, X, _)), L)

L=[“Колобок”,
“Гарі Потер. Філософський камінь”,
“Гарі Потер. Таємна кімната”,
“Гарі Потер. Келих вогню”]

РОЗДІЛ 4. РЯДКИ СИМВОЛІВ

Рядки символів відносяться до стандартного доменного типу ***string***. У мові Prolog рядки задаються у вигляді послідовностей ASCII-символів, що обмежуються парою символів подвійні лапки (**“ ”**). Наприклад, **“ABCD”** та **“\65\66\67\68”** – це два еквівалентні подання однакових рядків. Тут символи другого рядка задані через їх десяткові коди.

Подібно до мов C/C++ у пам’яті рядки є послідовностями символів, які в кінці обмежуються керуючим символом **‘\0’**, і можуть досягати розміру 64 кБ. Для введення рядків у мові Prolog використовується стандартний предикат ***readln(string)***. При читанні таким твердженням через буфер вводу-виводу може бути введено рядок до 250 символів.

4.1. Стандартні твердження мови Prolog для обробки рядків

Визначення довжини рядка

Довжина рядка, що дорівнює кількості його символів, може бути визначена стандартним твердженням

str_len (string, integer).

Перший параметр – рядок символів – є вхідним параметром і має бути заданим значенням, другий параметр – кількість символів у рядку – може бути вільною змінною. Наприклад:

```
GOAL:
str_len(“ABCD”, X)
-------------------------
X=4

str_len(“”, 0)
-------------------------
Yes
```

Відділення символьного префікса

Символьний префікс – це один перший символ рядка. Оскільки це окремий символ, то він має тип ***char***. Символьний префікс подібний до голови списку. Відділення префіксу реалізується стандартним твердженням

frontchar (string, char, string).

Тут перший параметр – це базовий рядок, другий параметр – це власне префікс, третій параметр – залишок (хвіст) рядка.

Операція відділення символьного префікса аналогічна операції РСГХ. Вона не може застосовуватися до порожніх рядків (**“”**). Також ця операція може виконувати обернену дію – приєднувати в якості префікса новий символ до рядка.

Зауваження. Тут і надалі при виведенні результуючих значень, що є символами типу ***char*** та рядків типу ***string***, обмежуючі парні символи апостроф (**‘ ‘**) та подвійні лапки (**“ ”**) не відображатимуться. Це природньо, адже вони не є складовою частиною значень типу ***char*** або ***string***. Якщо потрібно задати сивол апостроф (**‘**), то використовується зображення **’’’** або **‘\39’**. Якщо потрібно задати рядок із сивола подвійні лапки (**“**), то використовується зображення **”””** або **”\34”.**

Приклади:

GOAL:
frontchar(“ABCD”, X, Y)

X=A, Y=BCD

frontchar(X, ‘A’, “BCD”)

X=ABCD

frontchar(“A”, X, Y)

X=A, Y= */* тут змінна Y є порожнім рядком */*

frontchar(“”, X, Y)

No

frontchar(“ABCD”,’A’,”BCD”)

Yes

Відділення префіксного підрядка

Префіксний підрядок – це послідовність символів заданої довжини на початку рядка, тобто це теж рядок типу ***string***. Очевидно, що довжина префіксного підрядка не може перевищувати довжини базового рядка. Відділення підрядка заданої довжини реалізується стандартним твердженням

frontstr (integer, string, string, string).

Тут перший параметр – довжина підрядка, другий параметр – базовий рядок, третій параметр – це префіксний підрядок, четвертий параметр – залишок рядка.

На відміну від операції по відділенню символьного префікса твердження ***frontstr*** виконує виключно відділення підрядка. Тобто перші два параметри завжди є вхідним і повинні бути заданими значеннями, а два

останніх параметри завжди є вихідними і мають бути вільними змінними. Наприклад:

```
GOAL:
frontstr(2, ”ABCD”, X, Y)
-------------------------
X=AB, Y=CD

frontstr(0, ”ABCD”, X, Y)
-------------------------
X= , Y=ABCD                    /* тут змінна X є порожнім рядком */

frontstr(4, ”ABCD”, X, Y)
-------------------------
X=ABCD, Y=                     /* тут змінна Y є порожнім рядком */

frontstr(-1, ”ABCD”, X, Y)
-------------------------
No

frontstr(5, ”ABCD”, X, Y)
-------------------------
No
```

Конкатенація (з’єднання) рядків

Конкатенація рядків подібно до списків передбачає приєднання другого рядка в кінець першого. З’єднання рядків реалізується стандартним твердженням

concat (string, string, string).

Тут перші два параметри – рядки, що з’єднуються, третій параметр – результат з’єднання.

Твердження ***concat*** може також виконувати обернені дії – встановити один із двох рядків, що в конкатенації з іншим заданим рядком дає відомий результат. Таким чином, вільною змінною може бути будь-який із параметрів, але не більше ніж один. Наприклад:

```
GOAL:
concat(“AB”,”CD”,X)
-------------------------
X=ABCD

concat(X, “CD”, “ABCD”)
-------------------------
X=AB

concat(“AB”, X, “ABCD”)
```

X=CD

concat(“AB”, ”CD”, “ABCD”)

Yes

Перевірка рядка символів, що є іменем мови Prolog

Ім’я мови Prolog – це особливий різновид рядків символів, що є ідентифікатором. Імена складаються з букв і (або) цифр, при цьому першим символом обов’язково має бути буква. Перевірка здійснюється стандартним твердженням

isname (string).

Твердження ***isname*** має зміст логічної умови і може набувати або істинного, або хибного значення. Наприклад:

GOAL:
isname(“abCD12”)

Yes

isname(“_1234”)

Yes

isname(“12abCD”)

No

isname(“ab#$CD”)

No

Відділення префіксного атома

Атом (англ. – **token**) – це особливий різновид рядків символів, який має окремий цілісний зміст. Окремими атомами можуть бути наступні послідовності символів:

1) кожен рядок із одного керуючого символа (**‘\0’, ‘\1’, …, ‘\31’**) за виключенням символа табуляції (**‘\9’**) – це окремий атом;
2) кожне правильно записане ціле число без знаку – це окремий атом;
3) кожне правильно записане дійсне число без знаку – це окремий атом;
4) кожне ім’я мови Prolog – це окремий атом;
5) кожен рядок із одного символа-знаку основної частини таблиці ASCII (такі як **‘!’, ‘@’, ‘#’, ‘$’, ‘+’, ‘-’, ‘*’, ‘/’, ‘(’, ‘)’** та ін.) – це окремий атом;

6) більшість символів альтернативної частини таблиці ASCII (**'\128', …, '\255'**), наприклад, спеціальні знаки та знаки псевдографіки, утворюють окремі атоми. Частина символів національних алфавітів утворює атоми подібні до імен мови Prolog, а решта із них – це окремі атоми;
7) символ пропуск (**' '**) не є атомом і відокремлює їх;
8) керуючий символ табуляції (**'\9'**) теж не є атомом і відокремлює їх.

Наприклад, рядок **" x1=(123-3.14)*(-456+789E-02) "** складається із 14 атомів (пропуски на початку і в кінці рядка ігноруються):

"x1", "=", "(", "123", "-", "3.14", ")", "*", "(", "-", "456", "+", "789E-02", ")".

Для відділення префіксного (першого за порядком) атома із рядка символів використовується стандартне твердження

fronttoken (string, string, string).

Тут перший параметр – базовий рядок, другий параметр – це префіксний атом, третій параметр – залишок рядка. Усі символи пропуски і символи табуляції, що передують префіксному атому, ігноруються.

Операція відділення префіксного атома подібна до операції відділення символьного префікса. Вона дає хибний результат у випадку рядків, що не містять жодного атома, тобто складаються лише із символів пропусків чи символів табуляції, або порожніх рядків. Ця операція також може виконувати обернену дію – приєднувати в якості префікса новий атом до рядка. Але при цьому відокремлюючі символи пропуски чи символи табуляції не добавляються. Наприклад:

```
GOAL:
fronttoken("AB12CD     ", X, Y)
-------------------------
X=AB12CD, Y=                /* тут Y є рядком із п'яти пропусків*/

fronttoken("123.456E+02", X, Y)
-------------------------
X=123.456E+02, Y=           /* тут змінна Y є порожнім рядком, */
                            /* а змінна X є рядком "123.456E+02" */

fronttoken("AB     CD", X, Y)
-------------------------
X=AB, Y=     CD

fronttoken("     $AB12CD", X, Y)
-------------------------
X=$, Y=AB12CD

fronttoken(X, "AB", "CD")
-------------------------
```

X=ABCD

fronttoken(“ ”, X, Y)

No

Перетворення типів з використанням рядків

- **Заміна великих і малих букв** реалізується стандартним твердженням

upper_lower (string, string).

Перетворення може виконуватися у різних напрямах. Якщо в першому параметрі є великі англійські букви, то в другому параметрі вони замінюються на відповідні малі букви, а решта символів залишаються без змін. Якщо в другому параметрі є малі англійські букви, то в першому параметрі вони замінюються на відповідні великі букви, а решта символів залишаються без змін. Наприклад:

GOAL:
upper_lower(“ABcd12$#”, X)

X=abcd12$#

upper_lower(X, “ABcd12$#”)

X=ABCD12$#

upper_lower(“ABCD12$#”,”ABCD12$#”)

Yes

upper_lower(“abcd12$#”,”abcd12$#”)

Yes

upper_lower(“ABcd12$#”,”ABcd12$#”)

Yes

- **Перетворення рядка цифр у відповідне ціле число і навпаки** реалізується стандартним твердженням

str_int (string, integer).

Перетворення може виконуватися в різних напрямах. Наприклад:

GOAL:
str_int(“12345”, X)

X=12345 */* тут змінна X є цілим числом 12345 */*

str_int(X, -12345)

X=-12345 */* тут змінна X є рядком "-12345" */*

str_int("123456", X)

No */* значення 123456 не є **integer** */*

- **Перетворення рядка цифр у відповідне дійсне число і навпаки** реалізується стандартним твердженням
  str_real (string, real).

Перетворення може виконуватися в різних напрямах. Наприклад:

GOAL:
str_real("1.23", X)

X=1.23 */* тут змінна X є дійсним числом 1.23 */*

str_real(X, -1.23)

X=-1.23 */* тут змінна X є рядком "-1.23" */*

str_real("123456", X)

X=123456 */* значення 123456 є **real** */*

str_real("-1.23456E+03", X)

X=-1234.56

str_real("-1.23.45.6", X)

No */* некоректне подання дійсного числа */*

- **Перетворення ASCII-символа в рядок з одного символа і навпаки** реалізується стандартним твердженням
  str_char (string, char).

Перетворення може виконуватися в різних напрямах. Наприклад:

GOAL:
str_char("A", X)

X=A */* тут змінна X є символом 'A' */*

str_char(X, ‘A’)

X=A */* тут змінна X є рядком “A” */*

str_char(“A”, ‘A’)

Yes

str_char(“A”, ‘B’)

No

- **Перетворення ASCII-символа у відповідний числовий код і навпаки** реалізується стандартним твердженням
char_int (char, integer).

Перетворення може виконуватися в різних напрямах. Наприклад:
GOAL:
char_int(‘A’, X)

X=65

char_int(X, 65)

X=A */* тут змінна X є символом ‘A’ */*

char_int(‘B’, 65)

No

4.2. Операції, що визначені користувачем, для обробки рядків

Розглянуті вище операції реалізуються стандартними твердженнями. Окрім них може виконуватись багато інших дій, які реалізуються користувацькими правилами.

- **Вставка нового фрагмента в задану позицію рядка** може бути реалізована або за допомогою стандартних тверджень відділення префіксного підрядка та конкатенації, або за допомогою рекурсивного правила. Перший варіант базується на використанні стандартного твердження ***frontstr***, тому результат буде істинним, якщо позиція вставки не виходить за межі базового рядка. Рекурсивне правило дозволяє реалізовувати більш загальний алгоритм: якщо задана позиція не перевищує **1**, то вставка виконується на початок рядка; якщо задана позиція

перевищує реальну довжину базового рядка, то вставка виконується в кінець рядка. В якості прикладу пропонується обидва варіанти:

```
predicates
    vst_poz_1(integer, string, string, string)
    vst_poz_2(integer, string, string, string)
clauses
    vst_poz_1(P, Fr, S_in, S_out) :- P1=P-1, frontstr(P1, S_in, S1, S2),
                                     concat(S1, Fr, S3), concat(S3, S2, S_out).

    vst_poz_2(P, Fr, S_in, S_out) :- P>1, frontchar(S_in, H, S_in1), !,
                                     P1=P-1, vst_poz_2(P1, Fr, S_in1, S_out1),
                                     frontchar(S_out, H, S_out1).
    vst_poz_2(_, Fr, S_in, S_out) :- concat(Fr, S_in, S_out).

GOAL:
vst_poz_1(3, “CD”, ”ABEF”, X)
-------------------------
X=ABCDEF

vst_poz_1(-10, “CD”, ”ABEF”, X)
-------------------------
No

vst_poz_1(10, “CD”, ”ABEF”, X)
-------------------------
No

vst_poz_2(3, “CD”, ”ABEF”, X)
-------------------------
X=ABCDEF

vst_poz_2(-10, “CD”, ”ABEF”, X)
-------------------------
X=CDABEF

vst_poz_2(10, “CD”, ”ABEF”, X)
-------------------------
X=ABEFCD
```

- **Видалення фрагмента із заданої позиції рядка** теж може бути реалізоване або за допомогою стандартних тверджень відділення префіксного підрядка та конкатенації, або за допомогою рекурсивного правила. В обох випадках реалізації результат операції буде істинним,

якщо позиція видалення і довжина фрагмента відповідають реальній довжині базового рядка. В якості прикладу пропонується обидва варіанти:

```
predicates
    vydal_poz_1(integer, integer, string, string)
    vydal_poz_2(integer, integer, string, string)
clauses
    vydal_poz_1(P, L, S_in, S_out) :- P1=P-1, frontstr(P1, S_in, S1, S2),
                                      frontstr(L, S2, _, S3), concat(S1, S3, S_out).

    vydal_poz_2(P, L, S_in, S_out) :- P>1, frontchar(S_in, H, S_in1), !,
                                      P1=P-1, vydal_poz_2(P1, L, S_in1, S_out1),
                                      frontchar(S_out, H, S_out1).
    vydal_poz_2(1, L, S_in, S_out) :- frontstr(L, S_in, _, S_out).

GOAL:
vydal_poz_1(3, 2, "ABCDEF", X)
-------------------------
X=ABEF

vydal_poz_2(2, 4, "ABCDEF", X)
-------------------------
X=AF

vst_poz_1(-10, 2, "ABCDEF", X)
-------------------------
No

vst_poz_2(2, 10, "ABCDEF", X)
-------------------------
No
```

- **Реверс рядка** формує новий рядок, в якому порядок символів протилежний до заданого. Цю операцію можна реалізувати рекурсивними правилами з повторенням основної дії як на ПХР, так і на ЗХР. У першому випадку кожен черговий символьний префікс вхідного рядка приєднується в якості префікса до результуючого рядка. У другому випадку відділені на прямому ході символьні префікси на зворотньому ході будуть приєднуватися в кінець результуючого рядка. В якості прикладу пропонується обидва варіанти.

```
predicates
    revers_str1(string, string, string)
    revers_str2(string, string)
clauses
    revers_str1(S_in, S_out, SS_out) :- frontchar(S_in, H, S_in1), !,
                                        frontchar(S_out1, H, S_out),
                                        revers_str1(S_in1, S_out1, SS_out).
    revers_str1("", S, S).

    revers_str2(S_in, S_out) :- frontchar(S_in, H, S_in1), !,
                                revers_str2(S_in1, S_out1),
                                str_char(SH, H), concat(S_out1, SH, S_out).
    revers_str2("", "").

GOAL:
revers_str1("ABCD", "", X)
-------------------------
X=DCBA

revers_str2("DCBA", X)
-------------------------
X=ABCD
```

- **Перетворення рядка в список символів і навпаки** (із збереженням порядку елементів). Якщо заданим є рядок символів, то застосовується один із варіантів реалізації правила, в якому на ПХР виконується операція відділення символьного префікса, а на ЗХР виконується операція РСГХ для приєднання до списку нового елемента. Якщо заданим є список символів, то застосовується другий варіант реалізації правила, в якому на ПХР виконується операція РСГХ для одержання чергового елемента списку, а на ЗХР виконується операція приєднання символьного префікса. Приклад програмної реалізації:

```
domains
    list_ch=char*
predicates
    str_list_ch(string, list_ch)
clauses
    str_list_ch(S, [H|T]) :- frontchar(S, H, S1), !, str_list_ch(S1, T).
    str_list_ch(S, [H|T]) :- !, str_list_ch(S1, T), frontchar(S, H, S1).
    str_list_ch("", [ ]).

GOAL:
str_list_ch("ABCD", L)
-------------------------
```

L=[‘A’,’B’,’C’,’D’]

str_list_ch(S, [‘A’,’B’,’C’,’D’])

S=“ABCD”

- **Перетворення рядка символів у список атомів** здійснюється аналогічно до попередньої операції, але з використанням твердження ***fronttoken*** замість ***frontchar***. Варто відзначити, що спроба реалізувати обернену дію – перетворити отриманий список атомів у назад у рядок символів – може не дати початковий варіант рядка. Це пояснюється тим, що символи пропуски і табуляції не є атомами і назад у рядок символів не повернуться. Окрім цього короткі атоми можуть з’єднатися і утворити довші атоми. Приклад програмної реалізації:

```
domains
    list_tok=string*
predicates
    str_list_tok(string, list_t)
clauses
    str_list_tok(S, [H|T]) :- fronttoken(S, H, S1), !, str_list_tok(S1, T).
    str_list_tok(S, [H|T]) :- !, str_list_tok(S1, T), fronttoken(S, H, S1).
    str_list_tok(“”, [ ]) :- !.
    str_list_tok(_, [ ]).

GOAL:
str_list_tok(”X1=Y2*(123+456.789E-03)”, L)
---------------------------
L=[”X1”, ”=”, ”Y2”, ”*”, ”(”, ”123”, ”+”, ”456.789E-03”, ”)”]

str_list_tok(S, [”X1”,”=”,”Y2”,”*”,”(”,”123”,”+”,”456.789E-03”,”)”])
---------------------------
S=X1=Y2*(123+456.789E-03)

str_list_tok(”   12   34AB   CD56$#   ”, L)
---------------------------
L=[”12”, ”34”, ”AB”, ”CD56”, ”$”, ”#”]

str_list_tok(S, [”12”, ”34”, ”AB”, ”CD56”, ”$”, ”#”])
---------------------------
S=1234ABCD56$#

str_list_tok(”1234ABCD56$#”, L)
---------------------------
L=[”1234”, ”ABCD56”, ”$”, ”#”]
```

- **Перетворення рядка символів у список атомів з функторами** є розширенням розглянутої вище операції. Доцільність такої модифікації легко ілюструється прикладом перетворення рядка *"X1=Y2*(123+456.789E-03)"* у список атомів. Хоча в результаті розділення на атоми усі вони є рядками символів, проте мають різний зміст: *"X1", "Y2"* – ідентифікатори; *"=", "*", "(", "+", ")"* – знаки операцій; *"123"* – ціле число; *"456.789E-03"* – дійсне число. Таким чином, цікаво було б реалізувати таку операцію з одночасним наданням додаткової ознаки кожному атому. Для цього можна використати **функтори** подібно до складених доменів з альтернативами. Саме функтори виконуватимуть функцію додаткової ознаки. Звичайно, постане потреба в додатковому правилі, яке обиратиме відповідну ознаку для кожного атома. При цьому важливим є порядок визначення окремих категорій атомів: першими доцільно визначати цілі числа, а потім – дійсні числа; перевіряти ідентифікатори як імена мови Prolog можна перед числами або після них; останніми по залишковому принципу ідентифікуються символи – знаки операцій (їх теж можна розділити за пріоритетами). Також доцільно передбачити альтернативу для випадку некоректного атома. Приклад програмної реалізації:

```
domains
    tok_f = n(string); i(integer); r(real); c(char); e(string)
    /* n – name; i – integer; r – real; c – char; e – error */
    list_tok_f = tok_f*
predicates
    vybir(string, tok_f)
    str_list_tok_f(string, list_tok_f)
clauses
    vybir(S, X) :- isname(S), !, X = n(S);
                   str_int(S, Y), !, X = i(Y);
                   str_real(S, Y), !, X = r(Y);
                   str_char(S, Y), !, X = c(Y);
                   X = e(S).
    str_list_tok_f(S, [H|T]) :- fronttoken(S, Tok, S1), !, vybir(Tok, H),
                                str_list_tok_f(S1, T).
    str_list_tok_f(_, [ ]).

GOAL:
str_list_tok_f("X1=Y2*(123+456.789E-03)▲▼", L)
-------------------------
L=[n("X1"), c('='), n("Y2"), c('*'), c('('), i(123), c('+'), r(456.789E-03),
    c(')'), e("▲"), e("▼")]
```

РОЗДІЛ 5. ДИНАМІЧНІ БАЗИ ДАНИХ

Твердження програми, що записані в розділі ***clauses***, утворюють так звану статичну базу даних (СБД). Ні їх зміст, ні порядок, ні кількість не можуть бути змінені в процесі виконання програми. Якщо задача потребує модифікації самих тверджень у процесі виконання програми, то в мові Prolog використовуються так звані **динамічні бази даних** (ДБД).

5.1. Оголошення предикатів ДБД

У випадку використання твердженнь ДБД потрібно оголошувати відповідні предикати подібно до предикатів СБД. Це здійснюється в окремому розділі програми – ***database***. Предикати ДБД декларуються так само, як і звичайні предикати. Єдине, що вимагається, щоб терми відрізнялись від термів аналогічних предикатів з розділу ***predicates***. Наприклад:

```
. . .
database
    dlikes(symbol, symbol)
    dvlasnyk(symbol, knyga)
```

Одна динамічна база може містити твердження різних предикатів. При цьому в пам'яті твердження спільного предиката автоматично групуються в один блок. Порядок окремих блоків тверджень різних предикатів у пам'яті не суттєвий.

Якщо в пам'яті має оброблятися декілька різних ДБД, які жодним чином не пов'язані між собою, то використовують так звані **іменовані динамічні бази даних**. У цьому випадку предикати кожної окремої бази декларуються в окремому розділі ***database***, якому надається унікальне ім'я. Ім'я бази слідує відразу після слова ***database*** і відокремлюється від нього символом мінус (-). При цьому декілька різних предикатів можуть відноситися до однієї іменованої бази. Наприклад:

```
. . .
database – dbd1
    dlikes_1(symbol, napiy)
    dlikes_2(symbol, frukt)
    dlikes_3(symbol, colir)
database – dbd2
    dvlasnyk(symbol, rich)
```

Для роботи з твердженням ДБД використовують як стандартні предикати, так і правила, визначені користувачем.

5.2. Стандартні твердження мови Prolog для роботи з ДБД

Для роботи з твердженнями ДБД використовують як стандартні предикати, так і правила, що визначені користувачем. Стандартні інструменти мови Prolog забезпечують виконання основних базових операцій по формуванню ДБД. Користувацькими правилами можна реалізовувати більш складніші дії та забезпечувати гнучкість взаємодії із кінцевим користувачем.

- **Введення нового твердження на початок ДБД** реалізується стандартним твердженням

asserta (твердження [, ім'я ДБД]).

Тут перший параметр – твердження, що добавляється в базу; необов'язковий другий параметр – це ім'я бази, якщо вона іменована. Наприклад:

```
GOAL:
asserta(dlikes(kolya, yabluka)), asserta(dlikes(olya, tistechka)),
dlikes(X, Y)
------------------------
X = olya, Y = tistechka          /* друге введене твердження */
X = kolya, Y = yabluka           /* у ДБД виявилося першим */

asserta(dvlasnyk(kolya, knyga("Народ", "Колобок", 2016)), dbd2),
asserta(dvlasnyk(sofiya, transp(avto, "Fiat 500", 2012, red)), dbd2),
dvlasnyk(X, Y)
------------------------
X = sofiya, Y = transp(avto, "Fiat 500", 2012, red)
X = kolya, Y = knyga("Народ", "Колобок", 2016)
```

- **Введення нового твердження в кінець ДБД** реалізується стандартним твердженням

assertz (твердження [, ім'я ДБД]).

Тут перший параметр – твердження, що добавляється в базу; необов'язковий другий параметр – це ім'я бази, якщо вона іменована. Наприклад:

```
GOAL:
assertz(dlikes(kolya, yabluka)), assertz(dlikes(olya, tistechka)),
dlikes(X, Y)
------------------------
X = kolya, Y = yabluka           /* перше введене твердження */
X = olya, Y = tistechka          /* у ДБД виявилося першим */

assertz(dvlasnyk(kolya, knyga("Народ", "Колобок", 2016)), dbd2),
assertz(dvlasnyk(sofiya, transp(avto, "Fiat 500", 2012, red)), dbd2),
```

dvlasnyk(X, Y).

X = kolya, Y = knyga("Народ", "Колобок", 2016)

X = sofiya, Y = transp(avto, "Fiat 500", 2012, red)

- **Видалення тверджень із ДБД** реалізується стандартним твердженням

retract (твердження [, ім'я ДБД]).

Тут перший параметр – твердження, що вилучається. При цьому видаляється перше із можливо декількох тверджень, що задовольняють умові вибору. Необов'язковий другий параметр – це ім'я бази, якщо вона іменована. Наприклад:

GOAL:

asserta(dlikes(kolya, yabluka)), asserta(dlikes(olya, tistechka)),

assertz(dlikes(kolya, morozyvo)), assertz(dlikes(sofiya, morozyvo)),

dlikes(X, Y)

X = olya, Y = tistechka */* ДБД містить */*

X = kolya, Y = yabluka */* чотири твердження */*

X = kolya, Y = morozyvo */* у відповідному порядку */*

X = sofiya, Y = morozyvo

retract(dlikes(kolya,_)), dlikes(X, Y)

--- */* було видалено */*

X = olya, Y = tistechka */* перше із тверджень, */*

X = kolya, Y = morozyvo */* параметром якого */*

X = sofiya, Y = morozyvo */* є символічне ім'я kolya */*

retract(dvlasnyk(_, _), dbd2), 1=2; 1=1

--- */* методом ВПН реалізації */*

Yes */* повторень було вилучено усі */*

/ твердження із бази dbd2 */*

- **Збереження ДБД із пам'яті у файл на диску** реалізується стандартним твердженням

save (string).

Тут параметр домену ***string*** – ім'я дискового файла у форматі операційної системи MS-DOS. Якщо вказаний файл був відсутній на диску в поточному каталозі (папці), то він створюється; якщо вказаний файл вже існував, то він перезаписується. Твердження ДБД записуються у файл в текстовому форматі в тому самому порядку, як вони були сформовані в пам'яті.

- **Доповнення ДБД у пам'яті твердженнями із дискового файла** реалізується стандартним твердженням

consult (string).

Тут параметр домену ***string*** – ім'я дискового файла у форматі операційної системи MS-DOS. Вказаний файл має бути наявним на диску у поточному каталозі (папці). Очевидно, що у розділі ***database*** програми мають бути оголошенні відповідні предикати ДБД для тверджень із файла. Усі твердження із файла в тому самому порядку доповнюють існуючу в пам'яті базу даних в її кінці. Якщо ДБД містить твердження різних предикатів, то доповнення новими твердженнями із файла передбачає їх взаємне групування у пам'яті. Наприклад:

```
GOAL:
asserta(dlikes(kolya, yabluka)), asserta(dlikes(olya, tistechka)),
assertz(dlikes(kolya, morozyvo)), assertz(dlikes(sofiya, morozyvo)),
                                  /* у ДБД в пам'яті – 4 твердження, */
save("file1.dbd"),                /* у файлі збережено 4 твердження, */
consult("file1.dbd"),             /* у ДБД стало 8=4+4 тверджень */
dlikes(X, Y)
-------------------------
X = olya, Y = tistechka
X = kolya, Y = yabluka
X = kolya, Y = morozyvo
X = sofiya, Y = morozyvo
X = olya, Y = tistechka
X = kolya, Y = yabluka
X = kolya, Y = morozyvo
X = sofiya, Y = morozyvo
```

Розглянемо в якості прикладу комплексну задачу по роботі з ДБД:

- сформувати ДБД із тверджень програми про любителів *смаколиків* зі збереженням взаємного порядку;
- доповнити 5 нових тверджень, що вводяться з клавіатури;
- видалити всі твердження про любителів *яблук*;
- зберегти базу на диск;
- окремо зберегти на диск в інший файл частину бази з інформацією про вподобання *Колі*.

Програмана реалізація задачі може мати вигляд:

```
predicates
    likes(symbol, symbol)
    stvorennya
    dopovnennya(integer)
    vydalennya(symbol)
    vybir(symbol)
database
    dlikes(symbol, symbol)
```

```
clauses
    likes(sofiya, morozyvo).
    likes(kolya, yabluka).
    likes(kolya, morozyvo).
    likes(olya, tistechka).
    likes(andriy, cukerky).
    likes(andriy, yabluka).
    likes(vita, X) :- likes(olya, X).
    likes(tanya, X) :- likes(sofiya, X), likes(kolya, X).
    likes(petro, X) :- likes(olya, X); likes(andriy, X).
    likes(oksana, X) :- likes(petro, X), not(likes(kolya, X)).

    stvorennya :- likes(X, Y), assertz(dlikes(X, Y)), 1=2; 1=1.

    dopovnennya(N) :- N>0, !, N1=N-1,
                      write("введіть ім'я:"), readln(X),
                      write("введіть вподобання:"), readln(Y),
                      assertz(dlikes(X, Y)), dopovnennya(N1); 1=1.

    vydalennya(Y) :- retract(dlikes(_, Y)), 1=2; 1=1.

    vybir(X) :- dlikes(X1, Y), X1<>X, retract(dlikes(X1, Y)), 1=2; 1=1.

GOAL:
stvorennya, dopovnennya(5), vydalennya(yabluka), save("file_1.dbd"),
vybir(kolya), save("file_2.dbd")
-------------------------
Yes
```

Якщо потрібно перегрупувати твердження бази або впорядкувати її за певною ознакою, то це можна здійснити за принципом вибору:

- у початковій базі даних здійснюється пошук твердження, що є найменшим (найбільшим) за даною ознакою;
- знайдене твердження доповнюється в нову базу даних в кінець (на початок);
- із заданої бази даних видаляється знайдене твердження;
- послідовність попередніх кроків повторюється, поки початкова база не стане порожньою;
- для реалізації пошуку найменшого (найбільшого) за даною ознакою твердження доцільно використовувати рекурсивне правило.

РОЗДІЛ 6. ВЗАЄМОДІЯ З ДИСКОВИМИ ФАЙЛАМИ

6.1. Адресація файлового потоку вводу-виводу

На мові Prolog можна виконувати усі основні операції по обробці інформації, що зберігається у дискових файлах. Подібно до більшості мов програмування, наприклад C або C++, в мові Prolog розрізняють файли **текстові** та **бінарні**. Як і в інших мовах у програмі на мові Prolog використовують особливі логічні імена файлів, що представляють конкретні дискові файли. У мові Pascal такі елементи програми називають файловими змінними; у мові C такі елементи є вказівниками на спеціальну структуру FILE; у мові C++ такі елементи – це дескриптори файлових потоків тощо. У мові Prolog цю функцію виконують **файлові домени** або **логічні імена файлів**. Вони оголошуються в розділі ***domains*** через стандартний домен ***file*** відповідно до синтаксичного правила:

file = логічне ім'я1; логічне ім'я2;…; логічне ім'яN

Наприклад:

```
domains
    file = f1; f2; f3
```

У поточний момент роботи програми кожне окреме логічне ім'я може представляти один дисковий файл. Ім'я може бути звільнене і заново переназначене, тобто зв'язане з іншим дисковим файлом.

У більшості мов програмування допускається одночасна обробка багатьох файлів; усі операції читання і запису здійснюються через файловий потік вводу-виводу. У мові Prolog в поточний момент може виконуватись операція читання або запису лише з одним активним файлом. При цьому файловий потік вводу-виводу адресується на один із файлових пристроїв, що ідентифікується відповідним логічним іменем.

Для переадресації файлового потоку вводу (читання) на деякий файловий пристрій (файл) використовується стандартне твердження

readdevice (file).

Для переадресації файлового потоку виводу (запису) на деякий файловий пристрій (файл) використовується стандартне твердження

writedevice (file).

Серед логічних імен файлів у мові Prolog використовується ряд зарезервованих імен, що позначають стандартні файлові пристрої, зокрема:

- ***keyboard*** **–** стандартний файловий пристрій вводу клавіатура. Цей пристрій вважається активним по замовчуванню;
- ***screen*** **–** стандартний файловий пристрій виводу монітор. Цей пристрій вважається активним по замовчуванню;
- ***printer*** – друкуючий пристрій виводу, що ототожнюється з LPT-портом;
- ***com1*** **–** послідовний порт.

6.2. Стандартні твердження мови Prolog для роботи з файлами

Відкривання файлу і зв'язування з логічним файловим іменем

Взаємодія з дисковими файлами розпочинається з виконання операції по їх відкриванню в тому чи іншому режимі та зв'язуванню з логічним файловим іменем у програмі. У мові Prolog використовуються наступні основні режими відкривання файлу.

- **Відкривання файлу для запису** реалізується стандартним твердженням

openwrite (file, string).

Тут перший параметр – логічне файлове ім'я, другий параметр домену ***string*** – ім'я дискового файла у форматі операційної системи MS-DOS. Якщо вказаний файл відсутній на диску, то він створюється, якщо файл вже є, то його вміст очищається і файловий вказівник встановлюється на початок файлу. Наприклад:

openwrite(f1,"student.dat").

- **Відкривання файлу для читання** реалізується стандартним твердженням

openread (file, string).

Тут перший параметр – логічне файлове ім'я, другий параметр домену ***string*** – ім'я дискового файла у форматі операційної системи MS-DOS. Читати можна лише існуючий файл, при цьому файловий вказівник встановлюється на початок файлу. Наприклад:

openread(f1,"student.dat").

- **Відкривання файлу для модифікації** реалізується стандартним твердженням

openmodify (file, string).

Тут перший параметр – логічне файлове ім'я, другий параметр домену ***string*** – ім'я дискового файла у форматі операційної системи MS-DOS. Модифікація дозволяє одночасне виконання операцій і читання, і запису. Перезапис реалізується за принципом „на тому ж місці", тобто заміщення. Модифікувати можна лише існуючий файл, при цьому файловий вказівник встановлюється на початок файлу. Наприклад:

openmodify(f1,"student.dat").

- **Відкривання файлу для дозапису в кінець** реалізується стандартним твердженням

openappend (file, string).

Тут перший параметр – логічне файлове ім'я, другий параметр домену ***string*** – ім'я дискового файла у форматі операційної системи MS-DOS. Дозапис реалізується лише для існуючого файла, файловий вказівник

встановлюється в кінець файла. В цьому режимі можливі лише операції запису. Наприклад:

openappend(f1,"student.dat").

- **Закривання файлу**, відкритого в одному із режимів, реалізується стандартним твердженням

closefile (file).

Закривати раніше відкритий файл потрібно перед відкриванням його в якомусь іншому режимі, а також при завершенні роботи з ним. Це є неписане правило „якісного програмування". При закриванні буфер обміну файлового потоку вводу-виводу звільняється від даних останньої операції читання чи запису. Логічне ім'я файлу звільняється, воно може бути зв'язане або з іншим дисковим файлом, або з цим самим файлом у іншому режимі.

Читання і запис даних

Читання або запис виконується виключно в тому файлі, на який адресований файловий потік вводу-виводу твердженнями ***readdevice*** або ***writedevice.*** Файли, які відкриті у одному із перелічених режимів, вважаються текстовими. Внутрішні уніфікаційні підпрограми мови Prolog автоматично інтерпретують текст як дані того чи іншого типу або за його контекстом, або завдяки використанню стандартних тверджень для читання даних різних доменних типів.

- **Запис даних у текстовому форматі** реалізується стандартним твердженням ***write (список параметрів виводу).***

Непорожній список параметрів цього твердження може містити довільну їх кількість; параметри можуть бути різних доменних типів. При цьому всі елементи даних (цілі числа, дійсні числа, символи, рядки, складені домени, твердження тощо) утворюють потік ASCII-симолів.

- **Запис маркера кінця рядка або перехід на початок нового рядка** реалізується стандартним твердженням ***nl.***

Твердження ***nl*** дописує в поточну позицію файла особливий маркер кінця рядка – послідовність символів **"\13\10"**.

- **Читання даних різних доменних типів** реалізується одним із стандартних тверджень:
  - ***readint (integer)*** – для читання одного цілого числа;
  - ***readreal (real)*** – для читання одного дійсного числа;
  - ***readchar (char)*** – для читання одного ASCII-символа;
  - ***readln (string)*** – для читання рядка символів від поточної позиції файлового вказівника до маркера кінця рядка або кінця файла;

- ***readterm (ім'я домену, змінна)*** – для читання даних користувацьких доменів або тверджень. Тут перший параметр – ім'я користувацького домену. Якщо з файла має читатися твердження ДБД, то перший параметр – ім'я складеного домену з таким самим функтором, як у твердженні ДБД. Другий параметр – змінна відповідного доменного типу. Наприклад:

GOAL:
readterm(integer, X) */* readterm(integer, X) ⇔ readint(X) */*
5

X = 5

readterm(rich, X) */* rich – складений домен з альтернат. */*
knyga("Народ", "Колобок", 2016)

X = knyga("Народ", "Колобок", 2016)

readterm(rich, X) */* rich – складений домен з альтернат. */*
transp(velo, "Спорт-Трек", 2016, silver)

X = transp(velo, "Спорт-Трек", 2016, silver)

readterm(rich, X) */* rich – складений домен з альтернат. */*
colekc(znachky, 500)

X = colekc(znachky, 500)

Зауваження. Стандартні твердження вводу цілих чи дійсних чисел здійснюють читання лише одного елемента даних у поточному рядку. Тому числові дані у файлі слід записувати по одному елементу в рядку, інакше результат буде хибним. Наприклад:

12 34↵ */* це відкритий для читання */*
56 78↵ */* файл із цілими числами, */*
9 10↵ */* ↵ – маркер кінця рядка, */*
eof */* eof – маркер кінця файла */*

GOAL:
readint(X), readint(Y)

No

12↵ */* це інший відкритий для читання */*
34↵ */* файл із цілими числами, */*

```
56↵                    /* ↵ – маркер кінця рядка, */
eof                    /* eof – маркер кінця файла */
------------------------
GOAL:
readint(X), readint(Y), readint(Z)
------------------------
X = 12, Y = 34, Z = 56
```

- **Примусове очищення буфера обміну при операціях читання і запису** реалізується стандартним твердженням ***flush (file).***

Операції по позиціюванню файлового вказівника

Усі без винятку файли завершуються особливим маркером кінця файла – ***eof*** (від англ. **end of file**). При кожній операції читання чи запису відбувається зміщення файлового вказівника на відповідну кількість байтів. У випадку модифікації файла часто потрібно встановлювати вказівник у точно визначену позицію. Звичайно, що операції читання будуть коректними, якщо файловий вказівник не досягнув маркера ***eof***. Якщо ж запис у файл виконуватиметься від позиції маркера ***eof***, то такі дії призведуть до зміщення маркера ***eof*** на відповідну кількість байтів вперед.

- **Перевірка досягнення кінця файлу** реалізується стандартним твердженням ***eof (file).***

Параметром твердження є логічне ім'я файлу. Якщо файловий вказівник стоїть перед маркером ***eof***, то результат істинний, інакше – хибний.

- **Позиціювання файлового вказівника** у відкритому файлі реалізується стандартним твердженням

filepos (file, real, integer).

Тут перший параметр – логічне файлове ім'я; другий параметр дійсного типу – кількість байтів, на яку зміщується файлового вказівника; третій параметр, що є цілим числом, визначає точку відліку зміщення файлового вказівника: **0** – від початку файлу, **1** – від поточної позиції, **2** – від кінця файлу. Твердження ***filepos*** дозволяє не тільки переміщувати файловий вказівник, а й визначати його поточну позицію. Наприклад:

```
GOAL:
filepos(f1, 0, 10)          /* 10 байт від початку файлу */
------------------------
Yes

filepos(f1, 1, -10)         /* 10 байт назад від поточної позиції */
------------------------
Yes
```

```
filepos(f1, 0, X)          /* визначення позиції файлового */
-------------------------  /* вказівника від початку файлу, */
X = 10                     /* наприклад, зміщення рівне 10 байт */

filepos(f1, 2, 0)          /* перміщення файлового вказівника */
-------------------------  /* в кінець файла */
Yes

filepos(f1, 2, 0), filepos(f1, 0, X)
-------------------------
X = 1000                   /* X – розмір файла в байтах */
```

- **Задання розміру відкритого файла** передбачає відсікання хвоста файла від поточної позиції файлового вказівника і реалізується стандартним твердженням ***setfilesize (file).***

Параметром твердження є логічне ім'я файлу. При цьому маркер кінця файлу ***eof*** встановлюється в поточну позицію файлового вказівника.

Операції з файлами на рівні команд операційної системи

Із дисковими файлами можна виконувати ряд дій, які базуються на командах операційної системи. Такі операції проводяться із невідкритими (незв'язаними) файлами і передбачають використання імен дискових файлів у форматі операційної системи MS-DOS.

- **Перевірка наявності дискового файла** реалізується стандартним твердженням ***existfile (string)***.

- **Копіювання дискового файла** реалізується стандартним твердженням ***copyfile (string, string)***.

Параметрами є відповідно імена дискових файла-оригіналу і файла-копії.

- **Переіменування дискового файла** реалізується стандартним твердженням ***renamefile (string, string)***.

Параметрами є відповідно старе і нове імена дискового файла.

- **Видалення дискового файла** реалізується стандартним твердженням ***deletefile (string)***.

- **Перегляд змісту каталогу (папки)** реалізується стандартним твердженням ***dir (string, string, string)***.

Тут перший параметр – повне ім'я каталогу у форматі операційної системи MS-DOS або порожній рядок у випадку поточного каталогу; другий

параметр – шаблон (маска) файлів; третій параметр – вільна змінна, яка отримує ім'я обраного файлу із виведеного переліку.

6.3. Основні операції по обробці файлів у режимі послідовного доступу

Текстові файли часто обробляються в режимі послідовного доступу. Це означає, що доступ до кожного окремого елемента даних від поточної позиції файлового вказівника можливий шляхом послідовного перебору всіх проміжних елементів. Такий спосіб не потребує приведення всіх елементів даних до єдиного спільного розміру. Проте він накладає суттєві обмеження на модифікацію файла. Адже модифікація має проводитися за принципом „на тому ж місці". Це означає, що кожен новий елемент даних має бути приведений точно до такого ж розміру, як і старий, що заміщується.

Найчастіше до файлів послідовного доступу застосовують наступні операції: створення файлу і запис даних, читання або пошук даних у файлі, модифікаціяя файлу за принципом „на тому ж місці".

- **Створення файлу та запис даних** передбачає послідовне виконання таких дій:

- перевірка відсутності на диску файлу з обраним іменем;
- відкривання файлу в режимі запису;
- адресація файлового потоку виводу на відкритий файл;
- запис даних у файл;
- закривання файлу;
- переадресація потоку виводу на стандартний файловий пристрій виводу ***screen***.

В якості прикладу пропонується створення файлу і запис у нього восьми цілих чисел. Програмна реалізація може бути такою:

```
domains
    file=f
predicates
    zapys(integer)
    stvorennya(integer)
clauses
    stvorennya(N) :- write("Введіть ім'я файлу : "), readln(File_name),
                     not(existfile(File_name)), !, openwrite(f, File_name),
                     zapys(N), closefile(f),
                     writedevice(screen), write("кінець запису"), nl;
                     write("Такий файл вже існує"), nl.

    zapys(N) :- N>0, !, writedevice(screen),
                write("Введіть ціле число : "), readint(X),
```

```
        writedevice(f), write(X), nl, N1=N-1, zapys(N1); 1=1.

GOAL:
stvorennya(8)
-------------------------
Введіть ім'я файлу : cili.dat
Введіть ціле число : 3
Введіть ціле число : 0
Введіть ціле число : -4
Введіть ціле число : 5
Введіть ціле число : 0
Введіть ціле число : -2
Введіть ціле число : 0
Введіть ціле число : 1
кінець запису
```

- **Читання даних з файлу** передбачає послідовне виконання таких дій:

- перевірка наявності на диску файлу з обраним іменем;
- відкривання файлу в режимі читання;
- адресація файлового потоку вводу на відкритий файл;
- читання даних із файлу;
- закривання файлу;
- переадресація потоку вводу на стандартний файловий пристрій вводу ***keyboard***.

В якості прикладу пропонується визначити суми і кількості окремо від'ємних, окремо додатних і окремо нулів у раніше створеному файлі цілих чисел. Накопичення сум і кількостей зручно реалізувати рекурсивним правилом із повторюваними обчисленням на ЗХР. Звичайно, суму нулів шукати не потрібно. Програмна реалізація може бути такою:

```
domains
    file=f
predicates
    go
    chytannya(real, real, real, real, real)
    dodanok(integer, real, real)
    kilkist(integer, integer, integer, integer)
clauses
    go :- write("Введіть ім'я файлу : "), readln(File_name),
          existfile(File_name), !, openread(f, File_name), readdevice(f),
          chytannya(S_v, S_d, K_v, K_d, K_z),
          write("сума від'ємних = ", S_v, ",  їх кількість = ", K_v), nl,
          write("сума додатних = ", S_d, ",  їх кількість = ", K_d), nl,
          write(" кількість нулів = ", K_z), nl,
```

```
    closefile(f), readdevice(keyboard); write("Файл відсутній"), nl.

chytannya(S_v, S_d, K_v, K_d, K_z) :- not(eof(f)), !, readint(X),
                        chytannya(S_v1, S_d1, K_v1, K_d1, K_z1),
                        dodanok(X, V, D), kilk(X ,Kv, Kd, Kz),
                        S_v=S_v1+V, S_d=S_d1+D,
                        K_v=K_v1+Kv, K_d=K_d1+Kd, K_z=K_z1+Kz.
chytannya(0, 0, 0, 0, 0).

dodanok(X, V, D) :- X<0, !, V=X, D=0; X>0, !, V=0, D=X; V=0, D=0.

kilk(X, Kv, Kd, Kz) :- X<0, !, Kv=1, Kd=0, Kz=0;
                        X>0, !, Kv=0, Kd=1, Kz=0;
                                Kv=0, Kd=0, Kz=1.

GOAL:
go
-------------------------
Введіть ім'я файлу : cili.dat
сума від'ємних = -6,   їх кількість = 2
сума додатних = 9,   їх кількість = 3
кількість нулів = 3
```

- **Модифікація файлу** часто передбачає перезапис даних всередині файлу за принципом „на тому ж місці". Зрозуміло, що кожен новий елемент даних не може бути довшим від старого, на місці якого відбувається перезапис. З цією метою усі нові дані потрібно приводити до визначеного розміру старих даних. Оскільки файли послідовного доступу це фактично текстові файли, то приведення нових даних до потрібної довжини можна здійснювати шляхом доповнення незначущими пропусками зліва або справа у кожному рядку. При модифікації доцільно виконувати такі дії:

- перевірка наявності на диску файлу з обраним іменем;
- відкривання файлу в режимі модифікації;
- адресація файлових потоків вводу і виводу на відкритий файл;
- позиціювання файлового вказівника і читання даних, зміщення файлового вказівника і перезапис даних;
- закривання файлу;
- переадресація файлових потоків вводу і виводу на стандартні файлові пристрої ***keyboard*** та ***screen***.

В якості прикладу пропонується здійснити заміну на нулі всіх чисел між першим і останнім нулями у початковому файлі. При цьому гарантується, що у файлі всі цілі числа записані від початку рядка і точно по одному в рядку. У загальній схемі алгоритму можна виділити три основні фази:

- спочатку визначається позиція файлового вказівника після зчитування першого нуля;
- потім визначається позиція перед зчитуванням останнього нуля;
- в подальшому між цими двома позиціями повторюється складена операція з послідовності таких дій: читання цілого числа, формування рядка символів із нуля та потрібної кількості пропусків справа, повернення файлового вказівника на початок рядка, перезапис нового рядка.

Усі операції з повторенням реалізуються рекурсивними правилами. Програмна реалізація може бути такою:

```
domains
    file=f
predicates
    go
    poshuk1_0(real)
    poshuk2_0(real, real)
    read_write(real)
    if_0(integer, real, real, real)
    dopovn(integer, string, string)
clauses
    go :- write("Введіть ім'я файлу : "), readln(File_name),
          existfile(File_name), !, openmodify(f, File_name),
          readdevice(f), writedevice(f),
          poshuk1_0(P1), poshuk2_0(0, P2),
          filepos(f, P1, 0), read_write(P2), closefile(f),
          writedevice(screen), readdevice(keyboard);
          write("Файл відсутній"), nl.

    poshuk1_0(P) :- not(eof(f)), readint(X), X<>0, !, poshuk(P);
                    filepos(f1, P, 0).

    poshuk2_0(P1, P) :- filepos(f, P2, 0), not(eof(f)), !, readint(X),
                        if_0(X, P1, P2, P3), poshuk2_0(P3, P); P=P1.

    read_write(P) :- filepos(f, P1, 0), P1<P, !, readint(X), filepos(f, P2, 0),
                     Len=P2-P1-2, dopovn_0(Len, "0", Str0),
                     filepos(f, P1, 0), write(Str0), read_write(P); 1=1.

    dopovn(Len, S, SS) :- str_len(S, L), L<Len, !,
                          concat(S," ", S1), dopovn( Len, S1, SS);
                          concat(S, "\13\10", SS).

    if_0(X, P1, P2, P3) :- X<>0, !, P3=P1; P3=P2.
```

GOAL:
go

Введіть ім'я файлу : cili2.dat
Yes

Результат роботи із деяким файлом *cili2.dat* на диску можна проілюструвати наступним прикладом:

файл перед модифікацією	*файл після модифікації*
123↵	*123↵*
-12345↵	*-12345↵*
0↵	*0↵*
678↵	*0_ _↵*
-2↵	*0_↵*
0↵	*0↵*
-32768↵	*0_ _ _ _ _↵*
0↵	*0↵*
321↵	*321↵*
eof	*eof*

6.4. Обробка файлів у режимі прямого доступу

Якщо всі елементи даних в межах одного рядка у файлі приведені до єдиної спільної довжини, то такий файл можна обробляти в режимі прямого доступу. У цьому випадку кожен елемент даних – це блок байтів фіксованого розміру ***Size_el***. При цьому до розміру ***Size_el*** відноситься два байти, які реалізують маркер кінця рядка – послідовність символів **"\13\10"**. З файлами прямого доступу окрім раніше розглянутих операцій можна додатково виконувати такі дії: вставка нового елемента у задану позицію, видалення елемента із заданої позиції, перегрупування елементів за деякою ознакою. Для визначеності приймається нумерація елементів даних у файлі натуральними числами ***1, 2, 3, …, N***. Звичайно, що операції вставки та видалення елементів будуть коректними, якщо позиції таких елементів не перевищують початкової кількості елементів у файлі.

- **Вставка нового елемента у задану позицію**. У випадку вставки нового елемента в задану позицію ***K***, тобто ***K***-тим по порядку елементом, потрібно звільнити місце для його запису так, щоб не спотворити усі наступні за ним елементи. З цією метою файл збільшується на один елемент шляхом повторюваного перезапису елементів у наступну за ними позицію при зворотному русі від кінця файла до місця вставки (Рис. 1).

Попередньо потрібно визначити початкову кількість ***N*** елементів даних у файлі.

Таким чином потрібно виконувати наступну послідовність дій:

1) змістити файловий вказівник у позицію перед останнім елементом ***Pos1 = (N-1)*Size_el, filepos(f, Pos1, 0)*** ;
2) читається поточний елемент даних ;
3) прочитаний елемент даних перезаписується в наступну позиції ;
4) файловий вказівник повертається на три елементи даних назад ***Pos2 = -3*Size_el, filepos(f, Pos2, 1)*** ;
5) кроки 2), 3), 4) повторюються поки не буде перезаписаний елемент із задної позиції ***K*** ;
6) оскільки після останнього виконання кроку 4) зупинка зафіксована на елементі, що передує позиції вставки, то власне перед вставкою нового елемента потрібно виконати ще одне зміщення файлового вказівника на один елемент вперед ***filepos(f, Size_el, 1)*** ;
7) у звільненій позиції виконується запис нового елемента.

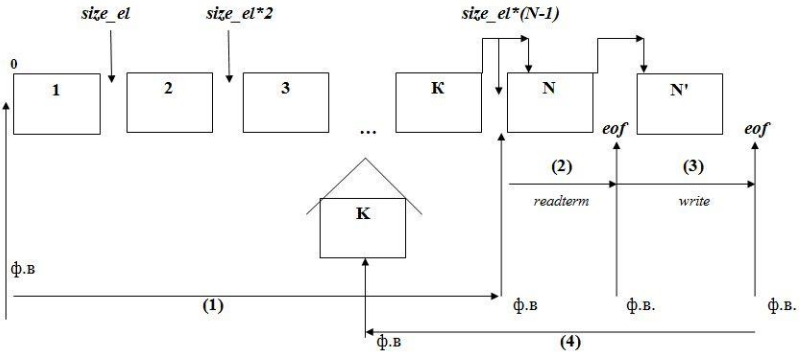

Рис. 1. Схема вставки нового елемента у файл прямого доступу

● **Видалення елемента із заданої позиції.** У випадку видалення елемента із заданої позиції ***K***, тобто ***K***-го по порядку, потрібно виконувати переміщення усіх наступних за ним елементів шляхом повторюваного їх перезапису у попередню позицію при прямому русі від місця видалення до кінця файла. В результаті таких дій на місці видаленого елемента буде перезаписано наступний за ним, а в кінці файла буде два однакових елементи. На завершення потрібно вилучити останній із них шляхом відсікання хвоста файлу, тобто встановленням маркера ***eof*** саме перед цим елементом (Рис. 2).

Таким чином потрібно виконувати наступну послідовність дій:

1) файловий вказівник зміщується на наступний елемент після елемента, що вилучається, ***Pos1=K*Size_el, filepos(f, Pos1, 0)*** ;

2) читається поточний елемент даних;
3) файловий вказівник повертається на два елементи назад ***Pos2 = -2*Size_el, filepos(f, Pos2, 1)*** ;
4) прочитаний елемент даних перезаписується в поточну позицію ;
5) файловий вказівник зміщується на один елемент даних уперед ***filepos(f, Size_el, 1)*** ;
6) кроки 2), 3), 4), 5) повторюються поки не буде перезаписаний останній елемент, тобто досягається кінець файлу ***eof*** ;
7) оскільки після останнього виконання кроку 5) зупинка зафіксована в кінці файла перед маркером ***eof***, то потрібно виконати зміщення файлового вказівника на один елемент назад ***filepos(f, Size_el, 2)*** , а потім маркер кінця файлу встановлюється в поточну позицію файлового вказівника ***setfilesize(f)***.

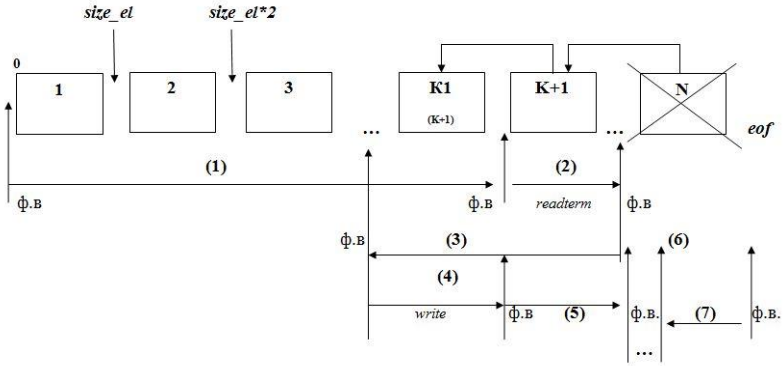

Рис. 2. Схема видалення елемента у файлі прямого доступу

- **Перегрупування елементів** можна розуміти як операцію сортування за деякою ознакою. Оскільки більшість алгоритмів сортування мають квадратичну складність по операціях перезапису, то їх реалізація у випадку файлів є достатньо затратною по часу. Для зменшення кількості операцій читання і запису окрім основного файлу даних доцільно використовувати додатковий індексний файл. Це особливо актуально у випадку довільних файлів послідовного доступу, коли окремі елементи даних мають різну довжину в байтах. У індексному файлі зберігаються позиції всіх елементів даних основного файла в тому порядку, який відповідає обраній умові впорядкованості. Такий підхід доцільно застосовувати і у випадку багатократних операцій вставки чи видалення елементів. Замість багаторазового виконання затратних по часу операцій перезапису елементів слід усі переміщення відображати лише в індексному файлі. А при видаленні елементів їх позиції в індесному файлі можна

замінювати відповідними від'ємними числами. Це дозволить навіть виконувати відновлення раніше видалених елементів.

Зауваження. Індексний файл може модифікуватися, тому доцільно всі числові дані відразу приводити до деякої достатньої спільної довжини з врахуванням можливого перезапису від'ємних чисел!

Зрозуміло, що остаточний перезапис елементів даних в основному файлі слід проводити в кінці сеансу роботи з файлом після завершення всіх модифікацій. При цьому створюється додатковий тимчасовий файл даних, в який послідовно перезаписується елементи із основного файлу за порядком їх позицій в індексному файлі. Раніше вилучені елементи, для яких позиції в індексному файли мають від'ємні значення, просто пропускаються. Після завершення перезапису старий варіант основного файлу можна зберегти як ***bak***-файл, а отриманий тимчасовий файл переіменувати в новий основний файл даних. Якщо ж потреби в ***bak***-файлі немає, то тимчасовий файл перетворюється на новий основний файл даних простим переіменуванням. Але тоді вже втрачається можливість відновлення раніше видалених даних.

У якості прикладу пропонується розглянути процес сортування по неспаданню деякого файла цілих чисел.

Початково формується індексний файл, в якому записано зміщення від початку основного файлу даних кожного елемента з врахуванням двох байтів-символів маркера кінця рядка. Це виконується в один прохід по основному файлу даних. У індексному файлі всі числові дані доповнені необхідною кількістю незначущих пропусків справа і приведені до спільної довжини для наступного перезапису від'ємних чисел.

початковий файл даних	*початковий індексний файл*
123↵	*0 _ _*
-12345↵	*5 _ _*
0↵	*13 _*
678↵	*16 _*
-7↵	*21 _*
5↵	*25 _*
-32768↵	*28 _*
0↵	*36 _*
321↵	*39 _*
eof	*eof*

Наступним етапом є власне саме сортування, яке проводиться за принципом вибору. При цьому вибирається перший за порядком

найменший елемент серед ще не використаних. У новому індексному файлі записується позиція входження обраного елемента в основному файлі даних. Далі замість вилучення вже опрацьованого елемента з основного файла даних у старому індексному файлі його позиція замінюється відповідним від'ємним числом. Така послідовність дій повторюється, поки в основному файлі даних не будуть "використані" всі елементи. Після завершення аналізу даних буде сформований новий індексний файл. У ньому теж доцільно всі елементи даних приводити до спільної довжини.

На останньому етапі формується результуючий файл даних. При цьому з нового індексного файла послідовно зчитуються позиції входження в старому файлі даних чергового елемента. Виконується позиціювання файлового вказівника, читання відповідного елемента та дозапис його у новий результуючий файл в кінець.

новий індексний файл	*новий файл даних*
28 _	*-32768↵*
5 _ _	*-12345↵*
21 _	*-7↵*
13 _	*0↵*
36 _	*0↵*
25 _	*5↵*
0 _ _	*123↵*
39 _	*321↵*
16 _	*678↵*
eof	*eof*

ЛІТЕРАТУРА

1. Дейнега Л.Ю., Камінська Ж.К., Левада І.В., Сердюк С.М. Практичне програмування мовою Visual Prolog: навчальний посібник. Запоріжжя: ЗНТУ, 2016. 236 с.
2. Любченко К.М. Мова програмування Prolog. Базовий курс: навчально-методичний посібник. Черкаси : ЧНУ імені Богдана Хмельницького, 2016. 136 с.
3. Різник О.Я. Логічне програмування: навчальний посібник. Львів : Видавництво Львівської політехніки, 2008. 332 с.
4. Шумейко О.О., Кнуренко В.М. Visual Prolog. Опануй на прикладах: навчальний посібник. Дніпропетровськ : Біла К.О., 2014. 404 с.
5. Bratko I. Prolog Programming for Artificial Intelligence. 4th ed. Pearson, 2011. 688 p.
6. Clocksin W.F., Mellish C.S. Programming in Prolog. 6th ed. Springer, 2012. 386 p.

Навчальне видання

Сяський Володимир Андрійович

ЛОГІЧНЕ ПРОГРАМУВАННЯ

З ПРИКЛАДАМИ ЗАСТОСУВАННЯ

Навчальний посібник

Підписано до друку 26.01.2017 р.
Формат 60×84 1/16. Папір офсетний.
Гарнітура Times New Roman Cyr.
Умовн. друк. арк. 5,58.
Тираж 50 прим.
Редакційно-видавничий відділ
Рівненського державного гуманітарного університету.
33028, м. Рівне, вул. Ст. Бандери, 12.