

22.183.492

Т65

Трушевський В. М.

**МОВИ ПРОГРАМУВАННЯ
ДЛЯ ШТУЧНОГО ІНТЕЛЕКТУ
ПРОГРАМУВАННЯ МОВОЮ
PROLOG**

Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

В. М. Трушевський

**МОВИ ПРОГРАМУВАННЯ
ДЛЯ ШТУЧНОГО ІНТЕЛЕКТУ
ПРОГРАМУВАННЯ МОВОЮ PROLOG**

Навчальний посібник

*Рекомендовано
Міністерством освіти і науки України*

ІБ ПНУС



754483

Львів
Видавничий центр ЛНУ імені Івана Франка
2009

12.128.492

УДК 004.434:004.8(075.8)

ББК 3973.2-18.2я73-2

T-65

Рецензенти:

д-р фіз.-мат. наук, проф. *В. А. Кривень*

(Тернопільський державний технічний університет імені Івана Пулюя)

д-р техн. наук, проф. *В. В. Пасічник*

(Національний університет "Львівська політехніка")

д-р фіз.-мат. наук, ст. наук. співр. *В. В. Пічкур*

(Київський національний університет імені Тараса Шевченка)

Рекомендовано

Міністерством освіти і науки України

як навчальний посібник для студентів вищих навчальних закладів

Прикарпатський національний університет імені Василя Стефаника

код 02125266

НАУКОВА БІБЛІОТЕКА

Inv. №

754483

Трушевський В. М.

T-65 Мови програмування для штучного інтелекту : програмування мовою Prolog : навч. посібник / В. М. Трушевський. – Львів : Видавничий центр ЛНУ ім. Івана Франка, 2009. – 150 с.

ISBN

Посібник складається з чотирьох частин. У першій частині подано вимоги до мов програмування задач штучного інтелекту. Другу частину присвячено теорії предикатів першого порядку, на якій основано мову Prolog. У третій частині розглянуто основи мови логічного програмування Visual Prolog. Четверта частина містить приклади розв'язування задач на мові Prolog.

Для студентів і аспірантів факультету прикладної математики та інформатики.

УДК 004.434:004.8(075.8)

ББК 3973.2-18.2я73-2

© Трушевський В. М., 2009

© Львівський національний університет імені Івана Франка, 2009

ISBN

ВСТУП

Вимоги до мов програмування інтелектуальних систем.

Основне завдання програмування штучного інтелекту полягає у формуванні подання знань та послуговувальних структур, необхідних для розв'язування інтелектуальної задачі. Вимоги до цих структур визначають необхідні властивості мови реалізації.

У літературі виділяють два рівні опису інтелектуальних систем: знань (knowledge level) та символів (symbol level). Рівень символів пов'язаний з конкретним формалізмом, який застосовують для подання знань у процесі розв'язування задач. Прикладом розгляду задачі на такому рівні є опис логіки предикатів, як мови подання. Рівень знань пов'язаний зі змістом інформації та способами її подання.

Такий поділ відображається на архітектурі систем та стилі їхнього розроблення. Оскільки користувачі використовують програму у межах своєї обізнаності та можливостей, програми реалізації штучного інтелекту мусять мати чітко виділений рівень знань. На рівні символів визначають мову подання. Нижче рівня символів знаходиться рівень організування програми та вирішення питань щодо її розроблення.

Багаторівневий підхід до розроблення інтелектуальних систем дає змогу програмісту не зважати на складності нижчих рівнів та сконцентрувати свої зусилля на питаннях певного рівня абстракції. У разі такого підходу з'являється змога виділити теоретичні основи штучного інтелекту та абстрагуватися від деталей конкретної реалізації або мови програмування. Ієрархію рівнів системи зображено на рис. 1.

Рівень знань окреслює можливості інтелектуальної системи. На цьому рівні вирішують такі питання: Які запити допустимі у

системі? Які об'єкти і відношення відіграють важливу роль у цій предметній області? Як додати у систему нові знання? Чи будуть факти змінюватися з часом? Як у системі будуть зреалізовані міркування про знання? Чи володіє ця предметна область добре зрозумілою семантикою? Наявна в ній незрозуміла чи неповна інформація? Важливим кроком у розробленні архітектури програми є детальний аналіз питань якогось рівня та вибір конкретного способу подання, який використовують на символічному рівні.

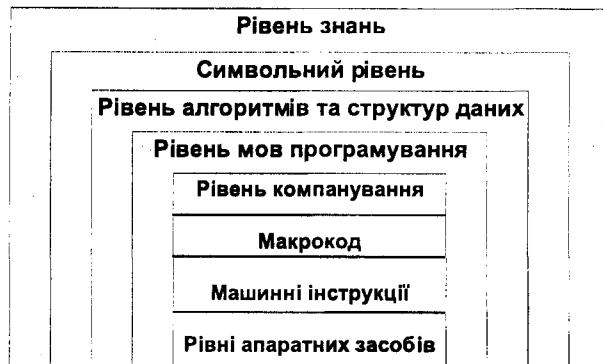


Рис. 1. Рівні системи

На символічному рівні важливим питанням є вибір мови програмування. Мова подання має не тільки давати змогу виражати необхідні знання, але і бути узгодженою, модифікованою, обчислювально-ефективною.

Подібно до розділення рівня знань та символів, у програмі можна розмежувати символічний рівень і алгоритми та структури даних, які використовують для реалізації символічного рівня. Багато алгоритмів та структур даних, що їх використовують у задачах оброблення природної мови, зведені до роботи з деревами та таблицями. Але є і специфічні для штучного інтелекту подання.

Нижче рівня алгоритмів і структур даних розміщений рівень мови. На цьому рівні важливу роль відіграє стиль програмування.

Хоча добрий стиль програмування дає можливість припустити розділення конкретних властивостей мови програмування та вищих рівнів, специфіка задач штучного інтелекту потребує їхньої глибокої взаємодії. Крім того, структура мови має задовольняти обмеженням, зумовленим ще більш нижчими рівнями комп'ютерної архітектури, охоплюючи операційну систему, архітектуру апаратних засобів, об'єм пам'яті та швидкодію процесора. У мовах LISP та PROLOG реалізовано вимоги символічного рівня та враховано архітектуру більш нижчих рівнів.

Вибір мови програмування для задач штучного інтелекту. Штучний інтелект сформувався як окрема галузь знань та засвідчив доцільність свого застосування для багатьох задач на основі мов LISP та PROLOG. Однак останнім часом вага цих мов під час розв'язання задач штучного інтелекту дещо знизилася. Це пояснюють вимогами до розроблення програмних систем. Системи штучного інтелекту слугують модулями до інших великих застосувань, тому стандарти розроблення змушують використовувати єдину мову програмування. Сучасні системи штучного інтелекту реалізують на багатьох мовах, включаючи Smalltalk, C, C++ та Java. Тим не менше, LISP та PROLOG продовжують відігравати свою роль у розробленні прототипів програм та нових методів розв'язування задач.

Крім того, ці мови слугують для обґрунтування багатьох засобів, які включають у сучасні мови програмування. Найяскравішим прикладом є мова JAVA, у якій використовують динамічне зв'язування, автоматичне керування пам'яттю та інші засоби, вперше реалізовані у мовах програмування задач штучного інтелекту.

1. ЧИСЛЕННЯ ПРЕДИКАТИВ

У цьому розділі розглянемо числення предикатів як мову подання штучного інтелекту. Зазвичай під **численням** розуміють складову деяких розділів математики, які визначають правила обчислень та оперування з об'єктами того чи іншого типу (диференційне, інтегральне, варіаційне числення), або дедуктивну систему. **Дедуктивною системою** називають спосіб задання множини шляхом вказання вихідних елементів (аксіом числення) та правил виведення, кожне з яких описує побудову нових елементів з вихідних. Численнями називають найважливіші дедуктивні системи – числення висловлювань та числення предикатів.

Розділ починається з короткого огляду числення висловлювань. Далі описано синтаксис та семантику числення предикатів. Обговорено правила виведення числення предикатів та їхнє застосування до розв'язування задач.

1.1. Числення висловлювань

Числення предикатів та числення висловлювань – це передусім мови. Використовуючи слова, фрази та речення, можна передавати властивості та відношення в оточуючому світі та міркувати про них. Для вивчення мови необхідно ознайомитися з її складовими.

1.1.1. Символи та речення

Означення 1.1. Символи числення висловлювань.

Символи числення висловлювань – це символи P, Q, R, S, \dots , значення істинності *true* (істина), *false* (хибно) та логічні зв'язки \wedge (диз'юнкція), \vee (кон'юнкція), \neg (заперечення), \rightarrow (імплікація), \equiv (еквівалентність).

Символи висловлювань (пропозиційні символи) утворюють висловлювання (proposition) або твердження відносно деякого світу. Вони можуть бути як істинні, так і хибні, наприклад, “автомобіль червоний” або “вода мокра”. Висловлювання позначають великими буквами. У численні висловлювань речення формують з елементарних символів відповідно до правил.

Означення 1.2. Речення числення висловлювань.

Кожний логічний символ та символ істинності є реченням.

Наприклад: *true*, P , Q та R – речення. *Заперечення* речення є реченням. Наприклад: $\neg P$ та $\neg false$. *Кон'юнкція, диз'юнкція, імплікація, еквівалентність* двох речень є реченням. Відповідні приклади речень: $P \wedge \neg P$, $P \vee \neg P$, $P \rightarrow Q$, $P \vee Q \equiv R$.

Легітимні речення називають *правильно побудованими формулами* (well-formed formulas – WFF), або ВПФ.

У виразах $P \wedge Q$ елементи P та Q називають *кон'юнктами*. У виразах $P \vee Q$ – *диз'юнктами*. У імплікації $P \rightarrow Q$, P – *припущення* імплікації, Q – *висновок* імплікації.

У реченнях числення висловлювань знаки $()$ та $[]$ використовують для групування символів у підвирази і, отже, дають змогу керувати порядком їхнього оцінення та присвоєння значень.

Вираз – це речення або правильно побудована формула (ВПФ) числення висловлювань тоді й лише тоді, коли він може бути сформульований як деяка послідовність допустимих символів згідно з встановленими правилами. Наприклад: $((P \wedge Q) \rightarrow R) \equiv \neg P \vee \neg Q \vee R$.

1.1.2. Семантика числення висловлювань

Оскільки програми штучного інтелекту мають бути узгоджені зі структурами, які їх подають, важливо довести, що правдивість результатів програм залежить тільки від правдивості початкового значення, тобто, що процедури виводу не містять логічних помилок. Для цього необхідна точна інтерпретація семантики.

Речення, що дає деякий опис світу, може бути як істинним, так і хибним. Присвоєння значення істинності логічним реченням називають *інтерпретацією*. Отже, *інтерпретація* (interpretation) –

це твердження відносно правдивості речення у деякому можливо-му світі.

Формально *інтерпретація* – це відображення логічних символів на множину $\{T, F\}$ (*true* – істина, *false* – хибно).

Кожне можливе відображення значення істинності висловлювання відповідає можливій інтерпретації світу. Наприклад, якщо P позначає висловлювання "йде дощ", а Q – "я на роботі", то набір висловлювань $\{P, Q\}$ має чотири різних функціональних відображення в таблиці істинності $\{T, F\}$, які відповідають чотирьом різним інтерпретаціям.

Означення 1.3. Семантика числення висловлювань.

Інтерпретація набору висловлювань – це присвоєння значення істинності, T або F , кожному пропозиційному символу. Символу *true* завжди присвоюється T , а символу *false* – значення F . Інтерпретації, або значення істинності речень (ВПФ), визначені так:

Істинність *заперечення*: висловлювання $\neg P \in F$, якщо P набуває значення T ; та висловлювання $\neg P \in T$, якщо P набуває значення F .

Істинність *кон'юнкції*: висловлювання набуває значення T , тільки якщо обидва кон'юнкти набувають значення T , інакше вираз набуде значення F .

Істинність *диз'юнкції*: висловлювання набуває значення F , тільки якщо обидва диз'юнкти набувають значення F , інакше вираз набуде значення T .

Істинність *імплікації*: висловлювання набуває значення F тільки тоді, коли припущення $\in T$, а висновок – F ; інакше вираз набуде значення T .

Істинність *еквівалентності*: висловлювання набуває значення T тільки тоді, коли обидва вирази мають однакові значення істинності для всіх можливих інтерпретацій; інакше вираз еквівалентності набуде значення F .

Значення істинності складних виразів часто описують *таблицями істинності*. Таблиця істинності містить усі можливі варіанти значень істинності для елементарних суджень (атомарних формул), які утворюють великі вирази, та задає значення істинності виразам для кожної можливої інтерпретації.

Демонструючи ідентичність таблиць істинності для двох різних речень у численні висловлювань, можна довести еквівалентність таких виразів. Для логічних виразів P , Q та R

$$\neg(\neg P) \equiv P$$

$$(P \vee Q) \equiv (\neg P \rightarrow Q)$$

Закон контрпозиції імплікації: $P \rightarrow Q \equiv (\neg Q \rightarrow \neg P)$.

Закон Моргана: $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$ та $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.

Закон комутативності: $(P \vee Q) \equiv (Q \vee P)$ та $(P \wedge Q) \equiv (Q \wedge P)$.

Асоціативний закон: $((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$.

Асоціативний закон: $((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$.

Дистрибутивний закон: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$.

Дистрибутивний закон: $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$.

Ці тотожності можуть бути використані для зведення виразу числення висловлювань до синтаксично різних, але логічно еквівалентних форм. У інших частинах посібника розглянуто правила виведення *modus ponens* та резолюції (resolution) для використання яких необхідно, щоб вирази були задані у визначеній формі. Правила виведення дають змогу замінювати логічний вираз іншими формами з еквівалентними значеннями істинності, що дуже важливо.

1.2. Основи числення предикатів

У численні висловлювань кожний елементарний символ (P , Q , ...) означає висловлювання деякої складності. Водночас немає способу отримання доступу до компонент окремого судження. Числення предикатів надає таку можливість. Наприклад, замість того, щоб увести єдиний символ висловлювання P , позначивши все речення "у вівторок йшов дощ", можна створити предикат *weather* (*погода*), який описує відношення між датою та погодою: *weather (tuesday, rain)*, або *погода (вівторок, дощ)*. За допомогою правил зведення можна змінити вирази числення предикатів, безпосередньо звертаючись до їхніх компонент та виводячи нові речення.

Крім того, числення предикатів дає змогу вводити у вирази змінні, за допомогою яких можна створювати узагальнюючі твердження відносно класів логічних об'єктів. Наприклад, можна задати, що для всіх значень X , де X – день тижня, твердження $weather(X, rain)$, або *погода* (X , дощ) є істинним, тобто кожен день йде дощ. Визначимо спочатку синтаксис мови предикатів, а далі обговоримо її семантику.

1.2.1. Синтаксис предикатів та речень

Перед тим, як визначити синтаксис мови, розглянемо алфавіт та граматику для створення символів мови.

Означення 1.4. Символи числення предикатів.

Алфавіт числення предикатів складається з таких елементів.

1. Набір букв англійського алфавіту як верхнього, так і нижнього регістру.
2. Набір цифр – 0, 1, ..., 9.
3. Символ підкреслення $_$.

Символи в численні предикатів починаються з букви, за якою слідує послідовність перелічених вище знаків.

Наведемо приклад допустимих знаків у алфавіті числення предикатів: $aR69p_z$. Наступний приклад недопустимих знаків: $\# \% @ / \& " ' "$.

Символи (або ідентифікатори) використовують для позначення об'єктів, властивостей та відношень у галузі міркування. Круглі дужки, коми та крапки використовують винятково для правильної побудови виразів, але ними не позначають об'єкти або відношення з визначеної області. Їх називають *невластивими символами* (improper symbol).

Символами числення предикатів можна описувати змінні, константи, функції або предикати. Константами називають визначені об'єкти або властивості з галузі міркування. Символьні константи мають починатися з букви нижнього регістру. Константи *true* та *false* зарезервовані як *символи істинності*.

Ідентифікатори змінних використовують для позначення загальних класів об'єктів або властивостей. Змінні, за допомогою яких описують ідентифікатори, починаються з великої літери.

Числення предикатів також допускає наявність функцій. Ідентифікатори функцій (подібно константам) починаються з символу нижнього регістру. Функції позначають відображення одного або декількох елементів множини (*область визначення функції*) у однозначно визначений елемент іншої множини (*область значень функції*). Крім таких звичайних арифметичних функцій, як додавання та множення, функції можуть визначати відображення між нечисловими областями.

Кожний функціональний символ пов'язаний з арністю (arity), яка засвідчує кількість параметрів цієї функціональної залежності. Наприклад, символ *father* (батько) міг би означати функцію арності 1, яка дає змогу визначити батька, а *plus* (додавання) міг би бути функцією арності 2, яка відображає два числа як їхню арифметичну суму.

Функціональний вираз – це ідентифікатор функції, за яким слідує його аргументи (або параметри). Аргументи – це елементи області визначення функції. Наприклад, $f(X, Y)$, *plus* (7, 3), *price* (bananas), *father* (ivan).

Кожний функціональний вираз означає відображення аргументів як єдиний об'єкт множини значень, який називають *значенням* функції. Наприклад, якщо *father* – це унарна функція, то *father* (ivan) є функціональним виразом, значенням якого (за бажанням автора) стає *petro*. Заміну функції її значенням називають *оцінюванням* (evaluation).

Поняття символу числення предикатів, або *терма*, формалізовано таким визначенням.

Означення 1.5. Символи та терми.

До символів числення предикатів відносять:

1. Символи істинності *true* та *false*.
2. Символи констант – це символьні вирази, які починаються з символу нижнього регістру.

3. *Символи змінних* – це символічні вирази, що починаються з символу верхнього регістру.

4. *Функціональні символи* – це символічні вирази, які починаються з малої літери.

Функціональний вираз (function expression) складається з функціональної константи арності n , за якої слідує n термів t_1, t_2, \dots, t_n , які взято у круглі дужки та розділено комами.

Терм числення предикатів може бути константою, змінною або функціональним виразом.

Отже, *термом* числення предикатів позначають об'єкти та властивості з області визначення задачі. Приклади термів: *X*, *blue*, *times* (3,2), *mother* (ira), *cat*.

Символи в численні предикатів можуть також бути предикатами. Предикатні символи, подібно константам та іменам функцій, починаються з букви нижнього регістру. Предикат вказує на відношення між декількома об'єктами (можливий варіант з нульовим числом об'єктів). Кількість об'єктів, зв'язаних таким чином, визначає арність предиката. Приклади предикатів: *like*, *equals*, *on*, *near*, *part_of*.

Атомарне висловлювання (атомарна формула) (atomic sentence) – це предикат арності n , за яким слідує n термів, взятих у круглі дужки та розділених комами. Приклади атомарних висловлювань:

likes (george, natalja)

likes (george, kate, tuesday)

friends (nick, george)

likes (X, george)

likes (X, X)

friends (father_of (david), father_of (andriy))

Символами предикатів у цих виразах стають *likes*, *friends*. Символ предиката може бути використаний з різною кількістю аргументів. Наприклад, *likes*. Вважають, що ним можна описати два різні відношення. Аргументи предиката виступають термами та можуть також містити змінні або функціональні вирази.

Означення 1.6. Предикати та атомарні речення.

Символи предиката – це символи, які починаються з малої літери. Предикати пов'язані з додатним цілим числом, яке визначає

арність предиката. Предикати з однаковим іменем, але різною арністю вважають різними.

Атомарне речення – це предикатна константа арності n , за якою слідує n термів, t_1, t_2, \dots, t_n , взятих у круглі дужки та розділених комами. Значення істинності *true* та *false* також є атомарними реченнями.

Атомарні речення також називають *атомарними виразами*, *атомами* або *реченнями*.

Можна комбінувати атомарні речення та формувати речення в численні предикатів, використовуючи логічні оператори \wedge , \vee , \neg , \rightarrow , \equiv .

Числення предикатів першого порядку охоплює два символи: *квантори змінних* (variable quantifier) \forall та \exists . Вони обмежують значення речення, яке містить змінну.

Квантор всезагальності \forall означає, що речення істинне для всіх значень змінної. У прикладі $\forall X \text{ likes}(X, \text{ice_cream})$ вираз істинний для всіх значень X в області визначення X . *Квантор наявності* \exists вказує, що речення істинне хоча б для одного значення з області визначення. Вираз $\exists Y \text{ friends}(Y, \text{petro})$ істинний, якщо є принаймні один об'єкт Y , який є другом Петра.

Означення 1.7. Речення числення предикатів.

Кожне атомарне речення є реченням.

1. Якщо s – речення, тоді його заперечення також є реченням.
2. Якщо s_1 та s_2 – речення, тоді відповідно їхня кон'юнкція, диз'юнкція, імплікація та еквівалентність також є реченнями.
3. Якщо X – змінна та s – речення, тоді $\forall X s$ є реченням та $\exists X s$ – речення.

Розглянемо приклади вірно побудованих речень. Нехай *mult* (помножити) та *plus* (додати) – символи функцій арності 2 і нехай *equal* (дорівнює) та *foo* є предикатними символами арності 2 і 3 відповідно.

plus (two, three) (додати два та три) – це функція, тобто не атомарне речення.

equal (plus (two, three), five) – атомарне речення.

$\exists X \text{ foo } (X, \text{two}, \text{plus}(\text{two}, \text{three})) \wedge \text{equal}(\text{plus}(\text{two}, \text{three}), \text{five})$ – речення, тому що обидва кон'юнкти є реченнями.

$(\text{foo}(\text{two}, \text{two}, \text{plus}(\text{two}, \text{three}))) \rightarrow \text{equal}(\text{plus}(\text{two}, \text{three}), \text{five}) = \text{true}$ – речення, тому що всі компоненти речення пов'язані логічними операторами.

Використовуючи псевдомову, опишемо рекурсивну функцію *verify_sentence*, яка отримує аргумент вираз та повертає значення *true*, якщо вираз є реченням.

```
function verify_sentence (expression);
begin
  case
    вираз є атомарним реченням: return true;
    вираз має вигляд  $Q \ X \ s$ , де  $Q$  – або  $\forall$  або  $\exists$ ,  $X$  – змінна,
      if verify_sentence(s) then return true
      else return false;
    вираз має вигляд  $\neg s$ :
      if verify_sentence(s) then return true
      else return false;
    вираз має вигляд  $s_1 \text{ op } s_2$ , де op – бінарний логічний
      оператор:
      if verify_sentence( $s_1$ ) and verify_sentence( $s_2$ ) then return true
      else return false;
    в інших випадках: return false
  end
end.
```

Завершуючи цей підрозділ, наведемо приклад використання числення предикатів. Розглянемо область визначення – набір сімейних відносин.

```
mother(galyna,sergiy)
mother(galyna,petro)
father(mykola,sergiy)
father(mykola,petro)
 $\forall X \forall Y \text{ father}(X, Y) \vee \text{mother}(X, Y) \rightarrow \text{parent}(X, Y)$ 
 $\forall X \forall Y \forall Z \text{ parent}(X, Y) \vee \text{parent}(X, Z) \rightarrow \text{sibling}(Y, Z).$ 
```

У цьому прикладі для визначення набору відношень батьків та дітей використано предикати *mother* (мати) та *father* (батько). У термінах цих предикатів імплікація створює картину загальних

уявлень двох інших відношень – батьківських (*parent*) та братських (*sibling*). Для реалізації процесу на комп'ютері необхідно його формалізувати, тобто визначити алгоритми виведення, які даватимуть змогу робити правильні висновки з набору тверджень числення предикатів. Визначимо спочатку семантику числення предикатів, а потім перейдемо до правил виведення.

1.2.2. Семантика числення предикатів

Семантика числення предикатів забезпечує формальну основу для визначення значень істинності коректно поставлених виразів. Істинність виразів залежить від відповідності констант, змінних, предикатів та функцій об'єктам і відношенням з області визначення. З істинності відношень в області визначення впливає істинність відповідних виразів.

Визначимо семантику числення предикатів формально. Спочатку визначимо *інтерпретацію* в області визначення D . Далі, використовуючи її, визначимо присвоєння значень істинності реченням мови.

Означення 1.8. Інтерпретація.

Нехай область визначення D – деяка непорожня множина.

Інтерпретація на D – це зв'язування логічних об'єктів D з кожною константою, змінною, предикатом та функціональним символом у виразі числення предикатів на підставі таких правил.

1. Кожній константі відповідає елемент з D .
2. Кожній змінній відповідає непорожня підмножина з D , яка є областю допустимих значень для цієї змінної.
3. Кожну функцію f n -арності (числа операндів) m визначають для m параметрів з D та задають відображення з D^m в D .
4. Кожен предикат p n -арності n визначають для n параметрів з D та задають відображення з D^n в $\{T, F\}$.

У разі такого визначення інтерпретації, щоб отримати значення виразу, потрібно присвоїти виразу значення істинності на цій інтерпретації.

Означення 1.9. Значення істинності виразів числення предикатів.

Нехай відомий вираз E та інтерпретація I для E на непорожній області визначення D . Значення істинності для E визначають таким чином.

1. Значення константи – це елемент з D , якому відповідає константа в інтерпретації I .
2. Значення змінної – це множина елементів з D , які відповідають цій змінній в інтерпретації I .
3. Значення функціонального виразу – це такий елемент з D , який отримують унаслідок оцінювання функції для значень параметрів, що відповідають інтерпретації.
4. Значення символу істинності *true* – це T , а *false* – F .
5. Значення атомарного речення рівно або T , або F , та визначають за інтерпретацією I .
6. Значення заперечення речення рівно T , якщо значення речення рівно F ; значення заперечення речення рівно F , якщо значення речення рівно T .
7. Значення істинності виразів, які використовують операції $\wedge, \vee, \rightarrow, \equiv$, визначають за значеннями їхніх операндів.
Для змінної X та речення S , що містить X , виконують такі відношення.
8. Значення виразу $\forall X S$ рівно T , якщо S рівно T для всіх значень X з I , інакше воно рівне F .
9. Значення виразу $\exists X S$ рівно T , якщо в інтерпретації є значення X , для якого S рівне T , інакше воно рівне F .

Квантифікація змінних – це важлива частина семантики числення предикатів. Змінна, яку використовують у реченні, виконує роль заповнювача та позначає *знакомісце*. На це місце може бути підставлена довільна константа, що допускається інтерпретацією. Тобто змінна – це шаблон для підстановки. У численні предикатів змінні мають бути зв'язані одним з двох кванторів: універсальності або наявності. Змінну вважають *вільною*, якщо вона не пов'язана квантором універсальності або наявності. Вираз є *замкнутим* (closed), якщо всі його змінні зв'язані кванторами. *Основний вираз* (ground expression) взагалі не має ніяких змінних.

Для позначення *області дії* квантора використовують круглі дужки. Приклад $\forall X(p(X) \vee q(Y) \rightarrow r(X))$ свідчить, що змінна X зв'язана квантором всезагальності з $p(X)$ та $r(X)$.

Наведемо декілька прикладів взаємозв'язку між операцією заперечення і кванторами всезагальності та наявності. Для предикатів p і q та змінних X і Y виконують такі співвідношення.

$$\begin{aligned} \neg \exists X p(X) &\equiv \forall X \neg p(X) \\ \neg \forall X p(X) &\equiv \exists X \neg p(X) \\ \exists X p(X) &\equiv \exists Y p(Y) \\ \forall X q(X) &\equiv \forall Y q(Y) \\ \forall X(p(X) \wedge q(X)) &\equiv \forall X p(X) \wedge \forall Y q(Y) \\ \exists X(p(X) \vee q(X)) &\equiv \exists X p(X) \vee \exists Y q(Y) \end{aligned}$$

Означення 1.10. Числення предикатів першого порядку (first-order predicate calculus) дає змогу зв'язувати знаком квантора змінні, які відповідають об'єктам з предметної області, але не предикати або функції.

Наприклад, $\forall (Likes) Likes(mykola, olena)$ не є правильно побудованим виразом у численні предикатів першого порядку. Відомі числення предикатів вищих порядків, в яких такі вирази піддаються інтерпретації.

Більшість граматично правильних англійських речень можуть бути наведені в численні предикатів першого порядку за допомогою символів, зв'язок та символічних змінних, визначених у цьому підрозділі. Важливо наголосити, що немає унікального відображення речень у вирази числення предикатів. Основна задача для програмістів систем штучного інтелекту полягає у тому, щоб знайти схему використання предикатів, оптимальну з погляду вразливості та ефективності кінцевого подання. Приклади англійських та українських речень, які описані засобами числення предикатів:

If it doesn't rain on Monday, Ivan will go to the mountains. (Якщо в понеділок не буде йти дощ, Іван піде у гори).

$\neg weather(rain, monday) \rightarrow go(ivan, mountains)$

All basketball players are tall. (Всі баскетболісти – високі).

$\forall X(basketball_player(X) \rightarrow tall(X))$

Some people like anchovies. (Деякі люди люблять анчоуси).

$\exists X(person(X) \wedge likes(X, anchovies))$

НАУКОВА БІБЛІОТЕКА

754483

1.2.3. Значення семантики на прикладі "світ блоків"

У цій частині роботи розглянемо питання про присвоєння значень істинності деяким виразам числення предикатів на прикладі "світ блоків". Промодельємо світ блоків, зображений на рис. 2.1. та сконструюємо алгоритм керування для руки робота. Для якісного наведення відношень у цьому світі можна використовувати речення числення предикатів. За допомогою цих речень потрібно описати, чи вільна верхня грань блока, чи можна взяти блок a і т. д. Припустимо, що комп'ютер володіє знаннями про розміщення кожного блока та руки, а також здатен відслідковувати переміщення блоків на столі (використовуючи тривимірну систему координат).

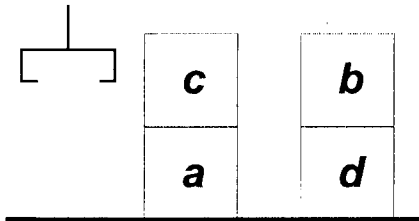


Рис. 1.1. Світ блоків:

$on(c,a)$ – блок c знаходиться на блоці a
 $on(b,d)$
 $ontable(a)$ – блок a лежить на столі
 $ontable(d)$
 $clear(b)$ – на b немає інших блоків
 $clear(c)$
 $hand_empty$ – рука порожня

Щоб можна було підняти один блок та поставити його на інший, обидва вони мають бути вільними (відкритими для руки). Створимо правило, що описує дію "блок вільний"

$$\forall X(\neg \exists Y on(Y,X) \rightarrow clear(X)).$$

Якщо змінна X набуде значення d , $Y = b$, тоді речення виявиться хибним. Щоб зробити це визначення істинним, потрібно блок b зняти з блоку d .

Крім вище визначеного правила, можна додати інші правила, за якими описують операції перекладання одного блока на інший. Наприклад, щоб покласти X на Y , спочатку потрібно звільнити руку, потім звільнити X , Y , взяти X ($pick_up(X)$) та поставити (put_down) його на Y

$$\forall X \forall Y ((hand_empty \wedge clear(X) \wedge clear(Y) \wedge pick_up(X) \wedge put_down(X,Y)) \rightarrow stack(X,Y)).$$

На рис. 1.1 наведено семантично інтерпретацію виразів числення предикатів, яка відображає константи та предикати у вирази з області визначення D (тут – це блоки та відношення між ними). Інтерпретація зіставляє значення істинності T з кожним виразом в описі. Для інших наборів блоків могла б бути інша інтерпретація. Питання полягає не в її унікальності. Інтерпретація, головне, має забезпечувати значення істинності для всіх виразів у наборі, а самі вирази створюють для достатньо детального опису світу. У наступному розділі ці ідеї будуть використані для формальної основи правил числення предикатів.

1.3. Правила виведення в теорії числення предикатів

1.3.1. Правила виведення

Семантика числення предикатів забезпечує основу для формалізації теорії **логічного виведення** (logical inference). Можливість логічно виводити нові правильні вирази з набору істинних тверджень – це важлива властивість числення предикатів. Логічно виведені вирази коректні, тому що вони **сумісні** зі всіма попередніми інтерпретаціями початкового набору виразів.

Вважають, що інтерпретація, яка робить речення істинним, **задовольняє** цьому реченню. Якщо інтерпретація задовольняє кожному елементу набору виразів, вважають, що вона задовольняє набору. Вираз X **логічно слідує** з набору виразів S числення предикатів, якщо кожна інтерпретація, яка задовольняє S , задовольняє і X . Це означає, наприклад, що довільний новий вираз числення

предикатів, доданий до світу блоків, який зображено на рис. 1.1., має бути істинним у цьому світі. Він має бути істинним і при довільній іншій інтерпретації, яку міг би мати цей набір виразів. Системи предикатів можуть мати безмежне число можливих інтерпретацій, тому практична необхідність перевіряти всі інтерпретації виникає рідко. В обчислювальному відношенні **правила виведення** (inference rule) дають можливість визначати, коли вираз як компонента інтерпретації логічно слідує з цієї інтерпретації.

Правило виведення забезпечує створення нових речень числення предикатів на підставі цих речень. Отже, правила виведення створюють нові речення, засновані на синтаксичній формі логічних тверджень. Якщо кожне речення X , отримано за допомогою деякого правила виведення на множині логічних виразів S , вважають, що це правило **обґрунтовано** (sound).

Якщо система правил виведення може створити будь-який вираз, який логічно слідуватиме з S , вважають, що ця система правил є **повною**. Наведене нижче правило *modus ponens* є прикладом обґрунтованого правила.

Означення 1.11. Задовольняти, модель, адекватність.

Для виразу X числення предикатів в інтерпретації I доречне таке визначення.

Якщо X приймає значення T на I при конкретних значеннях змінних, вважають, що I **задовольняє** X .

Якщо I задовольняє X при всіх значеннях змінних, тоді I є **моделлю** X .

X **виконанно** (satisfiable) тоді й тільки тоді, коли відома така інтерпретація і значення змінної, які йому задовольняють, інакше X **невиконано** (unsatisfiable).

Набір виразів **виконаний** тоді й тільки тоді, коли відомі інтерпретація та значення змінних, які задовольняють кожному елементу.

Якщо набір виразів **невиконаний**, то вважають, що він **суперечливий** (inconsistent).

Якщо X має значення T для всіх можливих інтерпретацій, то вважають, що X має силу, або **адекватний** (valid).

У прикладі (рис. 1.1) світ блоків є моделлю для логічного опису. В цій інтерпретації всі речення у прикладі були істинними. Якщо база знань реалізована як набір істинних тверджень для предметної галузі задачі, тоді предметна галузь слугує моделлю для бази знань.

Вираз $\exists X (p(X) \wedge \neg p(X))$ суперечливий, тому що він не може бути істинним ні при якій інтерпретації або значеннях змінних. З іншого боку, вираз $\forall X (p(X) \vee \neg p(X))$ адекватний.

Для перевірки адекватності виразів, які не містять змінних, можна використовувати метод таблиць істинності. Оскільки не завжди можна визначити адекватність виразу зі змінними, відомі методи доведення, які дають змогу генерувати довільний вираз, що логічно слідує з цього набору виразів. Їх називають **процедурами повного доведення** (complete proof procedures).

Означення 1.12. Процедура доведення.

Процедура доведення (proof procedure) – це комбінація правил виведення та алгоритму застосування цих правил до набору логічних виразів для створення нових речень.

Використовуючи наведені означення, можна формально визначити термін "логічно слідує".

Означення 1.13. Логічно слідує, обґрунтований та повний.

Вираз числення предикатів X **логічно слідує** з набору S виразів числення предикатів, якщо кожна інтерпретація та значення змінних, які задовольняють S , задовольняють і X .

Правило виведення **обґрунтовано** (sound), якщо кожний вираз числення предикатів, отриманий згідно з правилом із множини S виразів числення предикатів, також логічно слідує з S .

Правило виведення **повне** (complete), якщо на множині S виразів числення предикатів правило дає можливість вивести довільний вираз, який логічно слідує з S .

Правило *modus ponens* (правило відокремлення, або модус поненс) – це обґрунтоване правило виведення. Якщо вираз вигляду $P \rightarrow Q$ та інший вираз вигляду P істинні на інтерпретації I , тоді *modus ponens* дає змогу зробити висновок, що Q також істинний за

цієї інтерпретації. Дійсно, оскільки *modus ponens* є обґрунтованим правилом, Q істинно для всіх інтерпретацій, для яких P та $P \rightarrow Q$ є істинними.

Означення 1.14. *Модус поненс, модус толленс, виключення "I", введення "I", універсальне інстанціювання.*

Якщо відомо, що речення P та $P \rightarrow Q$ істинні, тоді *модус поненс* дає можливість вивести Q .

Відповідно до правила виведення *модус толленс* (*modus tollens*), якщо відомо, що $P \rightarrow Q$ є істинним та Q хибним, можна вивести $\neg P$.

Вияток "I" – правило, за яким виводять істинність обох кон'юнктив на підставі істинності кон'юнктивного речення. Наприклад, якщо $P \wedge Q$ істинно, можна зробити висновок, що P та Q істинні.

Введення "I" дає змогу вивести істинність кон'юнкції з істинності її кон'юнктив. Наприклад, якщо P та Q істинні, тоді кон'юнкція $P \wedge Q$ істинна.

Універсальне інстанціювання зводять до такого: якщо довільну змінну під квантором всезагальності в істинному реченні замінити довільним відповідним термом з області визначення, тоді результуючий вираз – істинний. Отже, якщо a належить тій самій області визначення, що і X та $\forall X p(X)$, тоді можна вивести $p(a)$.

Як простий приклад застосування правила *модус поненс* у численні висловлювань розглянемо речення: "Якщо йде дощ – земля буде вологою". Нехай P означає – йде дощ, Q – земля волога, тоді перше речення можна записати у вигляді $P \rightarrow Q$. Оскільки зараз дійсно йде дощ (P істинно), отримаємо систему аксіом.

$$\begin{array}{c} P \rightarrow Q \\ P \end{array}$$

Застосувавши правило *модус поненс* до набору істинних виразів, можна додати й те, що земля є вологою Q .

Правило *модус поненс* також може бути застосовано до виразів, які містять змінні. Розглянемо, наприклад, речення "Всі люди смертні", та "Сократ – людина, тому Сократ – смертний". Вислів "Всі люди смертні" може бути наведений у численні предикатів як

$$\forall X (man(X) \rightarrow mortal(X)). (\forall X (людина(X) \rightarrow смертна(X)).$$

"Сократ – людина"

$man(socrates).$ ($людина(сократ)$).

Замінивши у виразі X на $socrates$, отримаємо таке твердження: $man(socrates) \rightarrow mortal(socrates)$.

Застосувавши правило *модус поненс*, приходимо до висновку $mortal(socrates)$. Він може бути доданий до набору виразів, які логічно слідують з початкових тверджень. У наступному розділі розглянемо *алгоритм уніфікації* для визначення правомірності заміни X на $socrates$ та можливості застосування правила *модус поненс*.

1.3.2. Уніфікація

Уніфікація – це алгоритм визначення необхідних підстановок з метою *приведення* двох виразів відповідно до числення предикатів. Приклад такого процесу наведено у попередньому підрозділі, де терм $socrates$ з виразу $man(socrates)$ був використаний як підставлення для X у виразі $\forall X (man(X) \rightarrow mortal(X))$. Це підставлення дало змогу застосувати правило *модус поненс* та отримати виведення $mortal(socrates)$.

Змінні, що стоять під знаком квантора всезагальності забезпечують повну свободу у виконанні підставлень. Змінні під квантором існування можна вилучити з речень у базі даних, замінивши їх константами, які забезпечують істинність речення. Наприклад, $\exists X parent(X, ivan)$ може бути замінено виразом $parent(petro, ivan)$ або $parent(olena, ivan)$, зважаючи на те, що $petro$ та $olena$ є батьками $ivan$ у цій інтерпретації.

Процес вилучення змінних, пов'язаних квантором існування, ускладнюється тим фактом, що значення цих підставлень може залежати від значення інших змінних у виразі. Наприклад, у висловлюванні $\forall X \exists Y mother(X, Y)$ значення змінної Y під квантором існування залежить від значення X . *Сколемізація* (*skolemization*) – це заміна кожної змінної, пов'язана квантором існування, функцією декількох або всіх наявних у реченні змінних, яка повертає відповідну константу. У вищенаведеному прикладі, оскільки значення Y залежить від X , Y , можна замінити сколемівською функ-

цією (skolem function) f від X . Це породжує предикат $\forall X \text{ mother}(X, f(X))$. Сколемізація – це і процес, який також дає змогу зв'язувати змінні під квантором всезагальності з константами.

Процес уніфікації ускладнюється тим, що змінна може бути замінена довільним термом, охоплюючи інші змінні та функціональні вирази довільної складності. Ці вирази можуть також містити змінні. Наприклад, $\text{father}(\text{ivan})$ можна використати як підставлення для X у виразі $\text{man}(X)$.

Наведемо декілька реалізацій виразу $\text{foo}(X, a, \text{goo}(Y))$.

Їх можна отримати шляхом таких підставлень:

1. $\text{foo}(\text{fred}, a, \text{goo}(Z))$
2. $\text{foo}(W, a, \text{goo}(\text{jack}))$
3. $\text{foo}(Z, a, \text{goo}(\text{moo}(Z)))$.

У цьому прикладі екземпляри підставлень, або уніфікації, котрі роблять початкові вирази ідентичними кожному з трьох, можна записати у вигляді

$\{\text{fred}/X, Z/Y\}$
 $\{W/X, \text{jack}/Y\}$
 $\{Z/X, \text{moo}(Z)/Y\}$.

Запис $X/Y, \dots$ означає, що X є підставленням для змінної Y у початковому виразі. Підставлення також називають зв'язуванням. Кажуть, що змінна зв'язана зі значенням, яке використовують як підставлення.

У разі створення алгоритму уніфікації, який дає змогу обчислювати підставлення, необхідні для відповідності двох виразів, виникають деякі проблеми.

Хоча константу можна систематично використовувати як підставлення для змінної, довільну константу розглядають як "базовий екземпляр", який не може бути замінений. Не можна також два різних "базових екземпляри" використовувати як підставлення для однієї і тієї ж змінної.

Змінна не може бути уніфікована з термом, який містить її. Тому змінна X не може бути замінена на $p(X)$, оскільки це породжує безмежний вираз: $p(p(p(p(\dots X)\dots))$. Тест для такої ситуації названо *перевіркою входження* (occurs checks).

Якщо змінна зв'язана, в усіх інших уніфікаціях та процедурах виведення це має бути враховано. Якщо змінна зв'язана з константою, її не можна зв'язувати з іншим термом у разі наступних уніфікацій. Якщо змінну X_1 використовували як підставлення для іншої змінної X_2 , а потім було замінено константою, тоді в X_2 також потрібно відобразити це зв'язування. Безліч замінів, використаних у послідовності виведень, відіграє важливу роль, тому що вона може містити відповідь на початкове питання. Наприклад, якщо $p(a, X)$ уніфікувати за умовою правила $p(Y, Z) \rightarrow q(Y, Z)$, підставляючи $\{a/Y, X/Z\}$, модус поненс дає можливість вивести $q(a, X)$ у разі того самого підставлення. Якщо зіставити цей результат з умовою правила $q(W, b) \rightarrow r(W, b)$, тоді виведемо $r(a, b)$, урахувавши множини підставлень $\{a/W, b/X\}$.

Інше важливе поняття – це *композиція* підставлень уніфікації. Якщо S та S' є двома множинами підставлень, тоді композицію S та S' (запишемо SS') отримують після застосування S' до елементів S та додавання результату до S . Розглянемо приклад композиції послідовності підставлень $\{X/Y, W/Z\}$, $\{V/X\}$, $\{a/V, f(b)/W\}$. Вони еквівалентні єдиному підставленню $\{a/Y, f(b)/Z\}$. Останнє підставлення було виведено шляхом компонування $\{X/Y, W/Z\}$ з $\{V/X\}$ для отримання $\{V/Y, W/Z\}$ та компонування результату з $\{a/V, f(b)/W\}$ для отримання $\{a/Y, f(b)/Z\}$.

Важлива вимога алгоритму уніфікації – уніфікатор (unifier) має бути максимально загальним, тобто для довільних двох виразів необхідно знайти *найбільш загальний уніфікатор*. У разі втрати загальності у процесі уніфікації зменшується ймовірність досягнення кінцевого розв'язку або така можливість зникає повністю. Наприклад, речення $p(X)$ та $p(Y)$ можна уніфікувати довільним константним виразом вигляду $\{\text{ivan}/X, \text{ivan}/Y\}$. Але ivan не є найбільш загальним уніфікатором. Використовуючи як уніфікатор довільну змінну, можна отримати більш загальний вираз: $\{Z/X, Z/Y\}$. Розв'язки, отримані під час використання першого підставлення, завжди будуть обмежені константою ivan . Це знижує універсальність результату.

Означення 1.15. Найбільш загальний уніфікатор.

Якщо s – довільний уніфікатор виразу E , g – найбільш загальний уніфікатор (most general unifier – mgu) цього набору виразів, тоді під час застосування s до E з'явиться ще один уніфікатор s' такий, що $Es = Egs'$, де Es та Egs' – композиції уніфікацій, які застосовані до виразу E .

Уніфікація важлива для довільної системи розв'язування задач штучного інтелекту, яка використовує як засіб наведення числення предикатів. Уніфікація визначає умови, за яких два (або більше) вирази числення предикатів можуть бути еквівалентними. Це дає змогу використовувати для логічного зображення такі правила, як *резольюція* (resolution), хоча цей процес потребує пошуку з поверненням (backtracking) для знаходження всіх можливих інтерпретацій.

Нижче буде подано псевдокод для функції *unify* (уніфікувати), за якою обчислюють підставлення уніфікації (якщо це можливо) для двох виразів числення предикатів. Функція *unify* отримує як параметри два вирази числення предикатів та повертає або найзагальніше підставлення уніфікації, або константу *FAIL* (відмова), якщо уніфікація неможлива. Ця функція визначена рекурсивно: спочатку намагаються рекурсивно уніфікувати вихідні компоненти виразів. Якщо це вдається, всі підстановки, які повертаються внаслідок цієї уніфікації, застосовують до решти виразів. Далі виконується другий рекурсивний виклик функції *unify*, у якому завершується уніфікація. Рекурсія зупиняється, коли параметром стає символ (предикат, ім'я функції, константа або змінна), або коли всі елементи виразу будуть уніфіковані.

Щоб спростити роботу з виразами, в алгоритмі застосовують змінений синтаксис. Оскільки функція *unify* дає можливість перевірити синтаксичну відповідність шаблону, в ній ігнорують відмінність між предикатами, функціями та параметрами, властивими численню предикатів. Зобразивши вираз як *список* (list), – впорядкована послідовність елементів, першим з яких є предикат, за яким йдуть його параметри, ми спростуємо роботу з виразами. Списки, взяті у круглі дужки (), елементи списку відокремлені один від

одного пробілами. Приклади виразів як у численні предикатів, так і в синтаксисі списків наведено нижче.

Синтаксис числення предикатів	Синтаксис списку
$p(a, b)$	$(p\ a\ b)$
$p(f(a), g(X, Y))$	$(p\ (f\ a)(g\ X\ Y))$

Опишемо функцію *unify*.

```

Function unify(E1, E2);
begin
  case
    E1 та E2 –      константи або порожні списки: рекурсія
зупиняється
    if E1=E2 then return { }
    else return FAIL;
    E1 – змінна:
    if E1 входить у E2 then return FAIL
    else return {E2/E1};
    E2 – змінна:
    if E2 входить у E1 then return FAIL
    else return {E1/E2};
    E1 або E2 – порожні then return FAIL %списки різних розмірів
otherwise:
                                %E1 та E2 списки
  begin
    HE1:=перший елемент E1;
    HE2:=перший елемент E2;
    SUBS1:=unify(HE1, HE2);
    if SUBS1=FAIL then return FAIL;
    TE1:=apply(SUBS1, хвіст E1);
    TE2:=apply(SUBS1, хвіст E2);
    SUBS2:=unify(TE1, TE2);
    if SUBS2=FAIL then return FAIL;
    else return composition(SUBS1, SUBS2)
  end
end
% кінець case
end
  
```


1.3.3. Приклад уніфікації

Розглянемо роботу вищеописаного алгоритму на прикладі
 $unify((parents\ X\ (father\ X)\ (mother\ igor)),\ parents\ igor\ (father\ igor)\ Y).$

Оскільки ні один параметр не є атомарним символом, у разі першого виклику функція *unify* буде намагатись рекурсивно уніфікувати перші елементи кожного виразу за допомогою виклику

$unify(parents,\ parents).$

Ця уніфікація буде виконана та буде повернено порожнє підставлення $\{\}$. У разі застосування її до решти виразів не відбудеться ніяких змін. Далі викликається

$unify((X\ (father\ X)\ (mother\ igor)),\ igor\ (father\ igor)\ Y).$

Деревоподібний опис виконання алгоритму на цьому етапі зображено на рис. 1.2.

У другому виклику *unify* ні один з виразів не є атомарним, тому алгоритм розділяє кожен вираз на його перший компонент та решту частини. Потім слідує виклик

$unify(X,\ igor).$

Цей вираз закінчується успішно, оскільки обидва вирази атомарні, а один з них – змінна. Внаслідок виклику повертається підставлення $\{igor/X\}$. Його застосовують до "хвоста" кожного виразу, а функцію *unify* – до результатів підставлення (рис. 1.2).

$unify(((father\ igor)\ (mother\ igor)),\ ((father\ igor)\ Y)).$

Унаслідок цього виклику буде отримано уніфікацію $(father\ igor)$ з $(father\ igor)$. Це зумовлює виклики

$unify(father,\ father)$

$unify(igor,\ igor)$

$unify((),\ ()).$

Усі вони успішно завершуються, повертаючи порожню множину підставлень (рис. 1.3).

Далі функцію *unify* викликають для решти виразів

$unify(((mother\ igor)),\ (Y)).$

Це спричиняє виклики

$unify((mother\ igor),\ Y).$

$unify((),\ ()).$

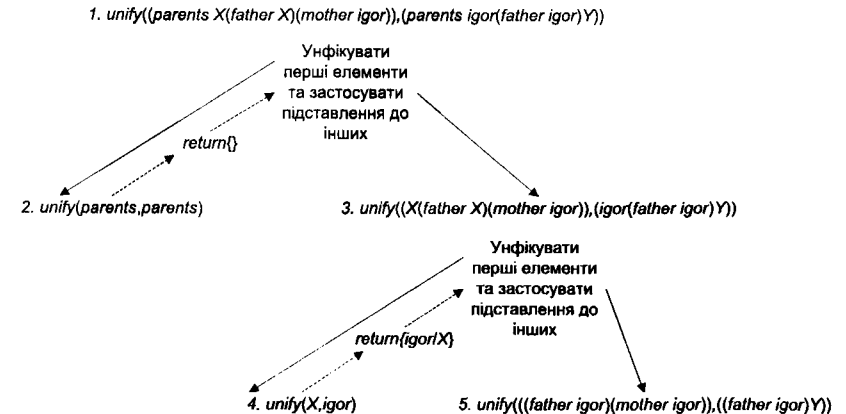


Рис. 1.2. Кроки уніфікації виразів $((parents\ X\ (father\ X)\ (mother\ igor))$ та $(parents\ igor\ (father\ igor)\ Y)$

У першому з них $(mother\ igor)$ уніфікується з Y . Другий виклик повертає $\{\}$. Комбінуючи з отриманим раніше підставленням, отримуємо відповідь $\{igor/X\ (mother\ igor)/Y\}$. Повний процес підставлень зображений на рис. 1.3. Виклики пронумеровані, щоб вказати порядок їхнього виконання. Підставлення, які повертаються кожним виразом, відображено на рисунку.

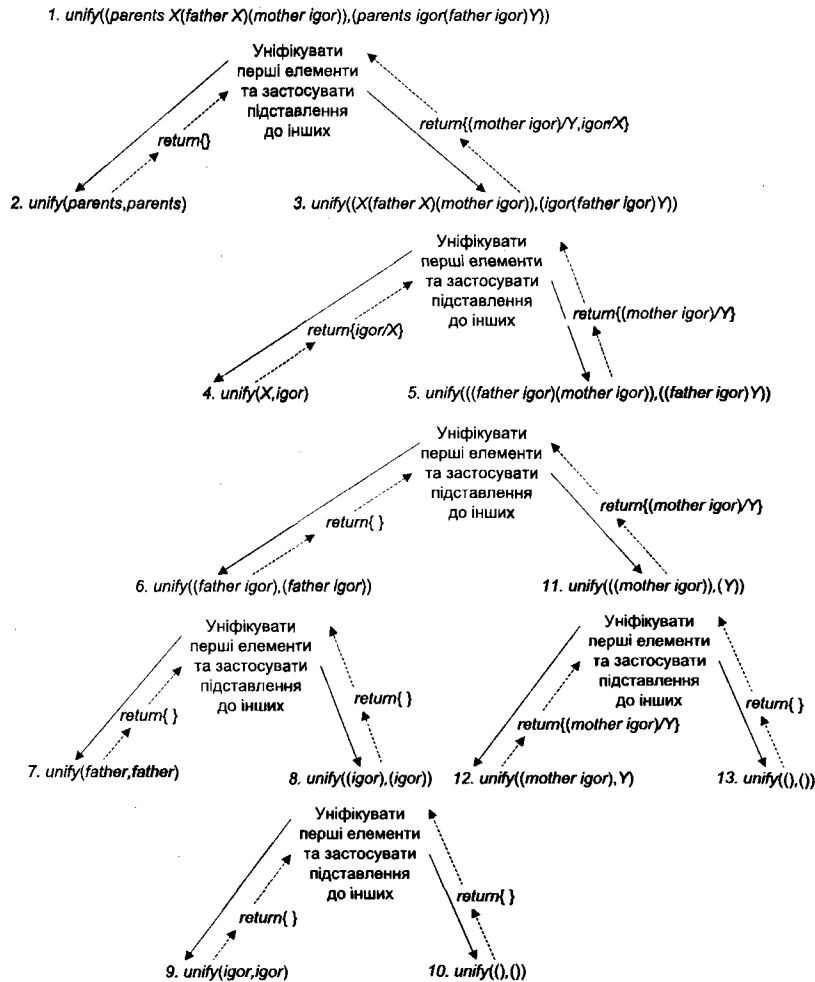


Рис. 1.3. Кінцеві кроки уніфікації виразів ((parents X (father X) (mother igor)) та (parents igor (father igor) Y))

1.4. Задачі

1. Дано висловлювання p : “Завтра буде холодно” та q : “Падатиме сніг”. Записати висловлювання $b-g$, використовуючи p , q та логічні зв’язки:

- завтра буде холодно й падатиме сніг;
- завтра буде холодно, але сніг не падатиме;
- завтра не буде холодно й не падатиме сніг;
- завтра падатиме сніг або буде холодно (або перше й друге);
- якщо завтра буде холодно, то падатиме сніг;
- завтра буде холодно або падатиме сніг, але не падатиме сніг, якщо буде холодно;

г) для того, щоб завтра було холодно, необхідно й достатньо, щоб падав сніг.

2. Побудувати таблиці істинності для кожного з висловлювань $a-f$:

- $p \rightarrow q$;
- $\bar{p} \sim q$;
- $(p \rightarrow q) \vee (\bar{p} \rightarrow q)$;
- $(p \rightarrow q) \wedge (\bar{p} \rightarrow q)$;
- $(p \sim q) \vee (\bar{p} \sim q)$;
- $(\bar{p} \sim q) \sim (p \sim q)$.

3. На основі властивості імплікації (без використання таблиць істинності та сквівалентних перетворень) довести, що формули $a-f$ є тавтологіями:

- $(p \wedge q) \rightarrow (p \rightarrow q)$;
- $p \rightarrow (p \vee q)$;
- $\bar{p} \rightarrow (p \rightarrow q)$;
- $(p \wedge q) \rightarrow p$;
- $(\neg(p \rightarrow q)) \rightarrow p$;
- $\neg(\neg(p \rightarrow q)) \rightarrow \bar{q}$.

4. Перевірити, чи сквівалентні формули $p \rightarrow (q \wedge r)$ та $(p \rightarrow q) \wedge (p \rightarrow r)$.

5. Предикат $Q(x, y)$ означає “ $x+y=x-y$ ”, а предметною областю кожної змінної є множина цілих чисел. Знайти значення істинності висловлювань $a-k$:

- $Q(1, 1)$;
- $Q(2, 0)$;
- $\forall y Q(1, y)$;
- $\exists x Q(x, 2)$;
- $\exists x \exists y Q(x, y)$;
- $\forall x \exists y Q(x, y)$;
- $\forall x \forall y Q(x, y)$;
- $\exists x \forall y Q(x, y)$;
- $\forall y \exists x Q(x, y)$;
- $\exists y \forall x Q(x, y)$.

6. Предметною областю кожної змінної предиката $P(x,y)$ є множина $\{1,2,3\}$. Записати висловлювання $a-f$ з використанням логічних зв'язок кон'юнкції та диз'юнкції:

- a) $\exists xP(x,3)$; b) $\forall yP(1,y)$; c) $\forall x\forall yP(x,y)$;
 d) $\exists x\exists yP(x,y)$; e) $\exists x\forall yP(x,y)$; f) $\forall y\exists xP(x,y)$.

7. Предикат $P(x)$ означає " $x=x^2$ ", а предметною областю змінної x є множина цілих чисел. Знайти значення істинності висловлювань $a-e$:

- a) $P(0)$; b) $P(1)$; c) $P(-1)$; d) $\exists xP(x)$; e) $\forall xP(x)$.

8. Записати заперечення висловлювань формулами логіки першого ступеня. Записати отримані висловлювання реченнями української мови:

- a) кожний студент групи любить математику;
 b) у групі є студент, який ніколи не бачив комп'ютера;
 c) у групі є студент, який прослухав усі запропоновані математичні курси;
 d) у групі є студент, який відвідав принаймні одну аудиторію кожного з навчальних корпусів університету.

9. Довести, що формули $\exists x\forall yP(x,y)$ та $\forall x\exists y\bar{P}(x,y)$ мають однакові значення істинності.

10. Довести, що формули $\forall x(P(x) \wedge Q(x))$ та $\forall xP(x) \wedge \forall xQ(x)$ мають однакові значення істинності.

11. Довести, що формули $\exists x(P(x) \wedge Q(x))$ та $\exists xP(x) \vee \exists xQ(x)$ мають однакові значення істинності.

12. Позначення $\exists!xP(x)$ відповідає реченню "У предметній області є таке єдине x , що $P(x)$ істинне". Нехай множина цілих чисел є предметною областю змінної x . Знайти значення істинності формул $a-g$:

- a) $\exists x(x > 1)$; b) $\exists x(x^2 = 1)$; c) $\exists x(x + 3 = 2x)$; d) $\exists!x(x = x + 1)$;
 e) $\exists xP(x) \rightarrow \exists x\bar{P}(x)$; f) $\exists xP(x) \rightarrow \exists!xP(x)$; g) $\exists x\bar{P}(x) \rightarrow \forall x\bar{P}(x)$.

13. Задана предметна область $M=\{1,2,3\}$ змінної x . Записати висловлювання $\exists!xP(x)$ за допомогою заперечення, кон'юнкції та диз'юнкції.

14. Розмиту логіку використовують у штучному інтелекті. Значення істинності заперечення висловлювання у розмитій логіці визначають як: одиниця мінус значення істинності цього висловлювання. Нехай висловлювання "Іван щасливий" має значення істинності 0,8, а "Сергій щасливий" – 0,4. Яке значення істинності мають висловлювання "Іван і Сергій щасливі" та "Іван і Сергій нещасливі"?

15. Значення істинності кон'юнкцій двох висловлювань у розмитій логіці визначають, як мінімум із значень істинності цих висловлювань. Нехай висловлювання "Іван щасливий" має значення істинності 0,8, а "Сергій щасливий" – 0,4. Яке значення істинності мають висловлювання "Іван і Сергій щасливі" та "Іван і Сергій нещасливі"?

16. Значення істинності диз'юнкції двох висловлювань у розмитій логіці визначають, як максимум із значень істинності цих висловлювань. Нехай висловлювання "Іван щасливий" має значення істинності 0,8, а "Сергій щасливий" – 0,4. Яке значення істинності мають висловлювання "Іван або Сергій щасливі" та "Іван або Сергій нещасливі"?

17. Обчислити значення істинності складного висловлювання $(p \vee \bar{q}) \wedge r \wedge (p \vee \bar{r})$ у розмитій логіці для таких значень істинності його атомів: $p=0,9$; $q=0,4$; $r=0,7$.

18. Обчислити значення істинності складного висловлювання $(\bar{p} \vee \bar{q}) \wedge (p \vee r) \wedge \bar{q}$ у розмитій логіці для таких значень істинності його атомів: $p=0,1$; $q=0,8$; $r=0,6$.

2. ОСНОВИ МОВИ PROLOG

Prolog – це найвідоміший приклад мови логічного програмування (logic programming language). Логічна програма – це набір специфікацій у рамках формальної логіки. Prolog заснований на теорії предикатів першого порядку. Назву мови програмування розшифровують як Programming in Logic (Програмування в логіці). Під час виконання програми інтерпретатор постійно реалізує виведення на підставі логічних специфікацій. Ідея використання можливостей наведення теорії предикатів першого порядку – це одна з головних переваг мови Prolog для комп'ютерних наук взагалі та штучного інтелекту зокрема. Застосування теорії предикатів першого порядку в мові програмування забезпечує прозорий, елегантний синтаксис та добре визначену семантику.

Розвиток мови Prolog започаткований з досліджень, пов'язаних з доказом теорем, а точніше, з розробленням алгоритмів спрощування резолюції (Robinson, 1965). У цій роботі розроблено процедуру доведення, яка названа резолюцією (resolution) та стала головним методом обчислень на мові Prolog.

Завдяки цим властивостям Prolog зарекомендував себе як корисний засіб дослідження таких експериментальних питань програмування, як автоматичне генерування коду, верифікація програм та розроблення високорівневих мов програмування. Prolog, як і інші, засновані на логіці, мови, підтримує декларативний стиль програмування, тобто конструювання програми в термінах високорівневого опису обмежень предметної області. На відміну від нього, процедурний стиль програмування допускає написання програми у вигляді послідовності інструкцій з виконання алгоритму. У логічному програмуванні комп'ютеру повідомляють “що є істинна”,

а не “як це зробити”. Це дає змогу програмістам зосередитися на розв'язуванні задачі та створенні специфікацій для предметної галузі, а не деталях написання низькорівневих алгоритмічних інструкцій вигляду “що робити далі”. Prolog охоплює механізм виведення, який оснований на співставленні зразків. За допомогою підбору відповідей на запити отримують відому інформацію. Прологівське знання про світ – це обмежений набір фактів та правил, заданих у програмі. Однією з найважливіших особливостей мови Prolog є те, що на додаток до логічного пошуку відповідей на поставлені запитання, він може мати справу з альтернативами та знаходити всі можливі розв'язки. Замість звичайної роботи програми від початку до кінця, Prolog може вертатися назад та переглядати більше одного “шляху” під час розв'язування всіх складових задачі.

Перша Prolog-програма була написана на початку 1970 років у Франції в рамках проекту щодо розуміння природної мови (Colmerauer та ін., 1973), (Roussel, 1975), (Kowalski, 1979). Теоретичні основи цієї мови описані у роботах (Kowalski, 1979), (Hayes, 1977), (Lloyd, 1984). Основний етап розвитку мови Prolog припадає на 1975–1979 рр., коли на кафедрі штучного інтелекту університету Единбурга Девід Уоррен (David Warren) і Фернандо Персейра (Fernando Pereira) відповідали за реалізацію цієї мови. Вони створили інтерпретатор мови Prolog. Ця версія стала першим стандартом та була описана у книзі (Clocksin і Mellish, 1984).

Prolog стає важливим інструментом у програмуванні задач штучного інтелекту та в розробленні експертних систем. Високий рівень абстракції, легкість та простота подання складних структур даних, змога моделювати логічні відношення між об'єктами та процесами істотно полегшують розв'язування задач у різних предметних галузях. Тому Prolog широко застосовують для створення адміністративних застосувань, web-застосувань, для керування великими та складними базами даних.

Цей розділ посібника присвячений опису основ мови Prolog 5.2.

2.1. Факти і правила

Програміст на мові Prolog спочатку описує *об'єкти* (objects) та *відношення* (relations), а далі – описує *правила* (rules), за яких ці відношення є істинними. Наприклад, речення

Сергій любить собак (Sergiy likes dogs)

встановлює відношення між об'єктами *Sergiy* та *dogs*; цим відношенням є *likes*. Нижче наведено правило, яке визначає, коли речення "*Сергій любить собак*" буде істинним:

Сергій любить собак, якщо собаки хороші (Sergiy likes dogs if the dogs are nice)

2.1.1. Факти

Відношення між об'єктами названо *фактом* (fact). Інша назва для відношень – *предикат*. Імена індивідів (*sergiy*, *dogs*) – називають *атомами* або *аргументами* предиката. *Аргумент* предиката – це кількість аргументів, які він приймає. Описуючи факти, важливо дотримуватися таких правил:

- Імена всіх відношень та об'єктів мають починатися з малої літери.
- Спочатку записують ім'я відношення, далі (у круглих дужках, через кому) імена об'єктів.
- Кожен факт закінчується крапкою.

Подаємо декілька речень на природній мові та відповідні їм факти з використанням синтаксису мови Prolog:

<i>Сергій любить Ірину</i> (Sergiy likes Iryna);	<i>likes</i> (sergiy, iryna);
<i>Ірина любить Сергія</i> (Iryna likes Sergiy);	<i>likes</i> (iryna, sergiy);
<i>Сергій любить собак</i> (Sergiy likes dogs);	<i>likes</i> (sergiy, dogs).

Факти, крім відношень, можуть передавати і властивості:

<i>Катерина – дівчина</i> (Cateryna is girl);	<i>girl</i> (caitlin);
<i>Керміт зелений</i> (Kermit is green);	<i>green</i> (kermit).

Отже, *факти* – це відношення або властивості, про які відомо, що вони мають значення "істина".

2.1.2. Змінні

Змінні слугують для позначення невизначених об'єктів. Використання змінних у логічних програмах відрізняється від використання змінних у традиційних мовах програмування. У логічних програмах змінна позначає невизначений, але не єдиний об'єкт. Імена змінних у мові Prolog мають починатися з великої літери (або з символу підкреслення "_"), за якою може стояти довільна кількість букв, цифр та символів підкреслення.

Вираз

likes (Everyone, iryna)

може описувати той факт, що "кожен любить Ірину". Або

likes (sergiy, Y)

подає множину тих людей, котрих любить Сергій.

Змінні у Prolog-програмах ініціалізуються у разі співставлення з константами у фактах та правилах. До ініціалізації змінна вільна; після присвоєння їй значення вона стає зв'язаною. Змінна залишається зв'язаною тільки той час, який необхідний для отримання розв'язку з запиту, потім Prolog звільняє її та шукає інший розв'язок.

Можна використовувати анонімні змінні, що їх позначають символом підкреслення для ігнорування непотрібних значень. Анонімна змінна може бути використана на місці довільної іншої змінної, та їй ніколи не присвоюють значення. Наприклад, такі факти:

owns (_, shoes);
eats (_)

могли бути використані для ствердження на природній мові:

У кожного є туфлі. (Everyone owns shoes).

Кожен їсть. (Everyone eats).

Анонімні змінні співставляють з довільними даними.

2.1.3. Правила

Правила дають змогу вивести один факт на підставі інших. Інакше кажучи, *правило* – це відношення, для якого відомо, що во-

но достовірне, якщо одне або декілька інших знайдених відношень є достовірними. Подаємо правила, які відповідають зв'язку "любити" (likes):

Iryna любить все, що любить Sergiy (Iryna likes everything that Sergiy likes);

Kateryna любить все зелене (Kateryna likes everything that is green).

Запишемо ці речення мовою Prolog:

likes (iryna, Something) :- likes (sergiy, Something);

likes (kateryna, Something) :- green (Something).

Символ ":" має зміст "якщо" та поділяє правило на дві частини: заголовок і тіло. **Заголовок** – це факт, який був би істинним, якщо б були істинними декілька умов. **Тіло** – це низка умов, які мають бути істинними, щоб можна було довести, що заголовок правила вірогідний. Отже, усі ці правила означають: "Щоб довести, що Ірина любить щось, треба довести, що Сергій любить це" та "Щоб довести, що Катерина любить щось, треба довести, що це щось зелене". Умов у тілі правила може бути декілька. Їх відокремлюють комою у разі кон'юнкції та крапкою з комою у разі диз'юнкції.

Розглянемо інший приклад:

Людина може купити машину, якщо машина подобається їй (likes), *якщо машина продається* (for sale). Це відношення на мові Prolog може бути записано так:

can_buy (Name, Model):-

person(Name),car(Model),likes(Name,Model),for_sale(Model).

Символ "," означає кон'юнкцію декількох тверджень. Твердження виражає такі відношення:

Name може купити (can_buy) Model, якщо

Name є людиною (person), та

Model є машиною (car), та

Name подобається (likes) Model, та

Model продається (for_sale).

Це правило буде істинним, якщо достовірні всі чотири умови у тілі правила. Достовірність умов у тілі правила перевіряють зліва направо.

2.1.4. Запити

Програма на мові Prolog – це набір специфікацій із логіки предикатів першого порядку, які описують об'єкти та відношення між ними у предметній галузі задачі. Набір специфікацій називають *базою даних* (database) конкретної задачі. Інтерпретатор Prolog відповідає на питання, що стосуються набору специфікацій. Запити до бази даних – це шаблони, які застосовують у тому самому синтаксисі, що і записи бази даних. Інтерпретатор обробляє запити, виконуючи пошук у базі даних у глибину зліва направо та визначаючи, чи є цей запит логічним наслідком специфікацій із бази даних.

Отже, *запит* (ціль) – це засіб добування інформації з логічної програми. За допомогою запиту визначають, чи виконано деяке відношення між об'єктами. Наприклад:

Does Sergiy like Iryna? (Чи Сергій любить Ірину?).

За правилами Prolog запишемо:

likes (sergiy, iryna).

Оскільки є такий факт, результат – yes(так).

Використовуючи змінну, можна задати таке питання:

What does Sergiy like? (Що любить Сергій?)

likes (sergiy, What).

Prolog завжди шукає відповідь на запит, починаючи з першого факту, та перебираючи всі факти, поки вони не закінчаться. На задане запитання отримаємо відповідь:

What=iryna

What=dogs

2 Solutions.

Оскільки були відомі факти likes (bill, iryna) та likes (bill, dogs).

Крім простих запитів, можна задавати складені, які містять декілька частин, відокремлених комою у разі кон'юнкції та крапкою з комою у разі диз'юнкції. Наприклад, можна було б задати питання:

likes (X, iryna), likes (X, dogs).

(Чи такий X, який любить ірину та dogs)

Отримаємо:

X=sergiy

1 Solutions.

2.2. Основні розділи Visual Prolog-програм

На відміну від інших версій, Visual Prolog – компілятор, який контролює типи: для кожного предиката оголошують типи об'єктів, які він може використовувати. Це оголошення типів дає змогу програмам бути скомпільованими безпосередньо у машинні коди. Як наслідок: швидкість виконання порівняна, а деколи і перевищує швидкості аналогічних програм на мовах C++ та Pascal.

До головних розділів Visual Prolog програм відносять:

розділ **clauses** (речень);

розділ **predicates** (предикатів);

розділ **domains** (доменів);

розділ **goal** (цілей).

Розділ **clauses** – це серце Visual Prolog-програми; саме у цьому розділі записують факти та правила, якими буде оперувати програма, намагаючись вирішити поставлену ціль.

Розділ **predicates** – це той, у якому оголошують предикати та домени (типи) їхніх аргументів (вбудовані у Visual Prolog предикати оголошувати не потрібно). Якщо у розділі **clauses** був описаний власний предикат, то його обов'язково потрібно оголосити у розділі предикатів. Унаслідок оголошення предикатів ви повідомляєте, до яких доменів (типів) належать аргументи цього предиката.

Розділ **domains** – слугує для оголошення доменів усіх аргументів предикатів, які використовують (стандартні домени оголошувати не потрібно). Відомі стандартні домени, які можна використовувати у разі декларування типів аргументів предикатів без опису в розділі **domains**.

Розділ **goal** – це розділ, у якому розміщено ціль Visual Prolog-програми. Запит програми може складатися зі списку підцілей. Якщо всі підцілі у розділі **goal** істинні, то програма завершується успішно. Якщо ж одна з підцілей неуспішна, то вважають, що програма завершується хибно.

Таблиця 3.1.

Основні стандартні домени.

Домен	Опис та реалізація
short	Коротке, знакове, ціле. Усі платформи 16 бітів (-32 768–32 767)
ushort	Коротке, беззнакове, ціле. Усі платформи 16 бітів (0–65 535)
long	Довге, знакове, ціле. Усі платформи 32 біти (-2 147 483 648–2 147 483 647)
ulong	Довге, беззнакове, ціле. Усі платформи 32 біти (0–4 294 967 295)
integer	Знакове ціле. Платформи 16 бітів (-32 768–32 767) Платформи 32 біт (-2 147 483 648–2 147 483 648)
unsigned	Беззнакове ціле. Платформи 16 бітів (0–65 535) Платформи 32 біти (0–4 294 967 295)
byte	Усі платформи 8 бітів (0–55)
word	Усі платформи 16 бітів (0–65 535)
dword	Усі платформи 32 біти (0–4 294 967 295)
char	Символ, реалізований як беззнаковий byte
real	Число з плаваючою крапкою (8 байтів). Еквівалентний типу double у мові C
string	Послідовність символів, реалізована як вказівник на масив символів, який закінчується нулем
symbol	Послідовність символів, реалізована як вказівник на вхід у таблиці ідентифікаторів, яка зберігає рядки ідентифікаторів

Розглянемо простий приклад програми для додавання та множення двох чисел.

Приклад 2.1.

```
domains
    product,sum = integer
predicates
    add_em_up(sum,sum,sum)
    multiply_em(product,product,product)
clauses
    add_em_up(X,Y,Sum):-
        Sum=X+Y.
    multiply_em(X,Y,Product):-
        Product=X*Y.
goal
    add_em_up(32,54,Sum).
```


Результатом виконання програми буде:

Sum=86

1 Solution.

Якщо у розділі goal задати multiply_em (31,13, Product), тоді отримаємо Product=403.

Припустимо, що ми хочемо подвоїти добуток 31 на 17. Задамо ціль:

multiply_em (31,17,Sum), add_em_up (Sum,Sum,Answer).

У результаті отримаємо помилку приведення типів, оскільки результуюче значення предиката multiply_em належить до домена product, а перші два значення предиката add_em_up до домена sum. Отже, якщо змінну у реченні використовують більше ніж в одному предикаті, вона має бути однаково оголошена у кожному з них.

2.3. Уніфікація та пошук з поверненням

Розв'язуючи реальну задачу, потрібно притримуватися певного шляху для її логічного завершення. Якщо отриманий результат не дає шуканої відповіді, вибирають інший шлях. Prolog, шукаючи розв'язок задачі, використовує метод спроб та повернень назад; цей метод названо *пошуком з поверненням*. Якщо, починаючи пошук розв'язку задачі, Prolog має вибрати між альтернативними шляхами, то він ставить маркер у місці розгалуження (яке названо *точкою повернення*) та вибирає першу підциль для перевірки. Якщо ця підциль не виконається, Prolog вернеться до точки повернення та спробує перевірити іншу підциль. Розглянемо головні правила пошуку з поверненням на прикладі:

Приклад 2.2.

```
domains
  name, thing = symbol
predicates
  likes (name, thing)
  reads (name)
  is_inquisitive (name)
clauses
  likes (petro, wine).
```

```
likes (galyna, skiing).
likes (galyna, books).
likes (galyna, films).
likes (Z, books):-
  reads (Z),
  is_inquisitive (Z).
reads (petro).
is_inquisitive (petro).
goal
likes (X, wine), likes (X, books).
```

Процес пошуку будемо зображати у вигляді цільового дерева. Перед початком дослідження цільового твердження дерево складається з двох частин неузгоджених підцилей (рис. 2.1). У наступних зображеннях цільового дерева узгоджену підциль помітимо підкресленням, а відповідне речення – запишемо під нею.

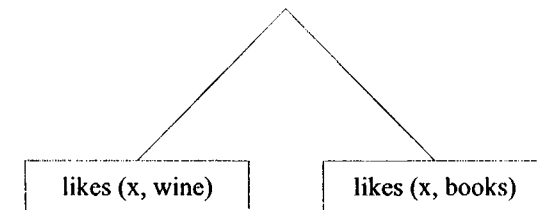


Рис. 2.1. Цільове дерево процесу пошуку

Правило 1.

Підцилі мають бути узгоджені за порядком, зверху вниз.

У цьому прикладі цільове дерево засвідчує, що мають бути узгоджені дві підцилі. Prolog визначає, яку підциль йому використовувати у разі спроби співставлення речення, виходячи із другого основного правила пошуку з поверненням.

Правило 2.

Предикатні речення перевіряють у тому порядку, в якому вони з'являються у програмі: зверху вниз.

Під час виконання програми підциль likes (X, wine) відповідає факту likes (petro, wine), та X зв'язується зі значенням petro (рис. 2.2).

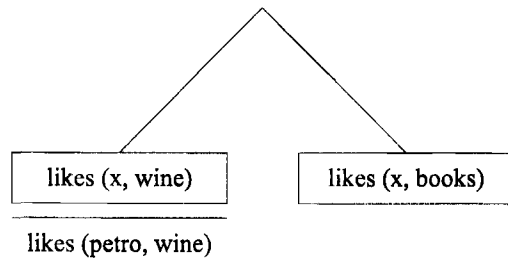


Рис. 2.2. Цільове дерево 2

Звернення до другої підцілі починає новий пошук з умовою $X=petro$. Пошук продовжується до речення $likes(Z,books):-reads(Z),is_inquisitive(Z)$. Унаслідок *уніфікації* аргументів змінна Z набуває значення $petro$. Тепер підціль відповідає лівій частині (заголовок правила). Продовження пошуку визначено третім фундаментальним правилом пошуку з поверненням.

Правило 3.

Коли підціль відповідає заголовку правила, необхідно узгодити тіло цього правила: тіло правила утворює нову множину підцілей для узгодження.

Цільове дерево набуде вигляду, зображеного на рис. 2.3.

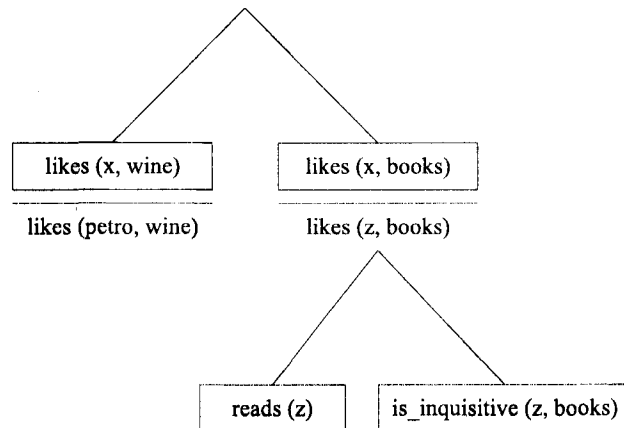


Рис. 2.3. Цільове дерево 3

Тепер необхідно вирішити дві підцілі $reads(Z)$ та $is_inquisitive(Z)$. Останнє результуюче цільове дерево зображено на рис. 2.4.

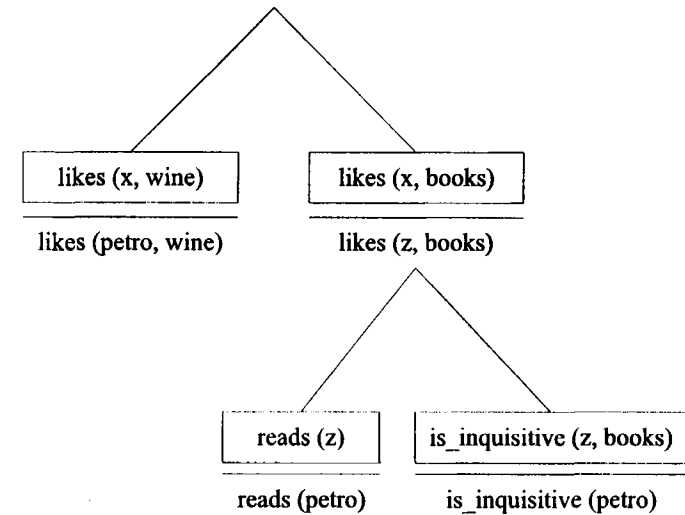


Рис. 2.4. Цільове дерево 4

Правило 4.

Цільове твердження вважають узгодженим, якщо відповідний факт знайдено для кожного листа цільового дерева.

Результатом виконання програми стане $X=petro$.

Виконуючи підціль, Prolog починає пошук з першого речення, визначаючого предикат. Результатом може бути один з двох випадків:

1. Prolog знаходить відповідне речення, тоді:

якщо є інше речення, яке, можливо, зможе ще узгодити підціль, Prolog виставляє вказівник (щоб відмітити точку повернення) та зв'язує всі вільні змінні у підцілі (які відповідають значенням у реченні) з відповідними значеннями;

якщо речення є заголовком правила, тоді оцінюють тіло цього правила. Підцілі в тілі правила мають бути задовільнені для успішного завершення звернення.

2. Prolog не може знайти відповідне речення. Цільове твердження не узгоджується, та Prolog виконує пошук з поверненням у спробі знову узгодити попередню підциль. Коли процес досягає останньої точки повернення, Prolog звільняє всі змінні, яким були присвоєні нові значення (після того, як була поставлена точка повернення), і знову намагається узгодити вихідне звернення.

Prolog починає пошук з першого рядка програми. Коли він повертається до звернення, новий процес пошуку починається з точки повернення, що була встановлена останньою. Якщо пошук безспішний, то знову починається пошук з поверненням. Коли процес пошуку вичерпав усі речення для всіх підцилей, тоді це означає, що цільове твердження залишається неузгодженим.

2.4. Керування пошуком рішень

Вбудований механізм пошуку з поверненням може привести до пошуку непотрібних рішень. Інколи може виникнути потреба продовжувати пошук додаткових рішень, навіть якщо цільове твердження вже узгоджене. Prolog забезпечує два інструментальні засоби, які дають змогу керувати механізмом пошуку з поверненням: предикат **fail**, який використовують для ініціалізації пошуку з поверненням, та **cut**, або відсікання (позначено **!**) – для заборони можливості повернення.

Prolog починає пошук з поверненням, коли виклик закінчується невдачею. У деяких випадках буває необхідно ініціювати виконання пошуку з поверненням, щоб знайти інше рішення. Для цього є спеціальний предикат **fail**, який завжди закінчується невдачею та ініціює повернення. Розглянемо такий приклад.

Приклад 2.3.

```
domains
  name = symbol
predicates
  father (name, name)
  everybody
clauses
```

```
father (leonid, kateryna).
father (mykola, olena).
father (mykola, maryna).
everybody:-
  father (X, Y),
  write (X," is ",Y,"'s father\n"),
  fail.
```

goal

everybody.

Результатом програми буде:

```
leonid is kateryna father
mykola is olena father
mykola is marilyn's father.
```

Предикат **everybody** використовує пошук з поверненням, щоб отримати всі розв'язки для **father (X, Y)**, заставляючи Prolog виконати пошук з поверненням крізь тіло правила **everybody**. В разі пошуку з поверненням відбувається повернення до останнього звернення, яке може призвести до множинного рішення. Таке звернення називають *недетермінованим*. Недетерміноване звернення є протилежним до *детермінованого*, результатом якого може бути лише одне рішення. Предикат **write** видруковує свої аргументи на екран та не може запропонувати нових рішень, тому Prolog має зробити повернення далі, до першої підцилі у правилі.

Prolog використовує можливість *відсікання*, яке необхідне для переривання пошуку з поверненням; відсікання позначено знаком оклику (**!**). Через відсікання неможливо здійснити повернення (пошук з поверненням).

Відомі два головні випадки застосування відсікання.

1. Якщо заздалегідь відомо, що визначені посилання ніколи не приведуть до змістовного рішення, необхідно застосувати відсікання. Програма стане швидшою та економнішою. Такий прийом названо *зеленим відсіканням*.

2. Якщо відсікання потребує сама логіка програми для виключення із розгляду альтернативних підцилей, то – *червоне відсікання*.

2.4.1. Використання відсікань

Розглянемо деякі приклади використання відсікань.

1. Запобігання пошуку з поверненням до попередньої підцілі у правилі:

```
r1 :- a, b, !, c.
```

Такий запис дає змогу повідомити Prolog про те, що вас задовольнить перше рішення, знайдене для підцілей *a* та *b*. Маючи змогу знайти множинні рішення під час звернення до *c* шляхом пошуку з поверненням, Prolog у цьому разі не може зробити повернення через відсікання та знайти альтернативне рішення для звернень *a* і *b*. Він також не може повернутися до іншого речення, яке визначає предикат *r1*.

2. Запобігання пошуку з поверненням до наступного речення.

Відсікання може бути використано, як спосіб повідомити Prolog, що він вибрав правильне речення для визначеного предиката. Наприклад, розглянемо такий фрагмент:

```
r(1) :- !, a, b, c.
```

```
r(2) :- !, d.
```

```
r(3) :- !, c.
```

```
r(_) :- write("This is catchall clause.").
```

Використання відсікання робить предикат *r* детермінованим. Припустимо, що відбулося звернення *r(1)*. Prolog переглядає програму в пошуку відповідності для звернення; він знаходить її з першим реченням, яке визначає *r*. Оскільки є більше ніж одне можливе рішення для звернення, Prolog ставить точку повернення біля цього речення. У тілі правила відсікання відмінняє точки пошуку з поверненням, підвищуючи ефективність виконання програми, а також гарантує, що останнє речення буде виконано лише у тому разі, коли ні одна з умов не відповідала зверненню до *r*. Конструкція такого типу подібна до конструкції *case* в інших мовах програмування.

2.4.2. Предикат *not*

Приклад свідчить, як можна використати предикат *not* для того, щоб виявити успішного студента: студента, в якого середній

бал (GPA) не менше 3,5 і в якого у цей час не продовжено випробувальний термін.

Приклад 2.4.

```
domains
    name = symbol
    gpa = real
predicates
    honor_student(name)
    student(name, gpa)
    probation(name)
clauses
    honor_student(Name):-
        student(Name, GPA), GPA>=3.5, not(probation(Name)).
    student("Betty Blue", 3.5).
    student("David Smith", 2.0).
    student("John Johnson", 3.7).
    probation("Betty Blue").
    probation("David Smith").
goal
    honor_student(X).
```

Результат програми: X=John Johnson.

Використовуючи предикт *not*, необхідно пам'ятати:

предикат *not* буде успішним, якщо не може бути доведена істинність підцілі.

У разі виклику *not* з вільними змінними, Prolog поверне повідомлення про помилку: "Free variable are not allowed in not" (Вільні змінні в *not* заборонені).

2.5. Рекурсія

Рекурсія – добре відомий спосіб для опису задач, які містять у собі підзадачу такого самого типу. Наприклад, пошук у дереві (дерево складається з дрібніших дерев) та рекурсивне сортування (для сортування списку, його поділено на частини, які сортують та об'єднують разом).

Рекурсивна процедура – це процедура, що викликає сама себе. У рекурсивній процедурі немає проблем із запам'ятовуванням ре-

результатів її виконання, тому що довільні обчислені значення можна передавати з одного виклику в інший як аргументи предиката, який викликають. Розглянемо приклад рекурсивної програми для задачі обчислення факторіалу заданого числа.

Приклад 2.5.

```

predicates
  factorial (unsigned, real)
clauses
  factorial (1, 1):-!.
  factorial (X, FactX):-
    Y=X-1,
    factorial (Y, FactY),
    FactX = X*FactY.

goal
  X=3,
  factorial (X, FactX).
```

Виконання програми відбувається поетапно:

Прямий хід

Перший крок – це виклик factorial (3, FactX). У програмі спрацьовує другий предикат: X=3, FactX уніфікується з FactX, Y=2.

Далі відбувається рекурсивний виклик factorial(2,FactY). Спрацьовує знову другий предикат, у цьому разі всі аргументи і проміжкові змінні копіюються у стек: X1=2, FactY уніфікується з FactX, Y1=1 (X1, Y1 – нові копії змінних).

Останнім викликом стане factorial (1, FactY1) (FactY1 – нова копія змінної). Після цього виклику спрацьовує перший предикат, FactY1 уніфікується з 1 та відсікаються повернення.

Зворотний хід

```

FactY=X1* FactY1=2*1.
FactX= X*FactY=3*2*1.
```

Недоліком рекурсії є втрати пам'яті, оскільки у разі кожного виклику копіюються змінні у стек. Розглянемо спеціальний випадок, коли процедура може викликати себе без збереження інформації про свій стан. Припустимо, що на останньому кроці виконання процедура В викликає сама себе. Коли В викликає В, стек для В, яка викликає, має бути замінений стеком для викликаного В. У цій операції аргументам присвоюються нові значення і далі

виконання процесу повертається до початку процедури В. Цю операцію названо *оптимізацією хвостової рекурсії* (tail recursion optimization), або оптимізацією останнього виклику (last-call optimization).

Для того, щоб задати хвостову рекурсію мають бути виконані умови:

- Виклик є останньою підциллю речення.
- Раніше у реченні не було точок повернення.

Подаємо приклад, який задовільняє обом умовам.

Приклад 2.6.

```

predicates
  count (ulong)
clauses
  count (N):-
    write ('r', N),
    NewN = N+1,
    count (NewN).

goal
  nl,
  count (0).
```

Предикат count(0) друкуватиме цілі числа, починаючи з 0, та зупиниться, коли відбудеться переповнення типу даних. Запас пам'яті залишається незмінним.

Розглянемо три помилкові способи організування хвостової рекурсії.

1. Якщо рекурсивний виклик не останній крок, тоді процедура не буде хвостовою рекурсією. Наприклад:

```

badcount1 (X):-
  write ('r', X),
  NewX = X+1,
  badcount1 (NewX),
  nl.
```

Кожен раз, коли badcount1 викликає сама себе, стек має бути збережений для того, щоб оброблення можна було повернути до процедури, що викликає та має виконуватись до nl.

2. Інший спосіб зробити хвостову рекурсію не оптимізованою – залишити можливу альтернативу неперервною до моменту ви-

конання рекурсивного виклику. Тоді стек має бути збережений, оскільки у випадку невдалого завершення рекурсивного виклику, процедура, яка викликає, може повернутися та почати перевіряти цю альтернативу. Наприклад:

```
badcount2 (X):-
    write ('\r', X),
    NewX = X+1,
    badcount2 (NewX).
badcount2 (X):-
    X < 0,
    write ("X is negative").
```

У цьому випадку перше речення badcount2 викликає себе, коли друге речення ще не виконано.

Для оптимізованої хвостової рекурсії перемістимо перевірку з другого речення в перше та застосуємо оператор відсікання:

```
cutcount2 (X):-
    X>=0,!
    write ('\r', X),
    NewX = X + 1,
    cutcount2 (NewX).
cutcount2(_):-
    write ("X is negative").
```

У випадку, коли $X \geq 0$, друге речення не має виконуватись, тому був застосований оператор відсікання.

3. Для втрати оптимізації хвостової рекурсії не обов'язково мати неперевірену альтернативу як окреме речення рекурсивної процедури. Неперевірена альтернатива може бути і в довільному предикаті, який викликається. Наприклад:

```
badcount3 (X):-
    write ('\r',X),
    NewX = X+1,
    check (NewX),
    badcount3 (NewX).
check (Z):-
    Z >= 0.
check (Z):-
    Z < 0.
```

Припустимо, що X – додатна величина. Коли badcount3 викликає себе, перше речення check досягає цілі, а друге речення check ще не перевірено. Тому badcount3 має зберегти копію свого стекового фрейму, щоб мати змогу повернутися та почати перевіряти друге речення check у випадку, якщо рекурсивний виклик завершиться невдало.

Підправити badcount3 можна так:

```
cutcount3 (X):-
    write ('\r', X),
    NewX = X+1,
    check (NewX),
    !,
    cutcount3 (NewX).
```

У цьому випадку оператор відсікання відмінює всі альтернативні рішення для check, а фрейм стека не потрібен, рекурсивний виклик може вільно йти далі.

Приклади badcount2 та badcount3 генерують точки повернення. Для їхньої оптимізації можна застосувати оператор відсікання (як було показано вище). У першому прикладі відсікання не допоможе, оскільки створення копій фреймів стека не пов'язано з неперевіреними альтернативами. Єдиний спосіб удосконалити badcount1 – це зробити обчислення таким чином, щоб рекурсивний виклик відбувався в кінці речення.

2.6. Структури даних

2.6.1. Складені об'єкти даних

Складені об'єкти даних дають змогу інтерпретувати деякі частини інформації як єдине ціле так, щоб потім їх було легко розділити знову. Візьмемо, наприклад, дату "лютий 14, 2005". Вона складена з трьох частин інформації – місяць, день та рік. Можна оголосити домен, який містить складений об'єкт date:

domains

date_cmp = date (string, unsigned, unsigned),

а далі записати:

D = date ("february", 14,2005).

Це об'єкт даних, який можна обробляти разом з символами та числами. Він починається з імені, що названо **функтором** (тут `date`), за яким слідують три аргументи.

Аргументи складеного об'єкта можуть самі бути складеними об'єктами. Наприклад, можна розглянути день народження як інформацію з такою структурою:

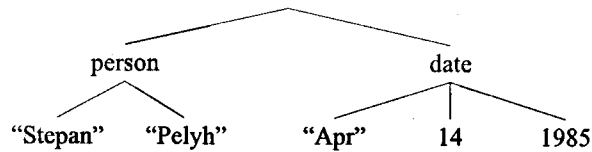


Рис. 2.5. Деревоподібна структура дати народження

На мові Prolog це виглядає так:

```
birthday (person ("Stepan", "Pelyh"), date ("Apr", 14, 1985))
```

У складеного об'єкта `birthday` в прикладі є дві частини: об'єкт `person ("Stepan", "Pelyh")` та об'єкт `date ("Apr", 14, 1985)`. Функторами для цих об'єктів будуть `person` та `date`.

Складений об'єкт може бути уніфікований зі змінною або з складеним об'єктом (який, можливо, містить змінні як частини у внутрішній структурі). Наприклад:

```
date ("Apr", 14, 1985)
```

зіставляється з `X` та присвоює `X` це значення.

Також `date ("Apr", 14, 1985)` зіставляється з `date (Mo, Da, Yr)` та присвоює змінним `Mo = "April"`, `Da = 14` та `Yr = 1985`. Prolog здійснює уніфікацію у двох випадках. По-перше, коли ціль зіставляється з заголовком речення. По-друге, через знак рівності (`=`), який є **інфіксним предикатом**. Інша програма перевіряє, чи збігаються прізвища двох людей та дає другій людині таку саму адресу, як і в першій.

Приклад 2.7.

```
domains
person      = person(name,address)
name        = name(first,last)
address     = addr(street,city,state)
```

```
street      = street(number,street_name)
city,state,street_name = string
first,last  = string
number     = integer
goal
P1 = person(name("jim","mos"),
            addr(street(5,"1st st"),"igo","CA")),
P1 = person(name(_, "mos"),Address),
P2 = person(name("jane","mos"),Address),
write("P1=",P1),nl,
write("P2=",P2),nl.
```

Важливою особливістю складених об'єктів є те, що вони дають можливість легко передавати групи величин як один аргумент.

У програмі можуть бути визначені складені домени. Наведемо загальний синтаксис їхнього опису:

```
domain = alternative1(D, D, ...);
        alternative2(D, D, ...);
```

...

Тут `alternative1` та `alternative2` – допустимі (але різні) функтори. Альтернативи розділено крапкою з комою. Запис `(D, D, ...)` – це список імен domenів, які оголошені десь у програмі або є стандартними типами domenів.

Ще один приклад використання складеного домена, де речення `your_age` (ваш вік) отримує аргумент типу `age` (вік), який може мати тип `string`, `real`, або `integer`.

Приклад 2.8.

```
domains
age = i(integer); r(real); s(string)
predicates
your_age(age)
clauses
your_age(i(Age)) :- write(Age).
your_age(r(Age)) :- write(Age).
your_age(s(Age)) :- write(Age).
Prolog не допускає наступний опис домену:

domains
age = integer; real; string % заборонено
```


Дозволено конструювати складені домени на декількох рівнях:

```
domains
articles = book(title, author);...      % Перший рівень
author = author(first_name, last_name) % Другий рівень
title, first_name, last_name = symbol  % Третій рівень
```

На рис. 2.6 наведено дерево багаторівневого складеного об'єкта.

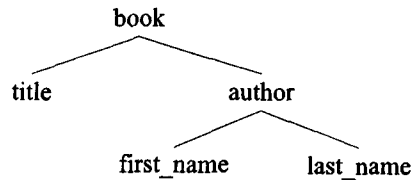


Рис. 2.6. Дерево багаторівневого складеного об'єкта

Декларація домену визначає тільки один рівень дерева, а не все дерево. Наприклад, book не може бути описаний таким реченням:

```
domains
book = book (title, author (first_name, last_name))    % Невірно
```

2.6.2. Деревя

Деревя відносять до *рекурсивних структур* даних. У Prolog дерево можна визначити так:

```
domains
treetype = tree (string, treetype, treetype); empty
```

Ця декларація свідчить про те, що дерево можна записувати як функтор *tree*, аргументами якого є рядок та два інші дерева, або як функтор *empty*, який визначає порожнє дерево. Якщо структуру описати без функтора *empty*, дерево поширюватиметься до безмежності та не буде способу закінчити рекурсію.

На рис. 2.7 наведено приклад одного з дерев.

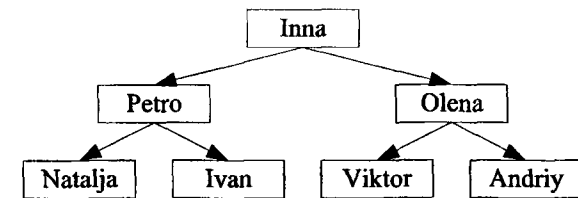


Рис. 2.7. Частина хронологічного дерева

У Prolog-програмі це дерево можна описати так:

```
tree ("Inna
tree ("Petro
tree ("Natalja", empty, empty),
tree ("Ivan", empty, empty)),
tree ("Olena",
tree ("Viktor", empty, empty),
tree ("Andriy", empty, empty))).
```

2.6.3. Обхід дерева

Однією з найважливіших операцій з деревом є відвідування всіх вузлів та оброблення їх певним чином або пошук деякого значення, або збір усіх значень. Ці процедури відомі як обхід дерева. Основний алгоритм для цього такий:

У разі, якщо дерево порожнє, не виконувати жодних дій.

Якщо не порожнє обробити поточне значення, далі перейти на ліве під-дерево, потім перейти на праве піддерево.

Як і саме дерево, алгоритм є рекурсивним, його записують двома реченнями:

```

traverse (empty).                % нічого не робити
traverse (tree (X, Y, Z)) :-
do_something_with_X,
traverse (Y),
traverse (Z).
```

Цей алгоритм відомий як обхід дерева у прямому порядку (зверху вниз, preorder). Шлях обходу дерева зображено на рис. 2.8.

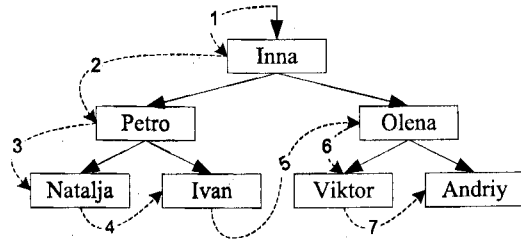


Рис. 2.8. Обхід дерева вглибину

Розглянемо програму, яка дає змогу переглянути та надрукувати всі вузли дерева.

Приклад 2.9.

```
domains
  treetype = tree (string, treetype, treetype); empty ()
predicates
  traverse (treetype)
clauses
  traverse (empty).
  traverse (tree (Name, Left, Right)):-
    write (Name, '\n'),
    traverse (Left),
    traverse (Right).
goal
  traverse (tree ("Inna"),
    tree ("Petro"),
    tree ("Natalja", empty, empty),
    tree ("Ivan", empty, empty)),
  tree ("Olena",
    tree ("Viktor", empty, empty),
    tree ("Andriy", empty, empty)))).
```

Дерево програма роздрукує так:

```
Inna
Petro
Natalja
Ivan
Olena
Viktor
Andriy.
```

2.6.4. Створення дерева

Один зі шляхів створення дерева – це вкладена структура з функторів та аргументів, як це було описано у попередньому прикладі. Загалом дерево створюють шляхом обчислень. На кожному кроці порожнє піддерево замінюють непорожнім у процесі уніфікації.

Створення дерева з одного вузла можна задати так:

```
create_tree (N, tree (N, empty, empty)).
```

Це можна розтлумачити як: "Якщо N поле даних, тоді tree (N, empty, empty) – це дерево з одного вузла, що містить дані N".

Для побудови структури дерева нам потрібні процедури, які будуть додавати новий вузол як ліве або праве піддерево. Наступній процедурі потрібні три дерева як аргументи. Вона вставляє перше дерево як ліве піддерево у друге, та результат присвоює третьому дереву:

```
insert_left (X, tree (A, _, B), tree (A, X, B)).
```

У цьому реченні немає тіла. Все, в чому полягає суть дії комп'ютера – це з'єднати аргументи одні з іншими у правильному порядку. Щоб додати праве піддерево, можемо скористатися таким реченням:

```
insert_right (X, tree (A, B, _), tree (A, B, X)).
```

Припустимо, ми хочемо вставити tree ("Petro", empty, empty) як ліве піддерево для tree ("Inna", empty, empty). Щоб це зробити, потрібно виконати цільове твердження:

```
insert_left (tree ("Petro", empty, empty),
  tree ("Inna", empty, empty),
  T).
```

Тоді T набуде значення: tree ("Inna", tree ("Petro", empty, empty), empty).

В іншому прикладі показано створення дерева, зображеного на рис. 2.7.

Приклад 2.10.

```
domains
  treetype = tree (string, treetype, treetype); empty ()
predicates
  create_tree (string, treetype)
```

```

insert_left (treetype, treetype, treetype)
insert_right (treetype, treetype, treetype)
run
clauses
create_tree (A, tree (A, empty, empty)).
insert_left (X, tree(A,_,B), tree(A, X, B)).
insert_right (X, tree (A, B,_), tree (A, B, X)).
run:-
/*Спочатку створюємо декілька одновузлових дерев*/
create_tree ("Natalja", Nt),
create_tree ("Ivan", Iv),
create_tree ("Petro", Pt),
create_tree ("Viktor", Vk),
create_tree ("Andriy", An),
create_tree ("Olena", Ol),
create_tree ("Inna", In),
/* потім їх з'єднуємо */
insert_left (Nt, Pt, Pt2),
insert_right (Iv, Pt2, Pt3),
insert_left (Vk, Ol, Ol2),
insert_right (An, Ol2, Ol3),
insert_left (Pt3, In, In2),
insert_right (Ol3, In2, In3),
/* виводимо результат */
write (In3, '\n').
goal
run.

```

У Prolog немає можливості змінити значення змінної після того, як присвоєння відбулося. Тому в програмі використовують багато змінних. Кожен раз, коли потрібно отримати нове значення, необхідна нова змінна. Але звично велика кількість імен змінних не потрібна, оскільки зазвичай процедури отримують нові змінні, викликаючи себе рекурсивно, кожен виклик має визначений набір змінних.

2.6.5. Бінарні пошукові дерева

У деревах є можливість зберігати дані так, щоб їх можна було швидко відшукати. Дерево, побудоване для такої мети, названо *пошуковим деревом*. У звичайному дереві, щоб знайти деяке зна-

чення, потрібно переглянути всі вузли. Час, який витрачається на пошук у звичайному дереві з N елементів, в середньому пропорційний N .

Бінарне пошукове дерево будують таким чином, щоб, від поточного вузла, можна було знайти, в якому з вузлів дерева знаходиться задане значення. Це робиться заданням відношення порядку між вузлами. Значення у лівому піддереві менші за поточний вузол, а у правому – більші (рис. 2.9).

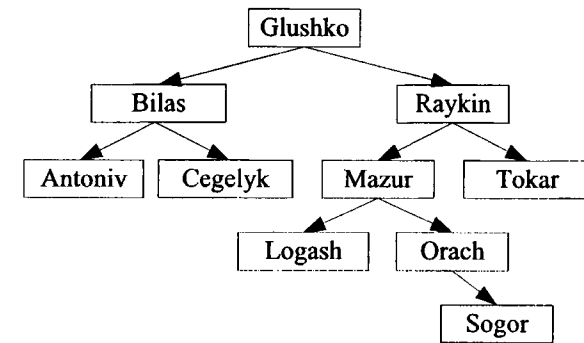


Рис. 2.9. Бінарне пошукове дерево

Час, необхідний для пошуку значення у бінарному пошуковому дереві, в середньому пропорційний $\log_2 N$. Алгоритм побудови дерева складається з таких кроків:

1. Якщо поточний вузол є порожнє дерево, тоді вставити у нього значення.
2. В іншому випадку, порівняти значення, яке необхідно вставити, зі значенням у поточному вузлі. Вставити значення у ліве або праве піддерево залежно від результату порівняння.

Для реалізації цього алгоритму необхідно записати три речення. Перше набуде вигляду:

```
insert (NewItem, empty, tree (NewItem, empty, empty)) :- !.
```

Це означає: "Результатом вставки NewItem (нового значення) в empty (порожнє дерево) буде дерево tree(NewItem, empty, empty)".

Знак оклику (! – відсікання) означає, що, якщо речення успішно виконалось, інші виконувати не потрібно.

Друге та третє речення здійснюють вставку в непорожні дерева:

```
insert (NewItem, tree (Element, Left, Right), tree
(Element, NewLeft, Right) :-
    NewItem<Element,
    !,
    insert (NewItem, Left, NewLeft).
insert (NewItem, tree (Element, Left, Right),
tree (Element, Left, NewRight) :-
    insert (NewItem, Right, NewRight).
```

Якщо NewItem<Element, тоді вставляємо його у ліве піддерево, інакше – у праве.

2.6.6. Сортування на основі бінарного дерева

Після того, коли дерево побудовано, можна легко переставляти всі елементи в алфавітному порядку. Алгоритмом для цього знову стає пошук у глибину:

Якщо дерево порожнє, тоді нічого не робимо.

У іншому разі слід переставити всі елементи лівого піддерева, потім поточний елемент, далі всі елементи правого піддерева.

```
retrieve_all (empty).                % Нічого не робити
retrieve_all (tree (Item, Left, Right)) :-
    retrieve_all (Left),
    do_something_to (Item),
    retrieve_all (Right).
```

Для N вузлів час виконання алгоритму пропорційний $N \log N$. Це найшвидший алгоритм сортування на сьогоднішній день.

2.6.7. Списки

Список – це об'єкт, який містить скінчене число інших об'єктів. Наведемо декілька прикладів списку на мові Prolog:

```
[1, 2, 3, 4]
[[george, kate], [allen, amy], [don, pat]]
```

```
[tom, dick, harry, fred]
```

```
[] – порожній список.
```

Список є рекурсивним складеним об'єктом. Він складається з двох частин – *голови* та *хвоста*. Голова – це перший елемент списку, а *хвіст* – це всі інші елементи (без першого). Перший елемент списку відокремлюється від його хвоста оператором " | ". Голова списку – це завжди елемент, а хвіст – завжди список. За допомогою оператора " | " та процедури уніфікації список можна поділити на компоненти.

Якщо список [tom, dick, harry, fred] відповідає шаблону [X|Y], тоді X = tom та Y = [dick, harry, fred].

Якщо список [tom, dick, harry, fred] відповідає шаблону [X,Y|Z], тоді X = tom, Y = dick та Z = [harry, fred].

Якщо список [tom, dick, harry, fred] відповідає шаблону [X,Y,Z|W], тоді X = tom, Y = dick, Z = harry, W = [fred].

Якщо список [tom, dick, harry, fred] відповідає шаблону [W,X,Y,Z|V], тоді W = tom, X = dick, Y = harry, Z = fred, V = [].

Окрім розбиття списку на окремі частини, уніфікацію можна використовувати для його побудови. Наприклад, якщо X = tom, Y = [dick] та L уніфікуються з [X|Y], тоді змінна L буде пов'язана зі списком [tom, dick].

Prolog потребує, щоб всі елементи списку належали до одного домену. Декларація домену для елементів має бути такого вигляду:

```
domains
    elementlist = elements*
    elements = ...
```

Тут elements має єдиний тип (наприклад: integer, real, symbol) або є набором відмінних один від одного елементів, відмічених різними функторами. Наприклад, щоб оголосити список, який складається з цілих, дійсних та ідентифікаторів, запишемо

```
domains
    elementlist = elements*
    elements = i (integer); r (real); s (symbol) % де i, r, s – функтори
```

Інший приклад свідчить, як роздрукувати список цілих чисел.

Приклад 2.11.

```
domains
```

```
list = integer* /* Або довільний тип, який ви бажаєте */
```

```

predicates
  write_a_list (list)
clauses
  write_a_list ([]). /*Якщо список порожній – нічого не робити. */
  write_a_list ([H|T]):- /* Присвоїти H – голова, T – хвіст, потім... */
    write (H),nl,
    write_a_list (T).
goal
  write_a_list ([1,2,3]).

```

У разі першого перегляду цільове твердження `write_a_list ([1,2,3])` задовольняє другому реченню: $H=1$ та $T=[2,3]$. Комп'ютер надрукує 1 та викличе рекурсивно `write_a_list([2,3])`. Цей рекурсивний виклик задовольняє другому реченню. На цей раз $H=2$ та $T=[3]$, так що комп'ютер надрукує 2 та знов рекурсивно викличе `write_a_list([3])`. Унаслідок отримаємо: $H=3$ та $T=[]$. Програма друкує 3 та робить рекурсивний виклик: `write_a_list([])`. Цьому цільовому твердженню підходить перше речення, після якого програма закінчує свою роботу.

Розглянемо рекурсивну програму для підрахунку кількості елементів у списку.

Приклад. 2.12.

```

domains
  list = integer* % або довільний інший тип
predicates
  length_of (list, integer)
clauses
  length_of ([], 0).
  length_of ([_|T], L):-
    length_of (T, TailLength),
    L = TailLength + 1.
goal
  length_of ([1,2,3], L).

```

Ілюстрацію всіх викликів зображено:

```

length_of ([1, 2, 3], L1).
length_of ([2, 3], L2).
length_of ([3], L3).
length_of ([], 0).
L3 = 0+1 = 1.
L2 = L3+1 = 2.
L1 = L2+1 = 3.

```

Іноді необхідно перетворити один список в інший. Наприклад, додати до кожного елемента 1.

Приклад 2.13.

```

domains
  list = integer*
predicates
  add1 (list, list)
clauses
  add1 ([], []). /* гранична умова */
  add1 ([Head|Tail], [Head1|Tail1]):- /* відділити голову списку */
    Head1 = Head+1, /* додати 1 до першого елемента */
    add1 (Tail, Tail1). /* викликати елемент з решти списку */
goal
  add1 ([1,2,3,4], NewList).

```

На звичайну мову це можна перекласти так:

щоб додати 1 до всіх елементів порожнього списку, потрібно створити інший порожній список.

Щоб додати 1 до всіх елементів довільного непорожнього списку, потрібно додати 1 до голови та зробити отриманий елемент головою результуючого списку, потім додати 1 до кожного елемента хвоста списку та зробити його хвостом результату.

У цьому прикладі `add1` є хвостовою рекурсією. Запрограмовано виконання таких дій:

1. Зв'язати голову та хвіст вихідного списку з `Head` та `Tail`.
2. Зв'язати голову та хвіст результату з `Head1` та `Tail1` (`Head1` та `Tail1` поки не визначені).
3. Додати 1 до `Head` та присвоїти результат `Head1`.
4. Рекурсивно додати 1 до всіх елементів `Tail` та присвоїти результат `Tail1`.

Коли все буде завершено, `Head1` та `Tail1` стануть головою та хвостом результату. Рекурсивний виклик буде останнім кроком.

Не завжди у списку потрібно замінювати кожен елемент. Наступний приклад створює новий список з вихідного, відкидаючи від'ємні елементи.

Приклад 2.14.

```

domains
  list = integer*

```

```

predicates
  discard_negatives (list, list)
clauses
  discard_negatives ([], []).
  discard_negatives ([H|T], ProcessedTail):-
    H < 0,          /* Якщо H від'ємне, то
пропустити*/
    !,
    discard_negatives (T, ProcessedTail).
  discard_negatives ([H|T], [H|ProcessedTail]):-
    discard_negatives (T, ProcessedTail).
goal
  discard_negatives ([2, -45, 3, 468], X).

```

Результатом виконання програми стане $X = [2, 3, 468]$.

Нижче наведено предикат, який копіює елементи списку, заставляючи кожен елемент з'являтися двічі:

```

doubletalk ([], []).
doubletalk ([H|T], [H,H|DoubledTail]):-
  doubletalk (T, DoubledTail).

```

Найпростіший спосіб додати елемент X у список L – це вставити його на початок, для цього достатньо записати:

```
add (X, L, [X|L]).
```

Вилучення елемента X зі списку L можна запрограмувати у вигляді відношення

```
delete (X, L, L1).
```

де $L1$ збігається зі списком L , з якого вилучений елемент X . Для предиката $delete (X, L, L1)$ маємо два випадки:

(1) Якщо X є головою списку, тоді результатом вилучення буде хвіст цього списку.

(2) Якщо X знаходиться у хвості списку тоді його потрібно вилучити звідти.

```

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

```

Предикат $delete$ можна використовувати і у зворотному напрямі для того, щоб додати елементи в список у довільні місця. Наприклад, для цілі

```
delete (4, List, [1, 2, 3]).
```

Отримаємо

```

List = [4, 1, 2, 3]
List = [1, 4, 2, 3]
List = [1, 2, 4, 3]
List = [1, 2, 3, 4].

```

Операцію внесення елемента X у довільне місце списку можна задати у вигляді:

```
insert (X, L, L1) :- delete (X, L1, L).
```

Іноді буває корисно побудувати всі перестановки списку. Для цього опишемо предикат, який визначає всі перестановки списку $permutation$ з двома аргументами. Аргументи – це два списки, один з яких є перестановкою іншого. Можливі такі випадки:

(1) Якщо перший список порожній, тоді і другий список має бути порожнім.

(2) Якщо перший список не порожній, тоді він має вигляд $[X|H]$, а перестановку цього списку можна отримати наступним чином: отримати список $H1$ як перестановку H , а потім внести X у довільну позицію $H1$.

```

permutation ([], []).
permutation ([X|H], P):- permutation (H, H1), add (X, H1, P).

```

Для цілі

```
permutation ([1, 2, 3], NewList).
```

Отримаємо:

```

NewList = [1, 2, 3]
NewList = [2, 1, 3]
NewList = [2, 3, 1]
NewList = [1, 3, 2]
NewList = [3, 1, 2]
NewList = [3, 2, 1].

```

Особливості Prolog полягають у тому, що коли задають речення для предиката з одного погляду, вони будуть виконані по-іншому. Розглянемо деякі приклади.

Визначимо предикат, який дає змогу виявити, чи належить заданий елемент списку.

```

member (X, [X|_]).
member (X, [_|Tail]) :- member (X, Tail).

```

Якщо голова списку не збігається з X , тоді потрібно перевірити, чи можливо знайти X у хвості списку. На ці речення

можна дивитися з двох різних поглядів: декларативного та процедурного.

З декларативного погляду X належить списку, якщо голова збігається з X ; якщо ні, тоді X належить списку, якщо він належить хвосту.

З цього погляду можна задати цільове твердження для знаходження всіх членів списку: `member (X, [1, 2, 3, 4])`. У наслідок отримаємо:

$X=1$
 $X=2$
 $X=3$
 $X=4$
 4 Solutions.

З процедурного погляду ці два речення можна трактувати так:

Щоб знайти елемент списку, потрібно знайти його голову, інакше знайти елемент у хвості.

Це трактування співвідноситься з цільовим твердженням: `member (2, [1, 2, 3, 4])`. Відповіддю стане `yes`.

Створимо предикат з трьома аргументами для приєднання одного списку до іншого.

`append (List1, List2, List3).`

Він об'єднує `List1` та `List2` і створює `List3`. Скористаємося рекурсією з процедурного погляду. Якщо `List1` порожній, тоді результатом об'єднання стане `List2`:

`append ([], List2, List2).`

Якщо `List1` не порожній, тоді можна об'єднати `List1` та `List2` для формування `List3`, зробивши `List1` головою `List3`. (У наступному твердженні змінну H використовують як голову для `List1` та для `List3`). Хвіст `List3` – це `L3`, він складається з об'єднання залишку `List1` та всього `List2`:

`append ([H|L1], List2, [H|L3]) :- append (L1, List2, L3).`

Предикат `append` виконується як: поки `List1` не порожній, рекурсивне речення передає по одному елементу в `List3`. Коли `List1` стане порожнім, перше речення уніфікує `List2` з хвостом `List3`, унаслідок отримаємо список `[L3|L2]`.

Для цільового твердження

`append ([1, 2, 3], [5, 6], L)`

відповідь – це `L = [1, 2, 3, 5, 6]`.

Розглядаючи `append` з процедурного погляду, визначають відношення між трьома списками. Це відношення зберігається навіть тоді, коли `List1` та `List3` відомі, а `List2` – ні. Воно також справедливе, якщо відомий тільки `List3`. Наприклад, для того, щоб визначити, які з двох списків можна об'єднати для отримання відомого списку, можна задати цільове твердження вигляду:

`append (L1, L2, [1, 2, 4]).`

Отримаємо рішення:

`L1 = [], L2 = [1, 2, 4]`

`L1 = [1], L2 = [2, 4]`

`L1 = [1, 2], L2 = [4]`

`L1 = [1, 2, 4], L2 = []`

4 Solutions.

Можна так застосувати `append`, щоб визначити, який список можна приєднати до `[3, 4]` для отримання списку `[1, 2, 3, 4]`.

Задамо цільове твердження

`append (L1, [3, 4], [1, 2, 3, 4]).`

Результатом буде: `L1=[1, 2]`.

Предикат `append` визначив відношення між *вхідним набором* та *вихідним набором* так, що відношення можна застосовувати в обох випадках:

Який вихід відповідає цьому входу?

Який вхід відповідає цьому виходу?

Стан аргументів у разі виклику предиката названо *потокотом параметрів*. Аргумент, який присвоюється на момент виклику, названо *вхідним аргументом* та позначено буквою *i*; вільний аргумент – це *вихідний аргумент*, який позначено буквою *o*.

Не всі предикати можуть бути викликані з різним потоком параметрів. Якщо речення Prolog може бути викликано з різним потоком параметрів, його називають *оберненим реченням*.

2.7. Пошук всіх цілей відразу

Деколи у задачах потрібно окреслити пошук усіх альтернативних розв'язків для цільового твердження, що потрібні всі відразу як частина єдиної складної структури даних. Цього можна досягти за допомогою пошуку з поверненням. Вбудований предикат `findall` використовує цільові твердження як один зі своїх аргументів та збирає всі розв'язки у єдиний список. Предикат `findall` має три аргументи:

`VarName` (назва змінної) – визначає параметр, який необхідно зібрати у список;

`myPredicate` (мій предикат) – зазначає предикат, з якого потрібно зібрати значення;

`ListParam` (список параметрів) – містить список значень, зібраних методом пошуку з поверненням.

Наприклад, подаємо використання `findall` середнього віку групи людей.

Приклад 2.15.

```
domains
  name, address = string
  age = integer
  list = age*
predicates
  person (name, address, age)
  sumlist (list, age, integer)
run
clauses
  sumlist ([], 0, 0).
  sumlist ([H|T], Sum, N):-
    sumlist (T, S1, N1),
    Sum=H+S1, N=1+N1.
  person ("Sherlock Holmes", "22B Baker Street", 42).
  person ("Pete Spiers", "Apt. 22, 21st Street", 36).
  person ("Mary Darrow", "Suite 2, Omega Home", 51).
run:-
  findall (Age, person (_, _, Age), L),
  sumlist (L, Sum, N),
  Ave = Sum/N,
```

```
write ("Average=", Ave),nl.
```

```
goal
run.
```

Речення `findall` у цій програмі створює список `L`, в якому зібраний вік кожної людини з предиката `person`. Якщо б потрібно було зібрати список усіх людей, яким 42 роки, тоді потрібно виконати таке цільове твердження:

```
findall (Who, person (Who, _, 42), List).
```

Підціль потребує від програми, щоб вона містила оголошення домену для результуючого списку:

```
slist=string*.
```

2.8. Внутрішня база фактів

Внутрішня база фактів складається з фактів, які можна безпосередньо додавати або вилучати з програми під час її виконання. Предикати, які описують внутрішню базу фактів, оголошують у розділі `facts` програми та можуть бути використані так, як предикати, описані у розділі `predicates`. Відомі два обмеження на предикати, оголошені у розділі фактів:

1. Дозволено додавати у базу даних тільки факти, але не правила.
2. Факти бази мають бути без вільних змінних.

Можливе оголошення декількох розділів `facts`, але для цього потрібно явно вказати назву кожного розділу `facts`

```
facts -- mydatabase
  myFirstRelation (integer)
  mySecondRelation (real, string)
  myThirdRelation (string).
```

Опис розділу `facts` з назвою `mydatabase` створює базу даних з назвою `mydatabase`. За замовчуванням базі фактів присвоюється стандартна назва `dbasedom`. У двох різних розділах `facts` та у розділі `predicates` не можна застосовувати однакові назви предикатів.

Для додавання одного факту під час виконання програми можливі предикати:

```
asserta (<the fact>)
asserta (<the fact>, facts_sectionName)
```

```
assertz (<the fact>)
assertz (<the fact>, facts_sectionName)
assert (<the fact>)
assert (<the fact>, facts_sectionName).
```

Предикат `asserta` вставляє новий факт у базу даних фактів перед наявними фактами для цього предиката, `assertz` та `assert` вставляють факти після відомих фактів цього предиката. Необов'язковий параметр `facts_sectionName` свідчить, у яку базу даних потрібно додавати факт.

Перший предикат наступного прикладу вставляє факт про `Svitlana`, який описаний предикатом `person`, після всіх фактів `person`, що зберігаються на поточний момент у пам'яті. Другий – факт щодо `Dmytro` перед всіма наявними фактами предиката `person`. Третій – факт щодо `Ivan` після всіх інших фактів `likes` у базі даних фактів `likes Database`, а четвертий вставляє факт щодо `Maryna` у тій самій базі фактів перед всіма іншими фактами `likes`.

```
assertz (person ("Svitlana", "Odesa", 35)).
asserta (person ("Dmytro", "Lviv", 26)).
assertz (likes ("Ivan", "money"), likesDatabase).
asserta (likes ("Maryna", "hard work"), likesDatabase).
```

Після виклику цих предикатів база фактів набуде такого вигляду:

```
% Внутрішня база фактів – dbasedom
person ("Dmytro", "Lviv", 26).
% ... інші факти person ...
person ("Svitlana", "Odesa", 35).
% Внутрішня база фактів – likesDatabase
likes ("Maryna", "hard work").
% ... інші факти likes ...
likes ("Ivan", "money").
```

Вилучити факт із внутрішньої бази фактів під час виконання програми можна за допомогою предиката `retract`, який має такий формат:

```
retract (<the fact>)
retract (<the fact>, databaseName).
```

Предикат `retract` вилучає перший факт з вашої бази даних, який збігається з фактом `<the fact>`, зв'язуючи вільні змінні `<the fact>` під

час виконання програми. Припустимо, що у програмі є розділи `facts`.

Приклад 2.16.

```
facts
  person (string, string, integer)
facts – likesDatabase
  likes (string, string)
  dislikes (string, string)
clauses
  person ("Petro", "Rivne", 35).
  person ("Petro", "Lugansk", 37).
  person ("Dmytro", "Brody", 26).
  likes ("Ivan", "money").
  likes ("Oleksiy", "money").
  likes ("Olga", "chocolate").
  likes ("Ivan", "broccoli").
  dislikes ("Petro", "broccoli").
  dislikes ("Dmytro", "beer").
```

Маючи такі розділи, можна задати цілі:

```
retract (person ("Petro", _, _)),           %1
retract (likes (_, "broccoli")),             %2
retract (likes (_, "money"), likesDatabase), %3
retract (person ("Petro", _, _), likesDatabase). %4
```

Перша ціль вилучить перший факт `person` щодо `Petro` з бази фактів `dbasedom`. За допомогою другої цілі з бази фактів `likesDatabase` буде вилучений перший факт, який збігається з `likes (X, "broccoli")`. Третя ціль вилучає перший факт, що збігається з `likes (_, "money")` із `likesDatabase`. Четверта ціль видасть помилку, тому що немає факту `person` у базі даних `likesDatabase`.

Предикат `retractall` вилучає всі факти, що збігаються зі зразком `<the fact>` та має такий синтаксис:

```
retractall (<the fact>)
retractall (<the fact>, databaseName).
```

З `retractall` вихідні значення отримати не можна. Чергова ціль вилучає всі факти про чоловіків з бази фактів:

```
retractall (person (_, _, male)).
```

Для вилучення всіх фактів з бази `mydatabase` можна задати ціль `retractall (_, mydatabase).`

Предикат `save` зберігає факти з бази (розділ `facts`) у файл:

```
save (fileName)
save (fileName, databaseName).
```

Під час виклику предиката `save` тільки з одним аргументом (без назви бази фактів), у файлі `fileName` будуть збережені факти з бази даних `dbasedom`, яку використовують за замовчуванням. У разі виклику предиката `save` з двома аргументами (назва файлу та назва бази фактів), у файлі будуть збережені факти з розділу `facts` бази фактів із назвою `databaseName`.

Для зчитування фактів з файлу є предикат `consult`, який вставляє факти в кінець (у відповідну базу)

```
consult (fileName)
consult (filename, databaseName)
```

Предикат `consult` може зчитувати файли тільки у тому форматі, який створює `save`. У файлах не мають бути:

- символ верхнього регістру, за винятком тих, які містяться всередині рядків у подвійних лапках;
- пробіли, за винятком тих, які містяться всередині рядків у подвійних лапках;
- коментарі;
- порожні рядки;
- ідентифікатори (`symbol`) без подвійних лапок.

2.9. Зовнішні бази даних

Система внутрішніх баз даних Visual Prolog, у якій використано предикати `asserta`, `assertz`, `retract` та `retractall`, є зручною та простою. Однак вона поступається у швидкості роботи з великими базами даних, з цих міркувань була створена система **зовнішніх баз даних** за допомогою якої можна, наприклад, створити:

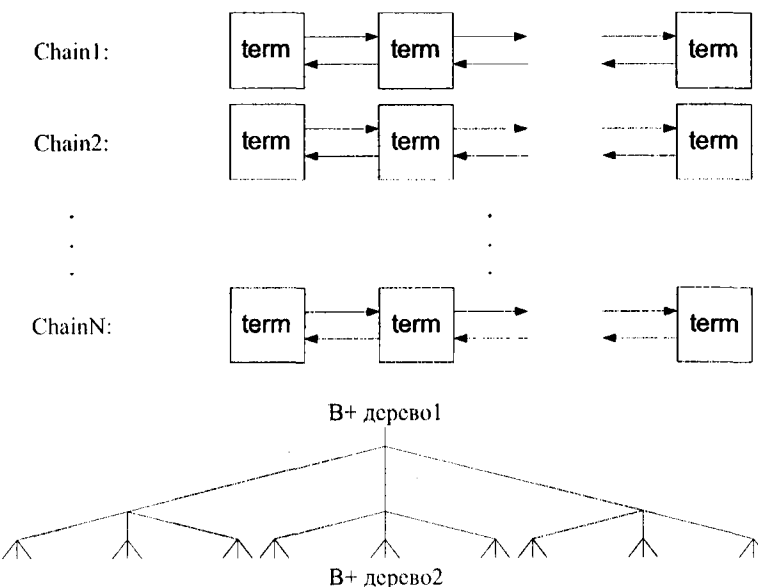
- систему керування запасами з великою кількістю записів;
- експертну систему з багатьма відношеннями та невеликою кількістю записів складної структури;
- облікову систему для запам'ятовування великих текстів у базі даних;

- базу користувача, у якій дані зв'язані нереляційним способом.

Предикати зовнішніх баз даних забезпечують такі можливості:

- ефективне оброблення великих об'ємів даних на диску;
- можливість розмістити базу даних як у файлі, так і в пам'яті;
- багатокористувацький доступ;
- гнучкість у роботі з даними;
- здатність зберегти та завантажити зовнішні бази даних у двійковому вигляді.

Зовнішня база даних складається з двох компонентів: одиниць даних, тобто термів Visual Prolog, згрупованих у ланцюжки, та відповідних *B+* дерев, які використовують для швидкого доступу до даних. Дані запам'ятовуються у ланцюжках (*chain*), *B+* дерева відіграють важливу роль при сортуванні бази та пошуку конкретного елемента. Структуру зовнішньої бази даних зображено на рис. 2.10.



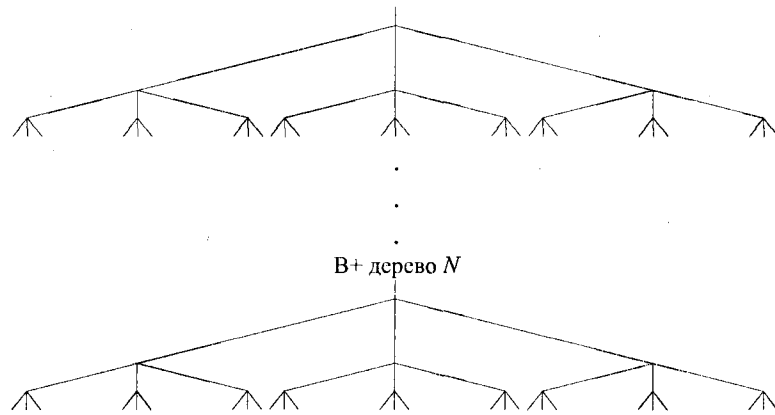


Рис. 2.10. Структура зовнішньої бази даних

Назви всіх стандартних предикатів керування базами даних підпорядковані таким узгодженням:

- перша частина імені (db_, chain_, term_ і т.д.) ідентифікує вхідні дані;
- друга частина імені (flush, btrees, delete і т.д.) вказує на дію, яка виконується над вхідними даними.

Наприклад, за допомогою db_delete – видаляють базу даних, шляхом chain_delete – ланцюг, а term_delete – дає змогу видалити терм.

З'являється можливість підтримувати відкритими декілька баз даних одночасно у пам'яті та на диску. Для вибору однієї з декількох відкритих баз даних при кожному виклику стандартних предикатів використовують *селектори*, які визначаються доменом db_selector. Він працює аналогічно домєну file в системі файлів. Наприклад, наступні оголошення декларують customers та parts селекторами зовнішніх баз даних:

domains

db_selector = customers; parts.

2.9.1. Ланцюгова структура баз даних

Зовнішня база даних є набором термів Visual Prolog. Прикладами термів можуть бути integer, real, string, symbol, величини та складені об'єкти, які в базі даних запам'ятовуються як ланцюжки (рис. 2.10). *Ланцюжок* (chain) може містити довільну кількість термів, а база даних – довільну кількість ланцюжків. Кожен ланцюжок ідентифікується іменем. Відношення та таблиці бази даних моделюють за допомогою ланцюжків термів. Наприклад, нехай ми маємо постачальника, покупця та базу товарів і хочемо помістити цю інформацію в одну базу з трьома відношеннями: одне для покупців, одне для постачальників, а одне для товарів. Це можливо зробити, якщо помістити всіх покупців у ланцюжок "покупці", всіх постачальників – у ланцюжок "постачальники", всі товари у ланцюжок "товари". Інший приклад засвідчує створення двох ланцюжкових баз даних: dba1 та dba2. Всі дані про покупців зберігаються в dba1, а всі дані про товари – в dba2.

Приклад 2.17.

domains

db_selector = dba1 ; dba2

customers = customer(customer_name, address)

parts = part(part_name, ID, customer_name)

customer_name, part_name = symbol

ID = integer

address = string

predicates

access

clauses

access:-

chain_terms(dba1,"chain1",customers,customer(Name,ADDR),_),

chain_terms(dba2,"chain2",parts,part(Part,Id,Name),_),

write("send ",Part," part num ",Id," to ",Addr),nl,fail.

access.

goal

% Створення баз даних dba1 та dba2

db_create(dba1,"dd1",in_memory),

db_create(dba2,"dd1.bin",in_file),

% Додавання фактів у ланцюжок покупців chain1 в dba1

```
chain_insertz(dba1,"chain1",customers,
customer("Joe Fraser","123 West Side"),_),
chain_insertz(dba1,"chain1",customers,
customer("John Smith","31 East Side"),_),
chain_insertz(dba1,"chain1",customers,
customer("Diver Dan","1 Water Way"),_),
chain_insertz(dba1,"chain1",customers,
customer("Dave Devine","123 Heaven Street"),_).
```

% Додавання фактів у ланцюжок chain2 в dba2

```
chain_insertz(dba2,"chain2",parts,
part("wrench",231,"John Smith"),_),
chain_insertz(dba2,"chain2",parts,part("knife",331,"Diver Dan"),_),
access,
db_close(dba1),
db_close(dba2),
db_delete("dd1",in_memory),
db_delete("dd1.bin",in_file).
```

Програма створює бази даних dba1 (в пам'яті) та dba2 (у файлі на диску). Потім у бази включають факти у два ланцюжки: chain1 та chain2. Після цього програма здійснює пошук покупця та товару, який замовив покупець. Знайшовши їх, повертає адресу, за якою необхідно доставити товар. Наприкінці бази даних закриваються та звільняється пам'ять.

Бази даних використовують шість стандартних доменів наведених у таблиці 2.2.

Таблиця 2.2.

Стандартні домени зовнішніх баз даних

Домен	Призначення для використання
db_selector	Домен для оголошення селекторів баз даних
bt_selector	Домен для оголошення селекторів B+ дерев
place	Домен для оголошення місця знаходження бази даних (у пам'яті або на диску)
accessmode	Домен, який вирішує, як використовувати файл
denymode	Домен, який визначає, як решта користувачів будуть відкривати файл
ref	Посилання на розміщення терму у ланцюжку

2.9.2. Предикати для роботи з базами даних

При створенні зовнішньої бази даних, або відкриванні наявної, можна помістити її у файл або в оперативну пам'ять, залежно від значення аргументу place при зверненні до db_create та db_open. Після завершення роботи з зовнішньою базою даних вона має бути закрита за допомогою виклику db_close. Якщо база даних розміщена у пам'яті, то її закриття за допомогою db_close не спричинить видалення її з пам'яті. Для звільнення пам'яті потрібно явно звернутися до предиката db_delete. За допомогою предиката db_open можна відкривати закриту базу даних. Розглянемо основні предикати для роботи з базами даних.

db_create (Dbase, Name, Place) – дає змогу створити нову базу даних. Якщо базу даних створюють на диску, тоді Name вказує на ім'я файла; якщо її створюються в оперативній пам'яті, тоді використовують як параметр для db_close та db_open. Dbase та Name визначають відповідно внутрішнє та зовнішнє ім'я бази даних. Місцезнаходження бази даних задають змінною Place, яка може набувати одне з трьох значень, описаних у таблиці 2.3.

Таблиця 2.3.

Значення змінної Place

Значення змінної Place	Опис
in_file	Для економії оперативної пам'яті базу даних розміщують на диску
in_memory	Для досягнення високої швидкості доступу базу даних розміщують в оперативній пам'яті
in_ems	Якщо встановлена карта розширення пам'яті, тоді базу даних розміщують в ЕМС розширеної пам'яті (тільки для DOS, на інших платформах має аналогічний ефект з in_memory)

db_open (Dbase, Name, Place) – дає можливість відкрити створену базу даних, ідентифіковану параметрами Name та Place. Якщо Place набуває значення in_memory, тоді Name відповідає символічному імені файла, а якщо Place набуває значення in_file, тоді Name відповідає імені файла, прийнятого в DOS.

db_copy (Dbase, Name, Place) – копіює базу даних незалежно від того, де вона знаходиться: у пам'яті чи на диску. Може бути використаний у декількох випадках, наприклад:

- для завантаження бази даних в оперативну пам'ять і для збереження її у файлі після оброблення;
- для копіювання бази даних середнього об'єму з диска у пам'ять (для прискорення доступу);
- для стиснення бази даних, яка містить багато порожніх записів, оскільки при копіюванні порожні записи вилучаються.

db_openinvalid (Dbase, Name, Place) – дає змогу відкрити базу даних, яка раніше була оголошена ушкодженою. Якщо в процесі оновлення бази було відключено живлення комп'ютера, вміст бази може бути втрачений. Базу даних вважають "ушкодженою" після оброблення довільного предиката, який змінює вміст бази. Це предикати: **chain_inserta**, **chain_insertz**, **chain_insertafter**, **term_replace**, **term_delete**, **chain_delete**, **bt_create**, **key_insert** та **key_delete**. Базу даних оголошують ушкодженою і після того, коли її закриває предикат **db_close**, а також після звернення до **db_flush** для звільнення буферів.

db_flush (Dbase) – звільняє буфери та записує їхній вміст на місце призначення в базі даних. Після оновлення базу даних вважають "ушкодженою" та залишається такою до запису за допомогою **db_flush** або до закриття.

db_close (Dbase) – закриває відкриту базу даних. Якщо база даних знаходиться на диску, тоді відповідний файл буде закритий. Разом з тим база даних не буде видалена, навіть якщо вона розміщена в оперативній пам'яті. Для видалення закритої бази даних використовують предикат **db_delete**.

db_delete (Name, Place) – звільняє пам'ять, якщо база даних розміщена у пам'яті, вилучає файл з диска, якщо база даних розміщена на диску. Якщо під час оброблення **db_delete** база даних Name не знаходиться на місці Place, виникає помилка.

db_btrees (Dbase, BtreeName) – у процесі пошуку з поверненням послідовно зв'язує BtreeName з іменем кожного B+ дерева в базі даних Dbase. Імена ланцюжків B+ дерев присвоюють у порядку сортування.

db_chains(Dbase,ChainName) – у процесі пошуку з поверненням по черговому зв'язує змінну ChainName з іменем кожного ланцюжка бази даних Dbase. Імена ланцюжків присвоюють у порядку сортування.

db_statistics(Dbase,NoOfTerms,MemSize,DbaSize,FreeSize) – вихідними даними предиката є статистичні відомості про базу даних Dbase. Аргументи предиката описані у таблиці 2.4.

Таблиця 2.4.

Аргументи предиката db_statistics

Аргумент	Опис
1	2
NoOfTerms	Зв'язується з загальним числом термів у базі даних
MemSize	Зв'язується з розміром (у байтах) внутрішньої таблиці для бази даних, яка зберігається в пам'яті
DbaSize	Зв'язується з загальним числом байтів, які зайняті термами та визначеннями бази даних Dbase. Якщо Dbase записана на диску та DbaSize набуває значення, що набагато менше ніж розмір файла, тоді файл може бути стиснутий з використанням db_copy
FreeSize	Зв'язується з величиною вільної пам'яті, яка залежить від того, де Dbase знаходиться у певний момент. Якщо в оперативній пам'яті, тоді FreeSize зв'язується з кількістю байтів вільної пам'яті; якщо Dbase розміщена у файлі, FreeSize зв'язується з кількістю вільних байтів на диску, який містить файл

Для включення термів у ланцюжок зовнішньої бази даних слід використовувати предикати **chain_inserta**, **chain_insertz** або **chain_insertafter**. Можна послідовно зв'язувати терми у ланцюжку та їхні ланцюжки з аргументами **chain_terms**. Предикат **chain_delete** видаляє ланцюжок термів з бази даних. Чотири стандартних предикати повертають вказівники бази даних – **chain_firs**, **chain_last**, **chain_next**, **chain_prev**.

chain_inserta(Dbase,Chain,Domain,Term,Ref),

chain_insertz(Dbase,Chain,Domain,Term,Ref) – ці предикати відповідають **asserta** та **assertz**. Водночас **chain_inserta** включає Term на початок ланцюжка, а **chain_insertz** розміщує Term у кінець лан-

цюжка. `Dbase` – це `db_selector` бази даних; `Domain` – домен змінної `Term`; `Ref` – вказівник на відповідний терм.

`chain_insertafter(Dbase, ChainName, Domain, Ref, Term, NewRef)` – розміщує `Term` після означеного терму вказівником `Ref`, причому `NewRef` зв'язується з вказівником, який відповідає терму `Term` після вставки.

`chain_terms(Dbase, Chain, Domain, Term, Ref)` – у процесі пошуку з поверненням по черговому зв'язує `Term` та `Ref` з кожним термом та відповідним йому вказівником у ланцюжку `Chain`.

`chain_delete(Dbase, Chain)` – вилучає визначений ланцюжок з бази даних.

`chain_first(Dbase, Chain),`

`chain_last(Dbase, Chain)` – ці предикати повертають вказівники першого та останнього елементів ланцюжка відповідно.

`chain_next(Dbase, Ref, NextRef),`

`chain_prev(Dbase, Ref, PrevRef)` – повертають вказівники наступного та попереднього термів відповідно до поточного терму у ланцюжку.

Наявні три предикати для роботи з базами даних, які обробляють терми: `term_replace`, `term_delete` та `ref_term`. При зверненні до будь-якого з цих предикатів необхідно оголосити домен терму як аргумент. Зручно оголосити всі терми бази даних як альтернативу одного домену, наприклад:

`domains`

```
terms_for_my_stock_control_database =
    = customer(Customer, Name, ZipCode, Address);
    supplier(SupplierNo, Name, Address);
    parts(PartNo, Description, Price, SupplierNo).
```

Немає обмежень на змішування доменів у зовнішній базі даних. Один ланцюжок може містити текстові рядки, інший – цілі числа, а третій – складені об'єкти і т. д.

`term_replace(Dbase, Domain, Ref, Term)` – замінює терм (з вказівником `Ref`) на новий терм з ім'ям `Term`.

`term_delete(Dbase, Chain, Ref)` – видаляє терм з вказівником `Ref`. Пам'ять, зайнята термом, буде звільнена.

`ref_term(Dbase, Domain, Ref, Term)` – зв'язує `Term` з термом, який розміщений за вказівником `Ref`.

Наведемо приклад програми, яка використовує майже всі розглянуті предикати. Працюючи з оперативною пам'яттю, програма виконує такі операції:

1. Записує 100 термів у базу даних.
2. Зчитує їх.
3. Замінює кожен другий терм.
4. Подвоює кількість термів.
5. Вилучає кожен другий терм.
6. Переглядає терми за допомогою предиката `ref_term`.
7. Обчислює розмір бази даних.

Далі програма копіює базу даних у файл та виконує ту ж саму послідовність дій вже з базою даних на диску. Після завершення підраховують (в сотих долях секунди) загальний час, який необхідний для виконання цих дій.

Приклад 2.18.

`domains`

`my_dom = f(string)`

`db_selector = my_dba`

`predicates`

`write_dba(integer)`

`read_dba`

`rd(Ref)`

`count_dba(integer)`

`count(Ref, integer, integer)`

`replace_dba`

`replace(Ref)`

`double_dba`

`double(Ref)`

`half_dba`

`half(Ref)`

`mixture`

`clauses`

`write_dba(0):-!`

`write_dba(N):-`

```
    chain_inserta(my_dba, "my_chain", my_dom,
                  f("Prolog system"), _),
```

```
    chain_insertz(my_dba, "my_chain", my_dom,
                  f("Prolog Compiler"), _),
```

```

    N1=N-1,write_dba(N1).
read_dba:-
    db_chains(my_dba,Chain),
    chain_terms(my_dba,Chain,my_dom,Term,Ref),nl,
    write("Ref=",Ref),write(", Term=",Term),fail.
read_dba:-
    db_chains(my_dba,Chain),
    chain_first(my_dba,Chain, Ref),
    rd(Ref),fail.
read_dba.
rd(Ref):-
    ref_term(my_dba,my_dom,Ref,Term),nl,
    write(Term),fail.
rd(Ref):-
    chain_next(my_dba,Ref,Next),!,rd(Next).
rd(_).
replace_dba:-
    chain_first(my_dba,"my_chain",Ref),
    replace(Ref).
replace(Ref):-
    term_replace(my_dba,my_dom,Ref,f("Prolog Toolbox")),
    chain_next(my_dba,Ref,NN),
    chain_next(my_dba,NN,Next),!,
    replace(Next).
replace(_).
half_dba:-
    chain_last(my_dba,"my_chain",Ref),
    half(Ref).
half(Ref):-
    chain_prev(my_dba,Ref,PP),
    chain_prev(my_dba,PP,Prev),!,
    term_delete(my_dba,"my_chain",Ref),
    half(Prev).
half(_).
double_dba:-
    chain_first(my_dba,"my_chain",Ref),
    double(Ref).
double(Ref):-
    chain_next(my_dba,Ref,Next),!,
    chain_insertafter(my_dba,"my_chain",
my_dom,Ref,f("Programmers Guide"),_),

```

```

    double(Next).
double(_).
count_dba(N):-
    chain_first(my_dba,"my_chain",Ref),
    count(Ref,1,N).
count(Ref,N,N2):-
    chain_next(my_dba,Ref,Next),!,
    N1=N+1,
    count(Next,N1,N2).
count(_ ,N,N).
mixture:-
    nl,
    write("Replace every second term:"),
    replace_dba,nl,
    write("Double the number of terms:"),
    double_dba,nl,
    write("Erase every second term:"),
    half_dba,nl,
    write("Use ref_term for all terms:"),
    read_dba,
    count_dba(N),nl,
    write("There are now ",N," terms in the database"),
    db_statistics(my_dba,NoOfTerms,
MemSize,DbaSize,FreeSize),nl,
    writef("NoOfTerms=%, MemSize=%, DbaSize=%,
FreeSize=%, NoOfTerms, MemSize,DbaSize,FreeSize).

goal
nl,nl,nl,
write("\tTEST OF DATABASE SYSTEM\n\t*****\n\n"),
time(H1,M1,S1,D1),
db_create(my_dba,"dd.dat",in_memory),nl,nl,
write("Write some terms in the database:"),
write_dba(50),
read_dba,
mixture,nl,nl,
write("Copy to file"),
db_copy(my_dba,"dd.dat",in_file),
db_close(my_dba),
db_delete("dd.dat",in_memory),
db_open(my_dba,"dd.dat",in_file),
mixture,

```



```

db_close(my_dba),nl,nl,nl,
write("Open the database on file"),
db_open(my_dba,"dd.dat",in_file),
mixture,
db_close(my_dba),
time(H2,M2,S2,D2),
Time = (D2-D1)+100.0*((S2-S1)+60.0*((M2-M1)+
60.0*(H2-H1))),nl,nl,
write("Time = ",Time,"/100 Sec"),nl.

```

2.9.3. B+ дерева

B+ дерево – це структура даних, яку можна застосовувати для досить ефективного методу сортування великої кількості даних; B+ дерева дають можливість використовувати ефективний алгоритм пошуку та є аналогами до вказівників бази даних (B+ дерева іноді порівнюють з вказівниками).

У Visual Prolog B+ дерева знаходяться у зовнішній базі даних. Кожен вхід у B+ дерево – це пара величин: *ключовий рядок* та зв'язаний з ним *вказівник бази даних*. При створенні бази даних спочатку утворюють запис та визначають ключ для цього запису. Потім Visual Prolog додає цей ключ та вказівник, який відповідає запису, у B+ дерево. Під час пошуку у базі даних все, що необхідно зробити, – застосувати ключ для цього запису, а B+ дерево видасть відповідний вказівник. Використовуючи його, можна дістатися до запису в базі. В процесі створення B+ дерева елементи розміщують у порядку, визначеному ключами. Отже, можна легко отримати відсортований список записів. B+ дерево, подібне до бінарних дерев, за винятком того, що в B+ дереві у кожному вузлі запам'ятовується більше ніж один ключовий рядок. B+ дерева збалансовані: це означає, що шлях пошуку кожного ключа у "листях" дерева має одну й ту ж саму довжину. Внаслідок цього пошук кожного ключа (серед величезної кількості ключів) потребує лише декількох операцій доступу до диска, залежно від того, як багато ключів запам'ятовується у кожному вузлі.

Хоча B+ дерева розміщуються у зовнішніх базах даних, вони не потребують вказівників на терми в тій самій базі даних. З'яв-

ляється можливість мати базу даних, яка містить послідовність ланцюжків, та другу базу даних з B+ деревом, яке вказує на терми в цих ланцюжках.

У B+ деревах ключі згруповані у *сторінки*, причому кожна сторінка має однаковий розмір, і всі сторінки можуть містити одне й те саме число ключів. Це означає, що всі ключі B+ дерева мають бути однакового розміру. Розмір ключів визначають за аргументом *KeyLen*, який задають у момент створення дерева. При спробі внесення у B+ дерево рядка, довжина якого більша ніж *KeyLen*, Visual Prolog вкоротить його. Під час створення B+ дерева необхідно задати величину аргументу *Order*. Цей аргумент визначає, скільки ключів буде запам'ятовано у кожному вузлі дерева. Найліпшу величину аргументу вибирають методом спроб та помилок.

Створюючи B+ дерево, необхідно передбачити можливість використання ключів, які повторюються. Наприклад, якщо створюють B+ дерево для бази даних покупців, в якій ключем є прізвище покупця, тоді необхідно врахувати всіх покупців з прізвищем Олійник. У цьому разі допоможе застосування подвійних ключів у B+ дереві.

Розглянемо стандартні предикати для оброблення B+ дерев:

```

bt_create(Dbase,BtreeName,Btree_Sel,KeyLen,Order),
bt_create(Dbase,BtreeName,Btree_Sel,KeyLen,Order,Duplicates)

```

– створюють B+ дерево. Аргумент *BtreeName* визначає ім'я нового дерева. Це ім'я стає аргументом для *bt_open*. Аргументи *KeyLen* та *Order* задають у момент створення B+ дерева. Після цього їх змінювати не можна. Якщо викликається *bt_create/5* або *bt_create/6* з аргументом *Duplicates=1*, тоді в B+ дереві можна використовувати подвійні ключі. Якщо *bt_create/6* викликають з аргументом *Duplicates=0*, тоді у дерево не можна включати ключі, що дублюються.

bt_open(Dbase,BtreeName,Btree_Sel) – відкриває вже створене B+ дерево, яке ідентифікується ім'ям, заданим в *bt_create*. У момент відкриття та створення B+ дерева повертається значення селектора дерева *Btree_Sel*. Селектор B+ дерева належить до заздалегідь визначеного домену *bt_selector* та вказує на B+ дерево, коли система виконає операції пошуку або позиціонування. Зв'язок

між іменем дерева та його селектором подібний до зв'язку між фізичним іменем файла та відповідним йому символічним іменем.

`bt_close(Dbbase,Btree_Sel)` – використовують для закриття B+ дерева; звільняє внутрішні буфери з іменем `BtreeName`.

`bt_delete(Dbbase,BtreeName)` – застосовують для видалення B+ дерева.

`bt_copyselctor(Dbbase,OldBtree_sel,NewBtree_sel)` – створює новий вказівник на вже відкритий селектор B+ дерева. Новий вказівник вказує на те саме місце, що і наявний. Після створення обидва селектори можуть вільно переміщатися, не впливаючи один на одного.

`bt_statistics(Dbbase,Btree_sel,NumKeys,NumPages,Depth,KeyLen,Order,PgSize)` – вертає статистичну інформацію для B+ дерева, на яке вказує `Btree_Sel`. Аргументи предиката наведено у таблиці 2.5.

Таблиця 2.5.

Аргументи предиката `bt_statistics`

Аргумент	Опис
<code>Dbbase</code>	Селектор, який визначає базу даних
<code>Btree_Sel</code>	Селектор B+ дерева
<code>NumKeys</code>	Загальна кількість ключів у B+ дереві
<code>NumPages</code>	Загальна кількість сторінок
<code>Depth</code>	Глибина дерева
<code>KeyLen</code>	Довжина ключа
<code>Order</code>	Порядок B+ дерева
<code>PgSize</code>	Розмір сторінки (у байтах)

`key_insert(Dbbase,Btree_Sel,Key,Ref),`

`key_delete(Dbbase,Btree_Sel,Key,Ref)` – стандартні предикати, які використовують для оновлення B+ дерева. Задавши для предиката `key_delete` обидва аргументи (`Key` та `Ref`), можна видалити визначений елемент з подвійним ключем.

`key_first(Dbbase,Btree_Sel,Ref),`

`key_last(Dbbase,Btree_Sel,Ref),`

`key_search(Dbbase,Btree_Sel,Key,Ref)` – кожне B+ дерево підтримує внутрішні вказівники на свої вузли. Предикати `key_first` та `key_last` дають змогу встановити вказівник на перший та останній

вузли у B+ дереві. За допомогою предиката `key_search` встановлюють вказівник на заданий ключ; процедура завершується успішно, якщо ключ знайдений, інакше внутрішній вказівник B+ дерева встановлюють на наступний ключ після того, де мав би бути розміщений ключ `Key`. Можна використовувати предикат `key_current`, щоб отримати ключ та вказівник бази даних на цей ключ. Якщо потрібно вказати точну позицію у B+ дереві з подвійними ключами, тоді необхідно `Ref` задати як вхідний параметр.

`key_next(Dbbase,Btree_Sel,NextRef),`

`key_prev(Dbbase,Btree_Sel,PrevRef)` – використовують для переміщення вказівника B+ дерева вперед та назад. Якщо вказівник знаходиться на одному з кінців B+ дерева, спроба пересунути його далі завершується невдачею, й вказівник B+ дерева буде працювати так, ніби він був пересунутий на одну позицію поза межі дерева.

`key_current(Dbbase,Btree_Sel,Key,Ref)` – повертає ключ з вказівником бази даних, що відповідає поточному положенню вказівника у B+ дереві. Предикат `key_current` завершується невдачею після звернення до предикатів `bt_open`, `bt_create`, `key_insert` або `key_delete`, а також якщо вказівник знаходиться перед першим (використовують `key_prev`) або після останнього ключа (`key_next`).

Розглянемо приклад доступу до бази даних через B+ дерева. Наведена нижче програма дає можливість обробляти декілька текстових файлів в одній базі даних. Можна вибирати та редагувати тексти. Відповідне B+ дерево застосовують для швидкого доступу до файлів та сортування їхніх назв.

Приклад 2.19.

`domains`

`db_selector = dba`

`predicates`

`% Список всіх ключів`

`list_keys(db_selector,bt_selector)`

`clauses`

`list_keys(dba,Bt_selector):-`

`key_current(dba,Bt_selector,Key,_),`

`write(Key,' '),`

`fail.`

```

list_keys(dba,Bt_selector):-
    key_next(dba,Bt_selector,_),!,
    list_keys(dba,Bt_selector).
list_keys(_,_).
predicates,
open_dbase(bt_selector),
main(db_selector,bt_selector),
ed(db_selector,bt_selector,string),
ed1(db_selector,bt_selector,string),
clauses
% Виконання циклу до натискання довільної клавіші
main(dba,Bt_select):-
    write("File Name: "),
    readln(Name),
    ed(dba,Bt_select,Name),!,
    main(dba,Bt_select).
main(_,_).
% Предикат ed засвідчує, що під час редагування немає невдач.
ed(dba,Bt_select,Name):-
    ed1(dba,Bt_select,Name),!.
ed(_,_,_).
%*****
% Є три можливості:
%% а) Якщо ім'я є порожній рядок – переглянути всі імена;
% б) Ім'я вже відомо – змінити зміст файлу;
% в) Нове ім'я – створити новий файл;
%*****/
ed1(dba,Bt_select,""):-!,
    key_first(dba,Bt_select,_),
    list_keys(dba,Bt_select),
    nl.
ed1(dba,Bt_select,Name):-
    key_search(dba,Bt_select,Name,Ref),!,
    ref_term(dba,string,Ref,Str),
    edit(Str,Str1,"Edit old",NAME,"",0,"PROLOG.HLP",RET),
    clearwindow,
    Str><Str1, RET=0,
    term_replace(dba, string, Ref, Str1).
ed1(dba,Bt_select,Name):-
    edit("",STR1,"Create New",
        NAME,"",0,"PROLOG.HLP",RET),

```

```

clearwindow,
""><Str1, RET=0,
chain_insertz(dba,file_chain,string,Str1,Ref),
key_insert(dba,Bt_select,Name,Ref).
open_dbase(INDEX):-
    existfile("dd1.dat"),!,
    db_open(dba,"dd1.dat",in_file),
    bt_open(dba,"ndx",INDEX).
open_dbase(INDEX):-
    db_create(dba,"dd1.dat",in_file),
    bt_create(dba,"ndx",INDEX,20,4).
goal
open_dbase(INDEX),
main(dba,INDEX),
bt_close(dba,INDEX),
db_close(dba).

```

2.10. Арифметичні операції

Використовуючи Prolog, можна виконувати всі чотири головні арифметичні операції (додавання, віднімання, ділення, множення) між цілими та дійсними числами. Шістнадцяткові та вісімкові числа починаються з "0x" або "0o" відповідно. Арифметичні операції обчислюють у такому порядку:

- якщо вираз містить підвирази у дужках, підвираз обчислюють першим;
- якщо вираз містить операції множення (*) або ділення (/, div або mod), ці операції виконують зліва направо;
- якщо вираз містить операції додавання (+) та віднімання (-), їх виконують так само: зліва направо.

Порядок операцій наведено у таблиці:

Таблиця 3.6.

Порядок виконання арифметичних операцій

Операція	Пріоритет
+, -	1
*, /, mod, div	2
-, + (унарні)	3

У програмі Prolog є повний набір вбудованих математичних функцій та предикатів, які використовують цілі та дійсні значення. Повний їхній список подано у таблиці 2.7:

Таблиця 2.7.

Арифметичні предикати та функції

Назва	Опис
$X \bmod Y$	Повертає залишок від ділення X на Y
$X \div Y$	Повертає частку від ділення X на Y
$\text{abs}(X)$	Повертає модуль свого аргументу
$\cos(X)$	Повертає косинус свого аргументу
$\sin(X)$	Повертає синус свого аргументу
$\tan(X)$	Повертає тангенс свого аргументу
$\arctan(X)$	Повертає арктангенс дійсного значення X
$\exp(X)$	Підносить e до степеня X
$\ln(X)$	Логарифм X при основі e
$\log(X)$	Логарифм X при основі 10
$\text{sqrt}(X)$	Корінь квадратний з X
$\text{random}(X)$	Присвоює X випадкове дійсне число; $0 \leq X < 1$
$\text{random}(X, Y)$	Присвоює Y випадкове дійсне число; $0 \leq Y < X$
$\text{round}(X)$	Заокруглює значення X . Результат дійсний
$\text{trunc}(X)$	Відкидає дробову частину. Результат дійсний
$\text{val}(\text{domain}, X)$	Вертає X перетворене до типу domain

Prolog дає змогу порівнювати арифметичні вирази так само, як символи, рядки та ідентифікатори, наприклад:

'a' < 'b' % Символи

"antony" > "antonia" % Рядки

P1 = peter, P2 = sally, P1 > P2 % Літерали (ідентифікатори)

У таблиці 2.8. наведено повний ряд відношень:

Таблиця 2.8.

Логічні відношення

Ідентифікатор	Відношення
<, >	Менше, більше
<=, >=	Менше або дорівнює, більше або дорівнює
=, <> або ><	Дорівнює або не дорівнює

Прикладом роботи з арифметичними операціями може стати програма для знаходження корнів квадратного рівняння $A \cdot X^2 + B \cdot X + C = 0$.

Приклад. 2.20.

```

predicates
  solve (real, real, real)
  reply (real, real, real)
  mysqrt (real, real, real)
clauses
  solve (A, B, C):-
    D=B*B-4*A*C,
    reply (A, B, D), nl.

  reply (_,_,D):-
    D < 0,
    write ("No solution"),!.

  reply (A, B, D):-
    D=0,
    X=-B/(2*A),write("x=", X),!.

  reply (A, B, D):-
    SqrtD=sqrt(D),
    X1=(-B+SqrtD)/(2*A),
    X2 = (-B - SqrtD)/(2*A),
    write ("x1 = ", X1," and x2 = ", X2).

goal
  solve (1.0, 2.0, 1.0).

```

Результатом виконання програми буде $x=-1$.

2.11. Уведення та виведення інформації

У програмі Prolog є три стандартні предикати для виведення інформації: write, nl, writef. Предикат write може бути викликаний з довільним числом аргументів:

write (Param1, Param2, Param3, ..., ParamN).

Ці аргументи можуть бути або константами зі стандартних доменів, або змінними. Якщо це змінні, то вони мають бути вхідними параметрами. Предикат nl (new line – новий рядок) завжди використовується разом з предикатом write. Він забезпечує перехід на новий рядок. Зворотний слеш “\” використовують у рядках як керуючий символ. Для того, щоб надрукувати безпосередньо символ “\”, потрібні два зворотні слеші один за одним. За зворотним сле-

шем може стояти довільний, утворений спеціальними символами для керування друком:

'n' – новий рядок та повернення каретки;

't' – табуляція;

'r' – повернення каретки.

Також за символом "\" може стояти до трьох десяткових цифр, які задають код ASCII.

Наведемо приклад, який видруковує не більше ніж по п'ять елементів списку в рядку.

Приклад 2.21.

```
domains
integerlist = integer*
predicates
writelist(integerlist)
write5(integerlist, integer)
clauses
writelist(NL):-
    nl,
    write5(NL, 0),nl.
write5(TL, 5):-!,
    nl,
    write5(TL, 0).
write5([H|T], N):-!,
    write(H, " "),
    N1=N+1,
    write5(T, N1).
write5([],_).
goal
writelist([2,4,6,8,10,12,14,16,18,20,22]).
```

Предикат `writef` дає змогу виконувати форматоване виведення, він має такий формат:

`writef(FormatString, Arg1, Arg2, ..., ArgN)`

Аргументи `Arg1, ..., ArgN` мають бути константами або зв'язаними змінними, що належать стандартним доменам. Складені домени форматувати не можна. Рядок форматування містить *звичайні символи* та *форматні специфікатори*; звичайні символи друкують без модифікації, а форматні специфікатори набувають такої

форми: `%-m.pf`. Символи специфікаторів формату, які слідує за знаком "%" є необов'язковими, їхні значення наведено у таблиці.

Таблиця 2.9.

Значення символів-специфікаторів формату

Символ	Значення
-	Вирівнювання поля по лівому краю (за замовченням – по правому)
m поле	Десяткове число, за яким визначають мінімальну довжину поля
p поле	Десяткове число, за яким описують або точність зображення з плаваючою крапкою, або максимальне число символів у рядку
f поле	Дас можливість описувати формати, відмінні від форматів, прийнятих за замовчуванням для об'єкта. Наприклад, поле f може описувати те, що ціле число має бути надруковане як беззнакове або шістнадцяткове

У таблиці наведено специфікатори формату поля f.

Таблиця 2.10.

Специфікатори формату поля f

Специфікатор формату	Опис
f	Дійсні разом з фіксованою крапкою (тип 123,4 або 0,004321)
e	Дійсні в експоненціальному описі (тип 1,234e2 або 4,321e-3)
g	Дійсні у форматі f або e (формат за замовченням)
d	Цілі, як знакові десяткові числа
D	Цілі, як знакові довгі десяткові числа
u	Цілі, як беззнакові десяткові числа
U	Цілі, як беззнакові довгі десяткові числа
o	Цілі, як вісімкові числа
O	Цілі, як вісімкові довгі числа
x	Цілі, як шістнадцяткові числа
X	Цілі, як шістнадцяткові довгі числа
c	Цілі, як символи (char)
s	Як рядки (symbols та string)
R	Як числа посилання у зовнішніх базах даних (тільки для доменів ref)
B	Як бінарні (тільки для домену binary)
P	Як предикатне значення

Приклади форматowanego виведення:

goal

```
A = "one",
B = 330.12,
C = 4.3333375,
D = "one two three",
writef ("A = '%-7' \nB = '%8.1e'\n", A, B),
writef ("A = '%1' \nB = '%8.4e'\n", A, B),nl,
writef ("C = '%-7.7g' \nD = '%7.7'\n", C, D),
writef ("C = '%-7.0f' \nD = '%0'\n", C, D),
writef ("char: %c, decimal: %d, octal: %o, hex:
%x", 'a', 'a', 'a', 'a').
```

У результаті отримаємо

```
A = 'one'
B = '3.3E+02'
A = 'one'
B = '3.3012E+02'
C = '4.333337'
D = 'one two'
C = '4'
D = 'one two three'
char: a, decimal: 97, octal: 141, hex: 61
```

Упрограмі Prolog охоплено декілька стандартних предикатів для читання з поточного пристрою введення:

`readln (Line)` – зчитує текстовий рядок у вільну змінну `Line`, для якої визначений домен типу `string`. Зчитує до 254 символів (плюс повернення каретки) з клавіатури та до 64 Кбайт з інших пристроїв. Якщо під час введення натиснути клавішу `<Esc>`, то `readln` буде невдалим;

`readint (X)` – зчитує ціле значення у вільну змінну `X`, для якої визначений домен цілого типу, поки не буде натиснута клавіша `<Enter>`. Якщо введення не відповідає синтаксису цілих, `readint` буде невдалим, а Prolog викликатиме механізм пошуку з поверненням. У разі натискання клавіші `<Esc>` під час введення з клавіатури `readint` також зазнає невдачі;

`readreal (X)` – зчитує дійсне значення у вільну змінну `X`, для якої визначений домен дійсного типу, поки не буде натиснута клавіша `<Enter>`. Якщо введення не відповідає звичайному син-

таксису дійсних чисел, то `readreal` зазнає невдачі. У разі натискання клавіші `<Esc>` під час введення з клавіатури `readreal` також зазнає невдачі;

`readchar (CharParam)` – зчитує один символ у вільну змінну `CharParam`, яка належить домену `char`. Якщо поточним пристроєм введення є клавіатура, `readchar` чекає, поки не буде натиснутий один символ, після чого повертає його. Внаслідок натискання клавіші `<Esc>` під час введення з клавіатури, `readchar` зазнає невдачі;

`readterm (DomainName, Term)` – зчитує рядок та перетворює його у терм вказаного домену. Якщо рядок не виглядає як об'єкт, сформований предикатом `write`, тоді `readterm` поверне помилку. Стандартний предикат `readtermerror` може бути використаний для отримання інформації про помилки, які відбулися у `readterm`;

`file_str (Filename, Text)` – зчитує текстовий файл у рядок або створює файл та записує у цей файл рядок. Якщо перед викликом `file_str` змінна `Text` вільна, `file_str` зчитує символи з файла `Filename`, поки не натрапить на символ кінця файла. Зміст файла `Filename` поміщається у змінну `Text`. Файл, який зчитується, не має перевищувати максимальної довжини рядка (64Кб). Якщо `Text` зв'язана змінна, тоді `file_str` створює файл з назвою `Filename`, в який записується вміст файлу, зв'язаного зі змінною `Text`. Причому, якщо файл з назвою `Filename` вже є, він буде перезаписаний.

Розглянемо приклад використання стандартних предикатів читання для роботи зі складними структурами даних.

Приклад 2.22.

```
domains
person = p (name, age, telno, job)
age = integer
telno, name, job = string
predicates
run - procedure ()
clauses
run:-
write ("Which name ? "), readln (Name),
write ("Job ? "), readln (Job),
write ("Age ? "), readint (Age),
```

```

write ("Telephone no ? "), readln (Telno),
P = p (Name, Age, Telno, Job),
write (P), nl,
write ("Is this compound object OK (y/n)"),
readchar (Ch), nl, Ch='y', !.
run :-
    nl, nl,
    write ("Alright, try again"),
    nl, nl, run.
goal
run.

```

Увівши відповіді на запити програми, результат виконання можна отримати у вигляді:

```

Which name ? Andry
Job ? Petrenko
Age ? 20
Telephone no ? 297-97-97
p ("Andry", 20, "297-97-97", "Petrenko")
Is this compound object OK (y/n).

```

2.12. Файлова система

У цьому розділі розглянемо файлову систему Visual Prolog та стандартні предикати для роботи з файлами. Visual Prolog дає змогу використовувати *current_read_device* (поточний пристрій читання) та *current_write_device* поточний пристрій запису. Зазвичай поточним пристроєм читання є клавіатура, а поточним пристроєм запису – екран дисплея. Пристрій поточного введення – виведення можна перевизначити під час виконання програми. Відкриваючи файл, Visual Prolog зв'язує символічне ім'я з дійсним іменем файла операційної системи та використовує символічне ім'я для направлення введення-виведення. Символічні імена файлів необхідно починати з маленької літери та оголошувати у домені file. Наприклад:

```
file = file1; source; auxiliary; somethingElse.
```

У будь-якій програмі дозволений тільки один домен file. Visual Prolog розпізнає п'ять вбудованих альтернатив file, наведених у таблиці.

Таблиця 3.11.

Вбудовані альтернативи домену file

Альтернатива	Опис
keyboard	Читання з клавіатури
screen	Запис на дисплей
stdin	Читання зі стандартного введення
stdout	Запис у стандартне виведення
stderr	Запис на стандартний пристрій для виведення помилок

Вбудовані альтернативи не мають траплятися в описі file. Розглянемо стандартні предикати для відкриття та закриття файлів.

openread (SymbolicFileName, OSFileName) – дає можливість відкривати файл OSFileName для читання та пов'язувати його з символічним іменем SymbolicFileName, оголошеним у домені file. Якщо файл не може бути відкритий, з'являється повідомлення про помилку.

openwrite (SymbolicFileName, OSFileName) – дає змогу відкривати файл OSFileName для запису. Якщо файл вже є на диску, його вмістиме витирають. У протилежному випадку створюють новий файл. Якщо файл не може бути створений, з'являється повідомлення про помилку.

openappend (SymbolicFileName, OSFileName) – дає можливість відкривати файл OSFileName для запису в кінець файла. Якщо файл не може бути відкритий для запису, з'являється повідомлення про помилку.

openmodify (SymbolicFileName, OSFileName) – дає змогу відкривати файл OSFileName і для запису, і для читання. Якщо файл вже створено, він не буде перезаписаний. Якщо система не може відкрити OSFileName, Visual Prolog повідомить про помилку. Для заповнення файла з довільним доступом предикат *openmodify* можна використовувати разом зі стандартним предикатом *filepos*.

filemode (SymbolicFileName, FileMode) – дає можливість відкривати файл у двійковому або у текстовому режимі. Якщо FileMode=0, файл SymbolicFileName встановлюють у двійковому режимі, якщо FileMode=1 – у текстовому. Для читання двійкового

файла можна використовувати тільки предикат `readchar` або предикати для доступу до двійкових файлів.

`closefile (SymbolicFileName)` – закриває вказаний файл.

`readdevice (SymbolicFileName)` – перевизначає поточний пристрій читання, якщо змінна `SymbolicFileName` визначена та файл відкритий для читання. Якщо `SymbolicFileName` є вільною змінною, тоді `readdevice` присвоїть їй ім'я поточного активного пристрою читання.

`writedevise (SymbolicFileName)` – перевизначає пристрій запису, якщо зазначений файл відкритий для запису. Якщо змінна `SymbolicFileName` є вільною, тоді `writedevise` присвоїть їй ім'я поточного активного пристрою запису.

Домен `file` містить три стандартні файлові потоки, опис яких наведено у таблиці 2.12.

Таблиця 2.12.

Файлові потоки

Файловий потік	Опис
<code>stdin</code>	Стандартне введення є файлом, доступним тільки для читання. За замовчуванням це клавіатура. <code>readdevice(stdin)</code> назначає пристроєм введення <code>stdin</code>
<code>stdout</code>	Стандартне виведення є файлом, доступним тільки для запису. За замовчуванням це екран терміналу. <code>writedevise(stdout)</code> назначає пристроєм виведення <code>stdout</code>
<code>stderr</code>	Стандартне виведення помилок є файлом, який доступний тільки для запису. За замовчуванням це екран терміналу. <code>writedevise (stderr)</code> назначає пристроєм для виведення повідомлень про помилки <code>stderr</code>

Розглянемо предикати для роботи з файлами:

`filepos (SymbolicFileName,FilePosition,Mode)` – керує позицією у файлі для читання та запису. Якщо `FilePosition` – зв'язана змінна, предикат може змінювати позицію читання та запису для файла з іменем `SymbolicFileName`. Якщо під час виклику `FilePosition` є вільною змінною, тоді `SymbolicFileName` повертає поточну позицію у файлі. Значення `SymbolicFileName` належить типу `long`. `Mode` є цілою величиною та вказує, як має інтерпретуватися значення `FilePosition` (0 – щодо початку файла, 1 – щодо поточної позиції, 2 – щодо кінця файла).

`eof(SymbolicFileName)` – перевіряє позицію у процесі читання на символ кінця файла.

Наведемо приклад програми, що перетворює один файл в інший з приведенням усіх символів до верхнього регістру.

Приклад 2.23

```
domains
    file = input; output
predicates
    convert_file
    repfile (FILE)
run
clauses
    convert_file :-
        repfile (input),
        readln (Ln),
        upper_lower (LnInUpper, Ln), %Переводить букви у великі
        write (LnInUpper), nl,
        fail.
    convert_file.
    repfile (_).
    repfile (F):-
        not (eof(F)),
        repfile (F).
run:-
    write ("Which file do you want convert ?"),
    readln (InputFileName),nl,
    write ("What is the name of the output file ?"),
    readln (OutputFileName),nl,
    openread (input, InputFileName),
    readdevice (input),
    openwrite (output, OutputFileName),
    writedevise (output),
    convert_file,
    closefile(input),
    closefile(output).
goal
run.
```

`flush (SymbolicFileName)` – дає змогу записувати вміст внутрішнього буфера у файл.

`existfile (OSFileName)` – дає можливість перевіряти, чи існує заданий файл `OSFileName`. Для перевірки того, чи є файл на диску, можна скористатися такою процедурою:

```
open (File, Name):-
    existfile (Name),!,
    openread (File, Name).
open( _, Name):-
    write ("Error: the file", Name, "is not found").
```

`searchfile (PathList, Name, FoundName)` – використовують для знаходження файла у списку шляхів. `PathList` – це рядок, який містить один або більше шляхів, розділених крапкою з комою, `Name` – ім'я шуканого файла. Якщо файл знайдено, `FoundName` буде зв'язана з повним його іменем.

`deletefile (OSFileName)` – знищує файл `OSFileName`.

`renamefile (OldOSFileName, NewOSFileName)` – перєіменовує файл з іменем `OldOSFileName` в `NewOSFileName`. Буде успішним, якщо файла з іменем `NewOSFileName` немає, а обидва імені є допустимими файловими іменами.

`disk (Path)` – використовують для зміни поточного диска або каталогу.

`copyfile (SourceName, DestinationName)` – використовують для копіювання файлів.

`openfile (SymName, OSName, OpenMode, Attributes, Creation)` – спеціальний предикат, який забезпечує відкриття файлів нестандартними способами. `SymName` та `OSName` – це те саме, що і в раніше описаних предикатах. Решта аргументів така:

`OpenMode` – це режими доступу та розділення до файла. Їх формують, об'єднуючи одне зі значень `fm_access_xx` (`fm_access_wo` "тільки для запису", `fm_access_rw` "для читання та запису", `fm_access_ro` "тільки для читання"), одне з `fm_sh_XXXXXX` (`fm_sh_deny rw` "забороняє всі режими доступу для інших процесів", всі константи для файлових атрибутів знаходяться у файлі `IODECL.CON`) та (необов'язково) `fm_returnerr` та `fm_writethru`. Якщо режим доступу не задано, тоді він буде встановлений з атрибутом "тільки для читання". Якщо режим розділення не задано, він буде встановлений з атрибутом "запис заборонено".

`Attributes` – це фізичні атрибути файла. Доступні атрибути для DOS – це `fa_ronly`, `fa_hidden`, `fa_system`, `fa_arch` та `fa_normal`. Якщо нічого не задано (0), атрибути будуть встановлені за замовчуванням `fa_normal`.

`Creation` – визначає, що робити у разі браку або наявності файла з вказаним ім'ям. Його формують додаванням одного параметра з групи `cr_ex_xx` та одного параметра з групи `cr_noex_xx`. За замовчуванням будуть встановлені параметри `cr_ex_fail` та `cr_noex_fail`, тобто неспадча при наявності та при браку файла. Реальна дія `Creation` за замовчуванням буде встановлена відповідно до режиму доступу:

```
fm_access_ro → cr_ex_open + cr_noex_fail
fm_access_wo → cr_ex_replace + cr_noex_create
fm_access_rw → cr_ex_open + cr_noex_create.
```

2.13. Робота з рядками символів

`Visual Prolog` дає змогу підтримувати набір стандартних предикатів для ефективної роботи з рядками символів. У розділі розглянемо базові предикати для роботи з рядками та предикати перетворення типів даних.

`frontchar (String1, Char, String2)` – предикат діє відповідно до рівності:

`String1 = об'єднання Char та String2`

та може бути використаний для розділення рядка на послідовність символів та для перевірки на наявність символу у рядку. Якщо аргумент `String1` зв'язаний з рядком нульової довжини, тоді предикат завершується безуспішно. Інший приклад демонструє використання предиката для перетворення рядка у список символів.

Приклад 2.24

```
domains
    charlist = char*
predicates
    string_chlist(string, charlist)
clauses
```

```

string_chlist("", []):-!.
string_chlist(S, [H|T]):-
    frontchar(S,H,S1),
    string_chlist(S1,T).
goal
string_chlist("ABC",Z).

```

Після оброблення цільового твердження змінна *Z* буде зв'язана зі списком ['A', 'B', 'C'].

fronttoken(String1,Token,Rest) – виконує три взаємопов'язані функції, залежно від типу потоку аргументів. У разі потоку (i,o,o) **fronttoken** знаходить першу лексему в *String1*, зв'язує її з *Token*, залишок *String1* зв'язує з *Rest*. Варіанти потоків (i,i,o), (i,o,i), (i,i,i) слугують для перевірки: чи зв'язані аргументи відповідають частинам *String1* (першій лексемі, всьому, що знаходиться після першої лексеми, або тому й тому), тоді **fronttoken** завершується успішно, інакше – безуспішно. Якщо був використаний потік (o,o,i), тоді предикат створює об'єднання *Token* і *Rest* та зв'язує *String1* з результатом. Ще один приклад засвідчує використання **fronttoken** для розбиття речення на список імен.

Приклад 2.25

```

domains
    namelist = name*
    name = symbol
predicates
    string_namelist(string,namelist)
clauses
    string_namelist(S,[H|T]):-
        fronttoken(S,H,S1),!,
        string_namelist(S1,T).
    string_namelist(_,[]).
goal
    string_namelist("ivan petro roman ira inna",X).

```

Після виконання цільового твердження змінна *X* буде пов'язана зі списком [ivan, petro, roman, ira, inna].

frontstr(NumberOfChars,String1,StartStr,EndStr) – поділяє рядок *String1* на дві частини, де *StartStr* містить *NumberOfChars* перших символів з *String1* та *EndStr* містить залишок. При зверненні до

frontstr перші два параметра мають бути пов'язані, а останні два – вільними.

concat(String1,String2,String3) – об'єднує два рядки *String2* та *String3* у рядок *String1*. Цей предикат завжди дає тільки один розв'язок, тобто є детермінованим.

str_len(StringArg,Length) – дає змогу виконувати такі завдання: визначає чи перевіряє довжину рядка або повертає рядок пробілів заданої довжини. У разі потоку (i,o) зв'язує змінну *Length* з довжиною рядка *StringArg*, якщо заданий потік (i,i) – перевіряє, чи має *StringArg* задану довжину *Length*, у разі потоку (o,i) повертає рядок пробілів заданої довжини.

isname(String) – перевіряє, чи є аргумент *String* допустимим ім'ям відповідно до синтаксису Visual Prolog програми. Ім'я має починатися з букви алфавіту або символу підкреслення, за яким слідує довільна кількість букв, цифр та символів підкреслення.

format(OutputString,FormatString,Arg1,Arg2,Arg3,...,ArgN) – виконує перетворення, аналогічні до предиката **writeln** (див. 2.10) та повертає результат у вигляді рядкової змінної.

subchar(String,Position,Char) – повертає символ у заданій позиції рядка. Перший символ рядка має позицію 1. Якщо позиція символу виходить за межі рядка, тоді **subchar** завершиться помилкою.

substring(Str_in,Pos,Len,Str_out) – повертає частину рядка. Змінна буде пов'язана з копією частини рядка *Str_in*, починаючи з символу на позиції *Pos* та довжиною *Len*.

searchchar(String,Char,Position) – повертає позицію першого входження заданого символу у рядок. Якщо символ не знайдений, тоді предикат завершиться неуспішним результатом. Слід відзначити, що до знаходження всіх позицій входження символу у рядок даний предикат не може бути застосований. Ще один приклад засвідчує пошук усіх позицій входження символу у рядок.

Приклад 2.26

```

predicates
    nd_searchchar(string,char,integer)
    nd_searchchar1(string,char,integer,integer)
    nd_sc(string,char,integer,integer,integer)
clauses

```

```

nd_searchchar(Str,Ch,Pos):-
    nd_searchchar1(Str,Ch,Pos,0).
nd_searchchar1(Str,Ch,Pos,Old):-
    searchchar(Str,Ch,Pos1),
    nd_sc(Str,Ch,Pos,Pos1,Old).
nd_sc(_,_,Pos,Pos1,Old):- Pos = Pos1+Old.
nd_sc(Str,Ch,Pos,Pos1,Old):-
    frontstr(Pos1,Str,_,Rest),
    Old1 = Old + Pos1,
    nd_searchchar1(Rest,Ch,Pos,Old1).
goal
nd_searchchar("abbablablabala",'a',P),
write(P,"\\n"),
fail.

```

Це є реалізація недетермінованого предиката `nd_searchchar`, який є повністю сумісний з `searchchar`.

`searchstring(SourceStr,SearchStr,Pos)` – повертає позицію першого входження заданого рядка в інший рядок. Якщо шуканий рядок `SearchStr` не знайдений, або його довжина більша за `SourceStr`, тоді `searchstring` завершиться безуспішно.

`char_int(Char,Integer)` – перетворює символ у ціле число, або, навпаки, залежно від того, які змінні пов'язані між собою. Якщо обидва аргументи пов'язані, тоді перевіряє, чи відповідають значення аргументів один одному.

`str_char(String,Char)` – перетворює рядок з одного символу у символний тип даних та навпаки. Якщо пов'язані обидві змінні, то `str_char` завершується успішно у тому разі, коли змінна `String` пов'язана з односимвольним рядком, що відповідає `Char`.

`str_int(String,Integer)` – перетворює рядок, який містить текстове подання цілого числа, у цілий тип, даних, або залежно від того, які змінні є пов'язані. У разі потоку параметрів (i,i) завершується успішно, за умови, що `Integer` пов'язана з цілим еквівалентом числа, зображеного за допомогою `String`.

`str_real(String,Real)` – перетворює рядок у дійсне число або дійсне число у рядок. У випадку потоку параметрів (i,i) завершується успішно, якщо `Real` пов'язана з дійсним числом, рівним числу, поданому у `String`.

`upper_lower(Upper,Lower)` – перетворює символи верхнього регістру у символи нижнього регістру. Якщо обидва параметри пов'язані, тоді `upper_lower` завершуються успішно у тому разі, якщо `Upper` та `Lower` пов'язані з рядками, які конвертуються один в один. Наприклад, цільове твердження:

```

goal
Str1=samPLeStriNG,
Str2=SAMpleSTRiNg,
upper_lower(Str1,Str2)

```

завершиться успішно.

`term_str(Domain,Term,String)` – універсальний предикат для перетворення, дає змогу перетворювати терм заданого домену у його рядкове подання, `Domain` задає, до якого домену належить терм (це не обов'язково стандартний домен, він може бути визначений також користувачем). Предикат `term_str` може замінити всі предикати перетворення типів `str_*`, наприклад, `str_real` може бути реалізований як `str_real(S,R):- term_real(real,R,S)`.

Ще один приклад демонструє використання предикатів `isname`, `str_int`, `str_char` для визначення типу лексем, які були введені з клавіатури. Визначено предикат `scanner`, який перетворює вхідний рядок у список лексем за допомогою предиката `fronttoken`.

Приклад 2.27

```

domains
tok = numb(integer); name(string); char(char)
toklist = tok*
predicates
scanner(string,toklist)
maketok(string,tok)
clauses
scanner("",[]).
scanner(Str,[Tok|Rest]):-
    fronttoken(Str, Sym, Str1), maketok(Sym, Tok), scanner(Str1, Rest).
maketok(S,name(S)):-isname(S).
maketok(S,numb(N)):-str_int(S,N).
maketok(S,char(C)):-str_char(S, C).
goal
write("Enter some text:"),nl,
readln(Text),nl,

```

```
scanner(Text,T_List),
write(T_List).
```

Необхідно зауважити, що перетворення між типами доменів `symbol` та `string`, а також між `char`, `integer` та `real` здійснюється автоматично, з використанням стандартних предикатів або в процесі оцінення математичних виразів. Дійсні числа заокруглюються в процесі автоматичного перетворення. Visual Prolog автоматично перетворює типи в разі потреби під час виклику предикатів:

```
predicates
  p(integer)
clauses
  p(X):-write("The integer value is ",X,'\n').
```

Інші цілі дадуть однаковий результат у цьому прикладі:

```
X=97.234, p(X);
X=97, p(X);
X='a', p(X).
```

2.14. Комп'ютерні проекти

- Для речень необхідно скласти правила Visual Prolog:
 - студент отримує стипендію, якщо він добре вчиться;
 - кожному подобається робота, якщо вона приносить задоволення та добре оплачується;
 - людина може купити машину, якщо машина подобається їй, має відповідну ціну та колір.
- Записати мовою Visual Prolog правила:

"Іван важко працює", "Якщо Іван важко працює, то він пасивний хлопець" та "Якщо Іван пасивний хлопець, то він не знайде ліпшої роботи". Використати правила виведення для обґрунтування висновку з цих гіпотез: "Іван не знайде ліпшої роботи".
- Довести, що у прикладах *a-c* висновки можна зробити з таких гіпотез:
 - "Усі леви жорстокі істоти", "Деякі леви не п'ють кави". Висновок: "Деякі жорстокі істоти не п'ють кави".

- "Усі колібри мають яскраве пір'я", "Жодний великий птах не їсть меду і не має яскравого пір'я". Висновок: "Колібри маленькі птахи".
 - "Кожний атлет сильний", "Кожний, хто сильний і розумний, доб'ється успіху", "Петро – атлет", "Петро – розумний". Висновок: "Петро доб'ється успіху".
- Задано базу фактів вигляду: *X* переміг *Y*. Написати програму, яка буде відповідати на такі запитання:
 - Хто програв *X*?
 - Скільки перемог одержав *X*?
 - Чи може *X* перемогти *Y* (може, якщо він вже переміг, або *X* може перемогти того, хто може перемогти *Y*)?
 - Чи правильно, що є *X*, який може перемогти всіх інших (може, якщо він вже переміг або *X* може перемогти того, хто може перемогти *Y*)?
 - Чи є коректною база фактів (ніхто не може перемогти сам себе)?
 - Задано базу фактів вигляду: *X* – донька *Y* та *Z*; *X* – син *Y* та *Z*. Написати програму, яка буде відповідати на такі запитання:
 - Хто батьки *X*?
 - Якої статі *X*?
 - Чи є родичами *X* та *Y*?
 - Чи існує пара, від якої всі з'явилися?
 - Хто є нащадками *X*?
 - Чи є коректною база фактів (чи не є хтось своїм предком, у кожного *X* тільки один батько та мати).
 - Задано базу фактів вигляду: автобус *X* проїжджає зупинку *Y* в час *T* (часи, хвилини). Написати програму, яка буде відповідати на такі запитання:
 - Скільки всього автобусів?
 - Скільки всього зупинок?
 - Який маршрут автобуса *X*?
 - Чи правда, що з довільної зупинки можна добратися до довільної іншої зупинки?

- e) Чи правда, що не має таких двох зупинок і такого автобуса, що цей автобус ходить через декілька інших зупинок швидше, ніж якийсь інший автобус (без проміжних зупинок)?
 - f) Як найшвидше добратися з зупинки *A* до зупинки *B* у час *T*?
 - g) Чи є коректною база фактів (немає двох зупинок на яких автобус опиняється одночасно)?
 - h) Скільки часу займає весь маршрут автобуса *X*?
 - i) Скільки маршрутів перетинаються (мають спільні зупинки)?
- 7 Використовуючи оптимізовану хвостову рекурсію, реалізувати такі програми:
- a) Піднесення числа *X* до заданого степеня *N* (*N* – натуральне число);
 - b) Знаходження факторіала заданого натурального числа *N*;
 - c) Обчислення *N*-го числа Фібоначчі;
 - d) Обчислення суми всіх цифр заданого натурального числа.
8. Використовуючи *рекурсивну* структуру даних, побудувати бінарне пошукове дерево (вхідні дані знаходяться у файлі) та здійснити його обхід у прямому, зворотному та внутрішньому порядках. Результат помістити у список.
9. Використовуючи структуру даних *список*, реалізувати такі програми:
- a) Підрахувати кількість додатних та від'ємних елементів списку;
 - b) Модифікувати список за таким правилом: від'ємні елементи піднести до квадрата, а додатні збільшити на задане натуральне число *n*;
 - c) Циклічно зсунути елементи списку на задану кількість позицій *n* у заданому напрямку (вліво чи вправо).
 - d) Видрукувати додатні елементи списку не більше, ніж по *N* елементів у рядку;
 - e) Підрахувати середнє арифметичне елементів списку;
 - f) Модифікувати список за правилом: відкинути додатні елементи, а від'ємні піднести до квадрата;
 - g) Посортувати елементи списку за кожним із методів (бульбашки, вставки, швидке сортування);
 - h) Портувати список на основі бінарного дерева;

- i) Знайти перетин двох списків;
 - j) Обернути список.
10. Створити базу даних з відомостями про пасажирів: П.І.Б., кількість місць, вага багажу. Визначити, чи є пасажир, багаж яких займає одне місце та перевищує 30 кг.
11. Створити базу даних студентів: прізвище, ім'я, рік народження. Отримати список студентів, які мають більше 20-ти років.
13. Створити базу даних, яка містить інформацію про робітників: П.І.Б., професія, оклад. Обчислити середньомісячну заробітну плату для інженерів.
14. Створити базу даних з розкладом руху потягів: номер потяга, пункт відправлення, пункт прибуття, час відправлення, час у дорозі, вартість квитка. Визначити маршрут руху з Луганська у Львів, час у дорозі та вартість квитка.
15. Створити базу даних з таблицею ігор чемпіонату з футболу: перша команда, друга команда, рахунок гри. Визначити чемпіона.
16. Створити базу даних з відомостями про вартість товару: назва товару, вартість. Визначити сумарну вартість вказаних у базі даних товарів, знайти товари з максимальною та мінімальною вартостями.
17. Створити базу даних книжок: автор, назва книги, видавництво, рік видавництва. Знайти всі книги, які були видані у видавництві "Наука" після 2000 року.
18. Створити базу даних читачів бібліотеки: прізвище, номер читачього квитка, дата відвідування бібліотеки. Визначити кількість читачів, які відвідали бібліотеку протягом якогось місяця.

3. ПРИКЛАДИ

3.1. Задача про Ханойські вежі

Гра “Ханойські вежі” – класичний приклад рекурсії. Нагадаємо, що суть гри полягає у тому, щоб перемістити всі диски з лівого на правий стержень, по одному за раз, причому диск більшого діаметра не може лежати вище диска меншого діаметра. Всього є три стержні, з яких два – вільні (рис. 3.1.).

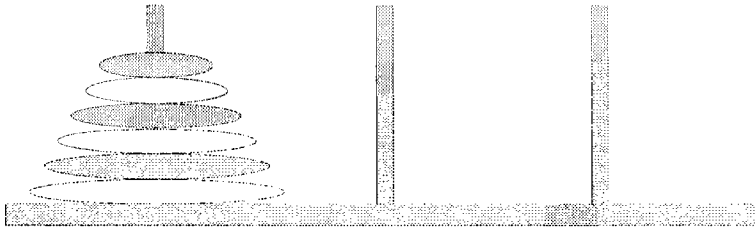


Рис. 3.1. Ханойська вежа

Для цієї гри розроблено таку стратегію:

- Один диск переміщують безпосередньо;
- N дисків переміщують поетапно:
 1. Перенести $N-1$ дисків на середній стержень;
 2. Перенести останній диск на правий стержень;
 3. Перенести $N-1$ дисків з середнього на правий стержень.

У програмі визначено три предикати:

- предикат `hanoi` з одним параметром, який задає кількість дисків;
- предикат `move`, за допомогою якого описують переміщення N дисків з одного стержня на інший, використовуючи третій стержень як проміжний.

- предикат `inform` – описує дію, яку необхідно виконати з конкретним диском.

Приклад 3.1.

```
domains
  loc =right; middle; left
predicates
  hanoi (integer)
  move (integer, loc, loc, loc)
  inform (loc, loc)
clauses
  hanoi (N):-
    move (N, left, middle, right).
  move (1, A, _, C):-
    inform (A, C),
    !.
  move (N, A, B, C):-
    N1=N-1,
    move (N1, A, C, B),
    inform (A, C),
    move (N1, B, A, C).
  inform (Loc1, Loc2):-
    write ("Move a disk from", Loc1, " to ", Loc2),nl.
goal
  hanoi(3).
```

Результатом виконання програми з трьома дисками буде:

```
Move a disk from left to right
Move a disk from left to middle
Move a disk from right to middle
Move a disk from left to right
Move a disk from middle to left
Move a disk from middle to right
Move a disk from left to right.
```

3.2. Задача про N ферзів

Задача полягає у розміщенні n -ферзів на дошці $n \times n$ таким чином, щоб жодні два не були один одного. Два ферзі не можуть знаходитися на одній вертикалі, горизонталі або діагоналі. Для розв’я-

зування цієї задачі позначимо клітинки, розміщені по діагоналі та вертикалі, від 1 до N . Нумерацію діагоналей поділимо на два типи так, щоб кожну з них можна було однозначно задавати за типом та номером, який обчислюють на підставі номерів рядів по горизонталі та вертикалі:

$\text{Diagonal} = N + \text{Column} - \text{Row}$ (Тип 1)

$\text{Diagonal} = \text{Row} + \text{Column} - 1$ (Тип 2).

Нумерацію діагоналей першого та другого типів на шаховій дошці розміру 4×4 зображено відповідно на рис. 3.2 та рис. 3.3.

	1	2	3	4
1	4	5	6	7
2	3	4	5	6
3	2	3	4	5
4	1	2	3	4

Рис. 3.2. Нумерація діагоналей першого типу

	1	2	3	4
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6
4	4	5	6	7

Рис. 3.3. Нумерація діагоналей другого типу

Щоб розв'язати цю задачу програмним шляхом, потрібно скласти список тих вертикалей, горизонталей та діагоналей, які є вільними, а також тих, на яких розміщені ферзі. Положення ферзя на шаховій дошці визначається номером рядка по горизонталі та вертикалі. У розділі domains вводимо опис:

$\text{queen} = q(\text{integer}, \text{integer}).$

Це оголошення засвідчує позицію ферзя. Для опису множини позицій можемо використати список:

$\text{queens} = \text{queen}^*.$

Також необхідно ввести декілька списків для вертикалей, горизонталей та діагоналей, які не зайняті ферзями. Ці списки можна описати як:

$\text{freelist} = \text{integer}^*.$

Будемо розглядати шахову дошку як один об'єкт за допомогою опису:

$\text{board} = \text{board}(\text{queens}, \text{freelist}, \text{freelist}, \text{freelist}, \text{freelist}).$

Списки *freelist* містять вільні горизонталі, вертикалі та діагоналі першого та другого типів відповідно.

Порожню дошку розміром 4×4 можна зобразити у вигляді:

$\text{board}([], [1,2,3,4], [1,2,3,4], [1,2,3,4,5,6,7], [1,2,3,4,5,6,7]).$

Дошка з одним ферзем у лівому верхньому куті:

$\text{board}([q(1,1)], [2,3,4], [2,3,4], [1,2,3,5,6,7], [2,3,4,5,6,7]).$

Тепер задача може бути розв'язана за допомогою опису відношення між порожньою дошкою та дошкою з N ферзями. Введем предикат

$\text{placeN}(\text{integer}, \text{board}, \text{board}),$

для якого запишемо два речення.

Ферзі розміщують по одному до того моменту, поки не будуть зайняті всі вертикалі та горизонталі:

$\text{placeN}(_, \text{board}(\text{D}, [], [], \text{X}, \text{Y}), \text{board}(\text{D}, [], [], \text{X}, \text{Y})) :-!$

У другому реченні предикат *place_a_queen* встановлює зв'язок між позиціями *Board1* та *Board2*. (У позиції *Board2* на одного ферзя більше, ніж у *Board1*.)

$\text{placeN}(N, \text{Board1}, \text{Result}) :-$

$\text{place_a_queen}(N, \text{Board1}, \text{Board2}),$

$\text{placeN}(N, \text{Board2}, \text{Result}).$

Для предиката *place_a_queen* використаємо такий опис:

$\text{place_a_queen}(\text{integer}, \text{board}, \text{board}).$

Ключовим питанням під час розв'язування задачі розміщення ферзів є опис того, як потрібно ставити чергового ферзя, починаючи з порожньої дошки. Розв'язуючи задачу, додаємо нового ферзя у список тих, які вже стоять на дошці:

$[q(R, C)|\text{Queens}].$

Серед горизонталей *Rows*, які лишились, потрібно знайти ту горизонталь *R*, на яку ми можемо поставити ферзя. Водночас горизонталь *R* потрібно видалити зі списку вільних горизонталей, унаслідок чого буде отриманий новий список вільних горизонталей *NewR*. Цю процедуру запишемо як

$\text{findandremove}(R, \text{Rows}, \text{NewR}).$

Отже, потрібно знайти незайняту вертикаль C . За допомогою R та C можуть бути обчислені діагоналі, на яких розміщено ферзь. Потім перевірити, чи знаходяться діагоналі $D1$ та $D2$ у списках незайнятих діагоналей. Нижче подано речення предиката `place_a_queen`, яке реалізує ці функції:

```
place_a_queen (N, board (Queens, Rows, Columns, Diag1, Diag2),
  board([q(R,C)|Queens], NewR, NewC, NewD1, NewD2)):-
  findandremove (R, Rows, NewR),
  findandremove (C, Columns, NewC),
  D1=N+C-R, findandremove (D1, Diag1, NewD1),
  D2=R+C-1, findandremove (D2, Diag2, NewD2).
```

Наведемо повний текст програми. У нього введено предикат `nqueens`, який вказує на розмірність задачі.

Приклад. 3.2.

```
domains
  queen = q (integer, integer)
  queens = queen*
  freelist = integer*
  board = board (queens, freelist, freelist, freelist, freelist)
predicates
  placeN (integer, board, board)
  place_a_queen (integer, board, board)
  nqueens (integer)
  makelist (integer, freelist)
  findandremove (integer, freelist, freelist)
  nextrow (integer, freelist, freelist)
clauses
  nqueens(N):-
    makelist (N, L),
    Diagonal = N*2-1,
    makelist (Diagonal, LL),
    placeN (N, board ([], L,L, LL, LL), Final),
    write (Final).
  placeN (_, board (D, [], [], D1, D2), board (D, [], [], D1, D2)):-!.
  placeN (N, Board1, Result):-
    place_a_queen (N, Board1, Board2),
    placeN (N, Board2, Result).
  place_a_queen (N, board (Queens, Rows, Columns, Diag1, Diag2),
    board ([q(R,C)|Queens], NewR, NewC, NewD1, NewD2)):-
```

```
  findandremove (R, Rows, NewR),
  findandremove (C, Columns, NewC),
  D1=N+C-R, findandremove (D1, Diag1, NewD1),
  D2=R+C-1, findandremove (D2, Diag2, NewD2).
  findandremove (X, [X|Rest], Rest).
  findandremove (X, [Y|Rest], [Y|Tail]):-
    findandremove (X, Rest, Tail).
  makelist (1, [1]).
  makelist (N, [N|Rest]) :-
    N1=N-1, makelist (N1, Rest).
```

goal

```
nqueens(5),
nl.
```

Результатом виконання програми стане:

```
board([q(1,2), q(2,4), q(3,1), q(4,3), q(5,5)], [], [], [9,8,2,1], [8,7,4,1]).
```

3.3. Пошук у лабіринті

Припустимо, що мандрівник повинен знайти скарб із золотом, який знаходиться у печері під землею, та вберегти його грабіжників та чудовищ. Підземелля складається з лабіринту коридорів, які зв'язують окремі печери. Відомо, що все золото заховане в одній печері, необхідно визначити маршрут, який дасть можливість знайти золото та повернутися неушкодженим. Карту підземелля задано на рис. 3.4.

За допомогою Visual Prolog введемо карту у програму та спробуємо визначити небезпечний маршрут. Кожен коридор зображено як окремий факт. Правила задані з використанням предикатів `go` та `route`. Відповідь міститиме список печер, які необхідно пройти, щоб здобути скарб та повернутися неушкодженим.

Усі пройдені печери заносять у список за допомогою предиката `route`, який визначено рекурсивно. Третій параметр предиката містить список усіх відвіданих печер. Якщо у список входить печера з золотом `gold`, тоді ціль досягнута. Інакше список відвіданих печер доповнюється `Nextroom` за умови, що перед цим мандрівник не побував ні в одній з небезпечних печер. Наведемо текст програми 3.3.

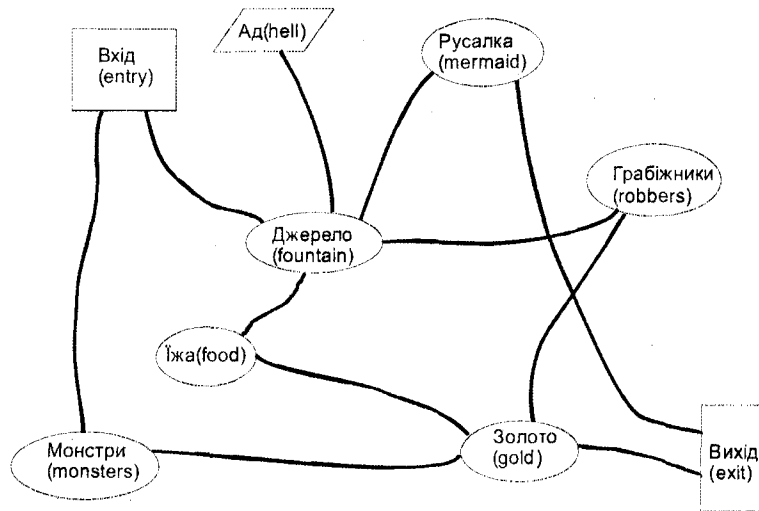


Рис. 3.4. Карта підземелля

Приклад. 3.3.

```
domains
    room = symbol
    roomlist = room*
predicates
    gallery(room,room) /* Коридор між двома кімнатами */
    neighborroom(room,room) /* Предикат необхідний, тому що */
    /* невідомо, в якому напрямку рухатись по коридору */
    avoid(roomlist)
    go(room,room)
    route(room,room,roomlist) /* Цим маршрутом треба йти */
    /* roomlist містить список всіх відвіданих печер */
    member(room,roomlist)
clauses
    gallery(entry,monsters).
    gallery(entry,fountain).
    gallery(fountain,hell).
    gallery(fountain,food).
    gallery(exit,gold).
    gallery(fountain,mermaid).
```

```
gallery(robbers,gold).
gallery(fountain,robbers).
gallery(food,gold).
gallery(mermaid,exit).
gallery(monsters,gold).
gallery(gold,exit).
neighborroom(X,Y):-gallery(X,Y).
neighborroom(X,Y):-gallery(Y,X).
avoid([monsters,robbers]).
go(Here,There):-route(Here,There,[Here]).
go(_,_).
route(Room,Room,VisitedRooms):-
    member(gold,VisitedRooms),
    write(VisitedRooms),nl.
route(Room,Way_out,VisitedRooms):-
    neighborroom(Room,Nextroom),
    avoid(DangerousRooms),
    not(member(NextRoom,DangerousRooms)),
    not(member(NextRoom,VisitedRooms)),
    route(NextRoom,Way_out,[NextRoom|VisitedRooms]).
member(X,[X|_]).
member(X,_[H]):-member(X,H).
goal
go(entry,exit).
```

До тексту програми можна додавати нові коридори та небезпечки, яких потрібно уникнути. Навіть якщо є декілька розв'язків задачі, програма видасть лише один. Щоб отримати всі можливі розв'язки, потрібно використати команду пошук з поверненням. Для цього додамо предикат fail у перше правило для предиката route:

```
route(Room,Room,VisitedRooms):-
    member(gold,VisitedRooms),
    write(VisitedRooms),nl,fail.
```

3.4. Пошук гамільтонового циклу в графі

Розглянемо задачу пошуку гамільтонових циклів у неорієнтованому графі, зображеному на рис. 3.5.

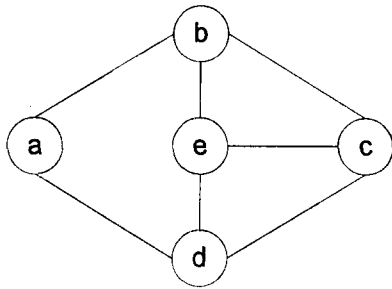


Рис. 3.5. Граф для пошуку всіх гамільтонових циклів

Для розв'язування цієї задачі у програмі потрібно описати граф. У Prolog граф можна зображати різними способами. Розглянемо один з них, коли граф описують як один об'єкт. У цьому разі графу відповідає пара списків: список вершин та список ребер. Для опису списку ребер необхідно спочатку ввести у розділі domains об'єкт ребро, який складається з двох суміжних вершин:

```
rib = r (symbol, symbol).
```

Тоді список ребер опишемо

```
rlist = rib*;
```

а список вершин

```
plist = symbol*.
```

Використовуючи введені домени, об'єкт граф можна зобразити як:

```
graph = graph (plist, rlist).
```

Отже, граф, зображений на рис. 3.4, набуде вигляду `graph([a,b,c,d,e], [r(a,b), r(a,d), r(b,c), r(b,e), r(c,d), r(e,d), r(e,c)])`.

Алгоритм побудови гамільтонового циклу визначимо так: починаємо з довільної вершини, шукаємо всі вершини суміжні з заданою, перевіряємо, чи можливий шлях, який проходить через усі вершини та з'єднує початкову вершину з однією із суміжних. Для реалізації цього алгоритму потрібно визначити предикат `route (symbol, symbol, graph, plist)`.

Цей предикат шукатиме шлях між двома заданими вершинами графу, який не містить циклів. Нижче подаємо його опис:

```
route (A, Z, Graph, Route):-
```

```
route1 (A, [Z], Graph, Route).
```

```
route1 (A, [A|Route1], _, [A|Route1]).
route1 (A, [Y|Route1], Graph, Route):-
    adjacent(X, Y, Graph),
    not (member (X, Route1)),
    route1 (A, [X, Y|Route1], Graph, Route).
```

Для того, щоб знайти шлях Route між A та Z, потрібно:

якщо $A=Z$, вважати $Route=[A]$ або знайти шлях Route1 з довільної вершини Y в Z, потім знайти шлях з A в Y, який не містить вершин з Route1. У цьому формулюванні неявно простежується, що є ще одне відношення, яке відповідає пошуку шляху з наступними обмеженнями: шлях не має проходити крізь вершини деякої підмножини (тут Route1) множини всіх вершин графу. У зв'язку з цим був визначений предикат

```
route1 (A, Route1, Graph, Route).
```

Аргументи відповідно з рис. 3.5 набудуть такого змісту:

A – деяка вершина;

Graph – граф;

Route1 – шлях в Graph;

Route – шлях в Graph з вершини A у початкову вершину шляху Route1, далі – вздовж Route1, до його завершення.

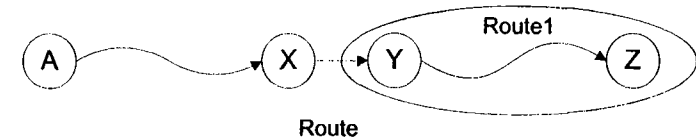


Рис. 3.6. Відношення route1

Між Route та Route1 є таке співвідношення:

```
route (A, Z, Graph, Route):-route1 (A, [Z], Graph, Route).
```

На рис. 3.6 зображено ідею рекурсивного визначення шляху Route1. Можливий граничний випадок, коли початкова вершина шляху Route1 (Y на рис. 3.5), збігається з початковою вершиною A шляху Route. У нашому прикладі шлях Route ототожнено з шляхом Route1 таким реченням:

```
route1 (A, [A|Route1], _, [A|Route1]).
```

Якщо початкові вершини цих двох шляхів не збігаються, тоді має бути така вершина X, що

1. Y – вершина суміжна з X,
2. X не міститься в Route1, тоді
3. X включається у Route1 як початкова вершина.

```
route1 (A, [Y|Route1], Graph, Route):-
    adjacent (X, Y, Graph),
    not (member (X, Route1)),
    route1 (A, [X,Y|Route1], Graph, Route).
```

Предикат adjacent визначає суміжну вершину X з Y, member – перевіряє, чи належить вершина X шляху Route1.

Описавши предикат route, далі можемо визначити процедуру пошуку гамільтонового циклу:

```
hamilton (graph ([Y|H], Rib), Route) :-
    adjacent (X, Y, graph ([Y|H], Rib)),
    route (Y, X, graph ([Y|H], Rib), Route),
    allpoint (Route, graph ([Y|H], Rib)).
```

Предикат adjacent буде по чергову вибирати вершини, суміжні з Y, route – знаходитиме шлях Route між вершиною Y та однією з суміжних вершин X, allpoint – перевірятиме, чи знайдений шлях Route містить всі вершини.

Нижче наведено повний текст програми.

Приклад. 3.4.

```
domains
    rib = r (symbol, symbol)
    plist = symbol*
    rlist = rib*
    graph = graph (plist, rlist)
predicates
    adjacent (symbol, symbol, graph)
    route (symbol, symbol, graph, plist)
    route1 (symbol, plist, graph, plist)
    member (rib, rlist)
    member (symbol, plist)
    hamilton (graph, plist)
    allpoint (plist, graph)
clauses
```

```
route (A, Z, Graph, Route):-
    route1 (A, [Z], Graph, Route).
route1 (A, [A|Route1], _, [A|Route1]).
route1 (A, [Y|Route1], Graph, Route):-
    adjacent (X, Y, Graph),
    not (member (X, Route1)),
    route1 (A, [X,Y|Route1], Graph, Route).
adjacent (X, Y, graph (Point, Rib)) :-
    member (r(X,Y), Rib);
    member (r(Y,X), Rib).
member (X, [X|_]).
member (X, [_|L]):-member (X, L).
hamilton (graph ([Y|H], Rib), Route) :-
    adjacent(X, Y, graph ([Y|H], Rib)),
    route(Y, X, graph ([Y|H], Rib), Route),
    allpoint (Route, graph([Y|H], Rib)).
allpoint (_, graph ([], _)).
allpoint (Route, graph ([H|T], Rib)):-
    member (H, Route), allpoint (Route, graph(T, Rib)).
goal
    hamilton (graph ([a,b,c,d,e], [r(a,b),r(a,d),r(b,c),r(b,e),r(c,d),
        r(e,d),r(e,c)]), Route).
```

Результат виконання програми – чотири гамільтонові цикли, які починаються вершиною a.

```
Route=["a","d","e","c","b"]
Route=["a","d","c","e","b"]
Route=["a","b","e","c","d"]
Route=["a","b","c","e","d"]
4 Solutions.
```

3.5. Пошук найкоротшого шляху

Кожному ребру графу можна приписати його вагу. Для того, щоб відношення Route та Route1 з попереднього прикладу працювали з вагами, потрібно ввести додаткові аргументи: W – вага маршруту Route, W1 – вага маршруту Route1. Ребро графу також буде містити вагу як додатковий аргумент:

```
rib = r (symbol, symbol, unsigned).
```

Нижче описано фрагмент для знаходження ваги маршруту між двома заданими вершинами

```
route (A, Z, Graph, Route, W):-
    route1 (A, [Z], 0, Graph, Route, W).
route1 (A, [A|Route1], W1, _, [A|Route1], W1).
route1 (A, [Y|Route1], W1, Graph, Route, W):-
    adjacent (X, Y, Wxy, Graph),
    not (member (X, Route1)),
    W2=W1+Wxy,
    route1 (A, [X,Y|Route1], W2, Graph, Route, W).
```

Предикат `adjacent` містить додатковий аргумент `Wxy` – вага ребра між вершинами `X` та `Y`.

Процедуру можна використати для знаходження мінімального шляху між двома вершинами `a` та `b`.

```
route(a,b,Graph,MinRoute,MinW),route(a,b,Graph,_,W),W>MinW.
```

Для того, щоб знайти шлях максимальної вартості, потрібно задати

```
route(a,b,Graph,MaxRoute,MaxW),route(a,b,Graph,_,W),W<MaxW.
```

Наведемо повний код програми знаходження шляху мінімальної вартості для зваженого графу, зображеного на рис. 3.7., між вершинами `a` і `d`.

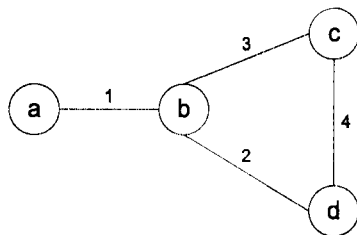


Рис. 3.7. Зважений граф

Приклад 3.5.

```
domains
    rib = r (symbol, symbol, unsigned)
    plist = symbol*
    rlist = rib*
    graph = graph(plist, rlist)
predicates
```

```
adjacent (symbol, symbol, unsigned, graph)
route (symbol, symbol, graph, plist, unsigned)
route1 (symbol, plist, unsigned, graph, plist, unsigned)
member (rib, rlist)
member (symbol, plist)
clauses
    route (A, Z, Graph, Route, W):-
        route1 (A, [Z], 0, Graph, Route, W).
    route1 (A, [A|Route1], W1, _, [A|Route1], W1).
    route1 (A, [Y|Route1], W1, Graph, Route, W):-
        adjacent (X, Y, Wxy, Graph),
        not (member (X, Route1)),
        W2 = W1+Wxy,
        route1 (A, [X,Y|Route1], W2, Graph, Route, W).
    adjacent (X, Y, W, graph (Point, Rib)) :-
        member (r(X, Y, W), Rib);
        member (r(Y, X, W), Rib).
    member (X, [X|_]).
    member (X, [_|L]):-member (X, L).
```

goal

```
Graph = graph([a,b,c,d], [r(a,b,1),r(b,d,2),r(b,c,3),r(c,d,4)]),
route(a,d,Graph,MinRoute,MinW),route(a,d,Graph,_,W),W>MinW.
```

Результат виконання програми буде:

```
Graph = graph(["a","b","c","d"],
    [r("a","b",1), r("b","d",2), r("b","c",3), r("c","d",4)]),
MinRoute = ["a","b","d"], MinW=3, W=8
1 Solution.
```

Цей спосіб не є оптимальним для знаходження максимальних та мінімальних шляхів між вершинами графу, оскільки він дає змогу переглядати всі можливі шляхи та не підходить для великих графів через високу часову складність. Для розв'язання задачі можна використовувати "жадібні" алгоритми.

3.6. Побудова каркасного дерева зв'язного графу

Визначимо процедуру `spantree` для побудови каркаса зв'язного графа, зображеного на рис. 3.8.

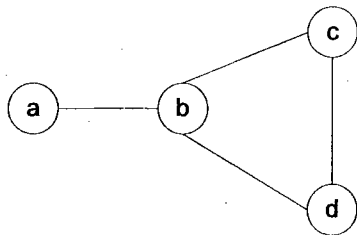


Рис. 3.8. Зв'язний граф

Граф будемо зображати як список ребер, для цього визначимо необхідні домени

```
rib = r (symbol, symbol)
rlist = rib*
graph = graph (rlist).
```

Алгоритм побудови каркасного дерева можна описати таким чином. Почати необхідно з порожньої множини ребер та поступово додавати нові, слідкуючи за тим, щоб не утворювалися цикли. Продовжувати цей процес до тих пір, поки не виясниться, що додавання ще одного ребра призведе до утворення циклу. Брак циклів можна забезпечити так: ребро приєднують до дерева у тому разі, коли одна з вершин міститься у каркасному дереві, а інша ще ні.

Побудову каркасного дерева Tree починаємо з першого ребра графу

```
spantree (Graph, Tree):-
  member (Rib, Graph),
  enlarge ([Rib], Tree, Graph), !.
```

Предикат member (Rib, Graph) вибирає перше ребро Rib з графу, процедура enlarge додає це ребро до каркасного дерева Tree, та шукає нове ребро, яке можна додати. Якщо ребро не знайдене, то завершує роботу.

```
enlarge (Tree1, Tree, Graph):-
  addrib (Tree1, Tree2, Graph),
  enlarge (Tree2, Tree, Graph).
enlarge (Tree, Tree, Graph):-
  not (addrib (Tree, _, Graph)).
```

Додає нове ребро предикат addrib.

```
addrib (Tree, [r(A,B)|Tree], Graph):-
```

```
adjacent (A, B, Graph),
point (A, Tree),
not (point (B, Tree)).
```

Для додавання шукають такі дві суміжні вершини (A та B), що одна належить каркасному дереву point (A, Tree), а друга ні not(point(B,Tree). Подаємо повний текст програми.

Приклад. 3.6.

```
domains
  rib = r (symbol, symbol)
  rlist = rib*
  graph = graph (rlist)
predicates
  spantree (rlist, rlist)
  equal (rlist, rlist)
  adjacent (symbol, symbol, rlist)
  enlarge (rlist, rlist, rlist)
  addrib (rlist, rlist, rlist)
  point (symbol, rlist)
  member (rib, rlist)
clauses
  spantree (Graph, Tree):-
    member (Rib, Graph),
    enlarge ([Rib], Tree, Graph), !.
  enlarge (Tree1, Tree, Graph):-
    addrib (Tree1, Tree2, Graph),
    enlarge (Tree2, Tree, Graph).
  enlarge (Tree, Tree, Graph):-
    not (addrib (Tree, _, Graph)).
  addrib (Tree, [r(A,B)|Tree], Graph):-
    adjacent (A, B, Graph),
    point (A, Tree),
    not (point (B, Tree)).
  adjacent (A, B, Graph):-
    member (r(A,B), Graph);
    member (r(B,A), Graph).
  point (A, Graph):-adjacent(A, _, Graph).
  member (X, [X|_]).
  member (X, [_|L]):-member (X, L).
goal
  Graph = [r(a,b), r(b,c), r(c,d), r(b,d)],
  spantree (Graph, Tree).
```

Результатом виконання програми стане перше знайдене каркасне дерево.

```
Graph = [r("a","b"), r("b","c"), r("c","d"), r("b","d")],
```

```
Tree=[r("c","d"), r("b","c"), r("a","b")]
```

1 Solution.

3.7. Пошук вглиб

Відомо багато різних підходів, які дають можливість вирішувати проблеми пошуку на графах. У цьому розділі розглянемо один з них – пошук вглиб.

Розроблення програми почнемо з опису графу у вигляді списку суміжності. Для вершини введемо новий домен

```
nodes=n(symbol,plist).
```

Тоді граф буде наведений як список вершин

```
nlist=nodes*.
```

Реалізуватиме алгоритм пошуку вглиб предикат `depth(A,Route,Graph)`, де `A` – вершина, з якої починається обхід графа, `Route` – шлях обходу, `Graph` – граф по якому здійснюють обхід. Для побудови шляху обходу введемо додаткову процедуру

```
depth1 (A, Route1, Route, Graph),
```

яка містить аргумент `Route1` – біжучий шлях обходу.

Обхід починаємо з довільної вершини `A` та відразу включаємо її в біжучий шлях обходу `Route1`

```
depth (A, Route, Graph):-depth1 (A, [A], Route, Graph).
```

Далі перевіряємо, чи не містить шлях `Route1` всі вершини графа `Graph`. Якщо ні, тоді вибираємо довільну вершину `Y` суміжну з `A`, яка ще не міститься в `Route1`, та додаємо її у шлях обходу. Наступну ітерацію починаємо вже з вершини `Y`. Якщо ж немає суміжної вершини з `Y`, повертаємося до вершини, з якої прийшли в `Y`, та шукаємо іншу суміжну вершину. Процес закінчуємо, коли шлях містить всі вершини графа. Описаний алгоритм можна зобразити у вигляді рекурсивної процедури

```
depth1 (A, Route1, Route, Graph):-
```

```
not (allpoint (Route1, Graph)),
```

```
Route2 = Route1,
```

```
add (A, Y, Route1, Route2, Graph),
depth1 (Y, [Y|Route1], Route, Graph),!.
depth1 (_, Route1, Route, Graph):-
allpoint (Route1, Graph), reverse (Route1, Route).
```

Предикат `add` шукає суміжну вершину `Y` з вершиною `A`, яка не міститься в `Route1`. Якщо такої вершини не знайдено, відбувається повернення до попередньої вершини.

```
add (A, Y, Route1, _, Graph):-
adjacent (A, Y, Graph),
not (member (Y, Route1)).
add (_, Y, Route1, [H1|Route2], Graph):-
add (H1, Y, Route1, Route2, Graph).
```

Наведемо повний код програми для обходу графа, зображеного на рис. 3.9 (а).

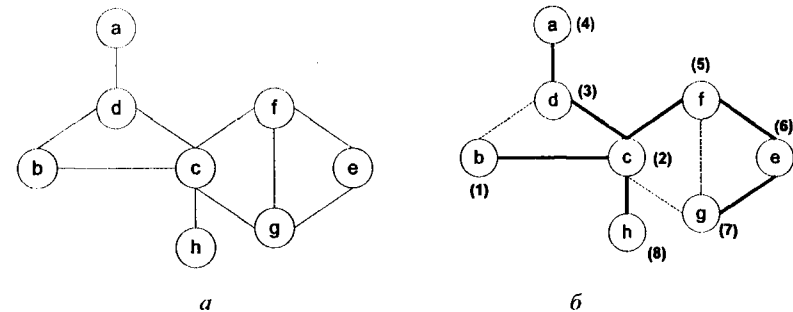


Рис. 3.9. Граф та каркасне дерево обходу графа пошуком вглиб

Приклад 3.7.

```
domains
```

```
rib = r (symbol, symbol)
```

```
plist = symbol*
```

```
nodes = n (symbol, plist)
```

```
nlist = nodes*
```

```
rlist = rib*
```

```
predicates
```

```
adjacent (symbol, symbol, nlist)
```

```
depth (symbol, plist, nlist)
```

```
depth1 (symbol, plist, plist, nlist)
```

```
add (symbol, symbol, plist, plist, nlist)
```

```

member (symbol, plist)
member1 (symbol, plist, nlist)
allpoint (plist, nlist)
reverse (plist, plist)
reverse1 (plist, plist, plist)
clauses
depth (A, Route, Graph):-
  depth1 (A, [A], Route, Graph).
depth1 (A, Route1, Route, Graph):-
  not (allpoint (Route1, Graph)),
  Route2 = Route1,
  add (A, Y, Route1, Route2, Graph),
  depth1 (Y, [Y|Route1], Route, Graph), !.
depth1 (_, Route1, Route, Graph):-
  allpoint (Route1, Graph), reverse (Route1, Route).

add(A, Y, Route1, _, Graph):-
  adjacent (A, Y, Graph),
  not (member (Y, Route1)).
add (_, Y, Route1, [H1|Route2], Graph):-
  add (H1, Y, Route1, Route2, Graph).
reverse ([H|List], RevList):-reverse1 (List, [H], RevList).
reverse1 ([H|T], T1, RevList):-reverse1 (T, [H|T1], RevList).
reverse1 ([], RevList, RevList).
adjacent (X, Y, NList) :-
  member1 (X, PList, NList),
  member (Y, PList).
member (X, [X|_]).
member (X, [_|L]):-member (X, L).
member1 (X, PList, [n(X,PList)|_]).
member1 (X, PList, [_|L]):-member1 (X, PList, L).
allpoint (_, []).
allpoint (Route, [n(X,_)|T]):-
  member (X, Route), allpoint(Route, T).
goal
depth(b,Route,[n(a,[d]),n(b,[c,d]),n(c,[b,d,f,g,h]),n(d,[a,b,c]),
  n(e,[f,g]),n(f,[c,e,g]),n(g,[c,e,f]),n(h,[c]))]).
Результатом виконання програми буде
Route=["b","c","d","a","f","e","g","h"]
1 Solution.

```

Порядок обходу вершин графа зображено на рис. 3.9 (б). Процедура `depth1` буде шлях обходу у зворотному порядку, тому був визначений предикат `reverse` для обертання шляху. Предикат `allpoint` дає змогу перевірити, чи всі вершини графа належать цьому шляху, `member1` – шукає список вершин суміжних із заданою.

3.8. Пошук вшир

Розглянемо інший спосіб систематичного обходу вершин графа – пошук вшир. Граф зобразимо як список суміжностей. На відміну від пошуку вглиб, обходити будемо всі суміжні вершини, починаючи з заданої. Розробляти програму почнемо з визначення предиката

`breadth (A, Route, Graph),`

де `A` – вершина, з якої почнемо обхід графа `Graph`, `Route` – шлях обходу.

Для побудови шляху обходу скористаємося додатковою процедурою

`breadth1 (Route1, Route2, Route, Graph),`

де `Route2` – список вершин, які ми вже обійшли, `Route1` – вершини, що стоять у черзі для обходу.

Обхід починаємо з вершини `A`, яку відразу ж заносимо у чергу `Route1`. Список `Route2` на початку порожній

`breadth (A, Route, Graph):-breadth1 ([A], [], Route, Graph).`

Далі беремо по одній вершині з черги `Route1`, переносимо їх у кінець списку `Route2`, вибираємо відповідні їм списки суміжних вершин, з яких додаємо в чергу `Route1` тільки ті, які ще не містяться в об'єднанні `Route2` з `Route1`. Процес закінчується тоді, коли черга `Route3` стане порожньою.

`breadth1 ([], Route, Route, _).`

`breadth1 ([H|Route1], Route0, Route, Graph):-`
`union (Route0, [H], Route2),`
`member1 (H, PList, Graph),`
`toAdd (PList, AddList, Route1, Route2),`
`union (Route1, AddList, Route3),`
`breadth1 (Route3, Route2, Route, Graph), !`

Для об'єднання списків був визначений додатковий предикат `union`, `member1` – знаходить для заданої вершини `H` список суміжних вершин `PList`, `toAdd` – створює новий список вершин `AddList` зі списку `PList`, які можна додати в кінець списку `Route1`.

Нижче наведено повний текст програми пошуку вшир для графу, зображеного на рис. 3.9 (а). Схематично порядок обходу графу зображено на рис. 3.10.

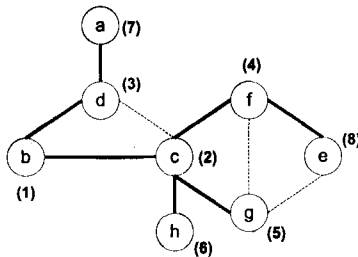


Рис. 3.10. Пошук вшир для графа з рис. 3.9(а)

Приклад 3.8.

domains

`rib = r (symbol, symbol)`

`plist = symbol*`

`nodes = n (symbol, plist)`

`nlist = nodes*`

`rlist = rib*`

predicates

`toAdd (plist, plist, plist, plist).`

`union (plist, plist, plist).`

`breadth (symbol, plist, nlist)`

`breadth1 (plist, plist, plist, nlist)`

`member (symbol, plist)`

`member1 (symbol, plist, nlist)`

clauses

`union ([], List, List).`

`union ([H|L1], List2, [H|L3]):-`

`union (L1, List2, L3).`

`toAdd ([], [], _, _).`

```
toAdd ([H|T], NewList, Route1, Route2):-
    union (Route1, Route2, Route3),
    member (H, Route3), !,
    toAdd (T, NewList, Route1, Route2).
toAdd ([H|T], [H|NewList], Route1, Route2):-
    toAdd (T, NewList, Route1, Route2).
breadth (A, Route, Graph):-
    breadth1 ([A], [], Route, Graph).
breadth1 ([], Route, Route, _).
breadth1 ([H|Route1], Route0, Route, Graph):-
    union (Route0, [H], Route2),
    member1 (H, PList, Graph),
    toAdd (PList, AddList, Route1, Route2),
    union (Route1, AddList, Route3),
    breadth1 (Route3, Route2, Route, Graph), !.
member (X, [X|_]).
member (X, [_|L]):-member (X, L).
member1 (X, PList, [n(X,PList)|_]).
member1 (X, PList, [_|L]):-member1 (X, PList, L).
goal
breadth(b,Route,[n(a,[d]),n(b,[c,d]),n(c,[b,d,f,g,h]),n(d,[a,b,c]),
    n(e,[f,g]),n(f,[c,e,g]),n(g,[c,e,f]),n(h,[c]))].
    Результатом виконання програми буде:
    Route=["b","c","d","f","g","h","a","e"]
    1 Solution.
```

3.9. Побудова експертної системи

Експертна система (ЕС) – це програма, що поводить себе подібно до експерта у деякій прикладній області. Типові застосування експертних систем полягають у використанні таких задач, як медичне діагностування, локалізація несправностей обладнання та інтерпретація результатів вимірів. Застосування для ЕС можна класифікувати як:

- *інтерпретацію* – побудова описів ситуацій за даними, які простежуються: розпізнавання, розуміння мови, аналіз зображень, визначення хімічної структури, інтерпретація сигналів, аналіз різного роду інформації;

- **прогноз** – виведення ймовірнісних наслідків з заданої ситуації, передбачення погоди, врожаю, прогнозування військових операцій;
- **діагностування** – висновок про порушення у системі на підставі спостережень, діагностування в медицині, в системах програмного забезпечення, в технічних пристроях;
- **проектування** – побудова конфігурації об'єктів при заданих обмеженнях, синтез можливих проектних рішень та їхній аналіз з погляду обмежень;
- **планування** – проектування плану дій: планування роботи, планування військових операцій;
- **моніторинг** – порівняння дійсної та бажаної поведінки експертної системи, запобігання небезпечним ситуаціям: у разі обслуговування атомних електростанцій, повітряного руху, фінансів, лікування захворювань;
- **системи навчання** – визначення рівня знань та навчання;
- **керування** – інтерпретація, прогноз, моніторинг, задачі, які не піддаються розв'язуванню традиційними математичними методами.

За допомогою ЕС вирішують задачі, що потребують для свого розв'язування експертних знань у деякій предметній області. Тому їх також називають системами, що ґрунтуються на знаннях. Крім того, ЕС пояснює користувачу свою поведінку так, як це робить людина-експерт. Досить часто до ЕС висувають додаткові вимоги – змугу мати справу з невизначеністю та неповнотою. У таких випадках необхідні міркування з використанням імовірнісного підходу.

Зазвичай для того, щоб побудувати ЕС, потрібно розробити механізми виконання таких функцій системи:

- розв'язування задач з використанням знань з конкретної предметної області, можливо водночас виникне необхідність мати справу з невизначеністю;
- взаємодія з користувачем разом з поясненням пояснення намірів та рішень системи під час та після закінчення процесу розв'язування задач.

Розробляючи ЕС, прийнято ділити її на три основні модулі (рис. 3.11.):

- (1) база знань;
- (2) машина логічного виведення;
- (3) інтерфейс користувача.

База знань містить знання, які відносять до конкретної прикладної галузі, в т. ч. окремі факти, правила, що описують відношення та явища, а також, можливо, методи, евристики та різні ідеї щодо задач у прикладній області. **Машина логічного виведення** вміє активно використовувати інформацію з бази знань. **Інтерфейс користувача** відповідає за безперервний обмін інформацією між користувачем та системою, надає користувачу змугу стежити за процесом розв'язування задач. Прийнято розглядати машину виведення та інтерфейс користувача як один крупний модуль, який називають **оболонкою експертної системи**.

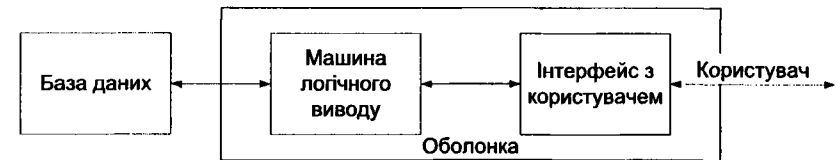


Рис. 3.11. Експертна система

Такий підхід дає змугу створити універсальну оболонку, після чого для кожного застосування достатньо підключити до системи нову базу знань.

Побудуємо невелику експертну систему, що буде вгадувати одну з семи тварин (якщо така можлива). Система задаватиме питання та будуватиме логічні висновки на підставі отриманих відповідей. Приклад наочно засвідчує пошук з поверненням, використання бази даних та ефективне застосування предиката not. Першим кроком у побудові ЕС є забезпечення її знаннями, потрібними для виконання міркувань. Механізм логічного виведення розглянемо на прикладі 3.9.

Приклад 3. 9.

```

facts
  xpositive (symbol, symbol)
  xnegative (symbol, symbol)
predicates
  animal_is (symbol)
  it_is (symbol)
  ask (symbol, symbol, symbol)
  remember (symbol, symbol, symbol)
  positive (symbol, symbol)
  negative (symbol, symbol)
  clear_facts
  run
clauses
  animal_is(cheetah):-it_is(mammal), it_is(carnivore),
    positive(has,tawny_color), positive(has,dark_spots).
  animal_is(tiger):-it_is(mammal), it_is(carnivore),
    positive(has, tawny_color), positive(has, black_strips).
  animal_is(giraffe):-it_is(ungulate), positive(has,long_neck),
    positive(has,long_legs), positive(has, dark_spots).
  animal_is(zebra):-it_is(ungulate), positive(has,black_strips).
  animal_is(ostrich):-it_is(bird), negative(does,fly),
    positive(has,long_neck), positive(has,long_legs),
    positive(has, black_and_white_color).
  animal_is(penguin):-it_is(bird), negative(does,fly),
    positive(does,swim), positive(has,black_and_white_color).
  animal_is(albatross):-it_is(bird), positive(does,fly_well).
  it_is(mammal):-positive(has,hair).
  it_is(mammal):-positive(does,give_milk).
  it_is(bird):-positive(has,feathers).
  it_is(bird):-positive(does,fly), positive(does,lays_eggs).
  it_is(carnivore):- positive(does,eat_meat).
  it_is(carnivore):- positive(has,pointed_teeth),
    positive(has, claws), positive(has,forward_eyes).
  it_is(ungulate):-it_is(mammal), positive(has,hooves).
  it_is(ungulate):-it_is(mammal), positive(does,chew_cud).
  positive(X,Y):-xpositive(X,Y), !.
  positive(X,Y):-not(xnegative(X,Y)), ask(X,Y,yes).
  negative(X,Y):-xnegative(X,Y), !.
  negative(X,Y):-not(xpositive(X,Y)), ask(X,Y,no).
  ask(X,Y,yes):-!,write(X," it ",Y,'\n'), readln(Reply),nl,

```

```

  frontchar(Reply,'y',_), remember(X,Y,yes).
  ask(X,Y,no):-!, write(X," it ",Y,'\n'), readln(Reply),nl,
    frontchar(Reply,'n',_), remember(X,Y,no).
  remember(X,Y,yes):-assertz(xpositive(X,Y)).
  remember(X,Y,no):-assertz(xnegative(X,Y)).
  clear_facts:-write("\n\nPlease press the space bar to exit\n\n"),
    retractall(_dbasedom),readchar(_).
  run:-animal_is(X),!,write("\nYour animal may be a (an) ",X),nl,
    clear_facts.
  run:-write("\nUnable to determine what"),
    write("your animal is.\n\n"),clear_facts.
goal
  run.

```

Кожну тварину описано за її ознаками. Відповіді на запитання користувача реалізовані через предикати `positive(X,Y)` та `negative(X,Y)`. Система може задати таке запитання:

Does it have hair? (В неї є шерсть?).

Отримавши відповідь, програмі необхідно мати змогу зберегти її в базі даних так, щоб далі використовувати цю відповідь у своїх міркуваннях. Для простоти у приклад включено тільки позитивні та негативні відповіді. Щоб їх використати у базі даних, вводять два предикати

```

facts
  xpositive (symbol, symbol)
  xnegative (symbol, symbol).

```

Факт, що тварина не має шерсті, буде записаний так:
`xnegative (has, hair).`

Правила `positive` та `negative` використовують для контролю відповідей користувача та задання нових питань.

```

positive(X,Y):-xpositive(X,Y), !.
positive(X,Y):-not(xnegative(X,Y)), ask(X,Y,yes).
negative(X,Y):-xnegative(X,Y), !.
negative(X,Y):-not(xpositive(X,Y)), ask(X,Y,no).

```

Треба відмітити, що друге правило як для `positive`, так і для `negative`, гарантує, що, задавши запитання, у користувача не має виникнути протиріч.

Предикат ask формулює питання та запам'ятовує відповіді. Якщо відповідь починається з букви у, тоді система припускає, що відповіддю є yes (так), а якщо з n – тоді no (ні).

```
ask(X,Y,yes):-!, write(X," it ",Y,'\n'), readln(Reply), nl,
  frontchar(Reply,'y',_), remember(X,Y,yes).
ask(X,Y,no):-!, write(X," it ",Y,'\n'), readln(Reply), nl,
  frontchar(Reply,'n',_), remember(X,Y,no).
remember(X,Y,yes):-assertz(xpositive(X,Y)).
remember(X,Y,no):-assertz(xnegative(X,Y)).
/*Знищення всіх фактів*/
clear_facts:-write("\n\nPlease press the space bar to exit\n"),
  retractall(_,dbasedom),readchar(_).
```

Нижче наведено результати спілкування програми з користувачем:

```
has it hair
у
does it eat_meat
у
has it tawny_color
у
has it dark_spots
n
has it pointed_teeth
n
does it give_milk
n
has it black_strips
у
Your animal may be a (an) tiger
Please press the space bar to exit.
```

3.10. Комп'ютерні проекти.

1. Задано множину пар вершин, які відповідають ребрам простого графа. Визначити, чи є граф зв'язним. Якщо виявиться, що граф не є зв'язним, то знайти кількість його компонентів.

2. Знайти всі вершини графа, для яких відомий шлях заданої довжини від виділеної вершини.

3. Задано матрицю суміжності графа і натуральне число r . Знайти кількість шляхів довжини r між двома вершинами. (Граф може бути як орієнтованим, так і неорієнтованим).

4. Задано матрицю суміжності мультиграфа. Вказати, чи має він ейлерів цикл, а якщо ні, то ейлерів шлях. Побудувати ейлерів цикл або шлях, якщо вони можливі.

5. Задано матрицю суміжності простого графа. Побудувати цикл найменшої довжини, який проходить через кожне ребро графа, принаймні, один раз.

6. Задано список ребер зваженого графа та дві його вершини. За алгоритмом Дейкстри знайти довжину найкоротшого шляху між заданими вершинами та побудувати цей шлях.

7. Задано простий граф. Знайти всі його мости. Застосувати алгоритм пошуку вглиб.

8. Задано зважений граф. Побудувати каркасне дерево мінімальної вартості.

9. Знайти діаметр зваженого графа, тобто максимальну відстань між можливими парами його вершин.

10. Задана система односторонніх доріг. Знайти шлях, який з'єднує міста A та B і не проходить через задану множину міст.

11. Відомо, що заданий граф не є деревом. Перевірити, чи можна вилучити з нього одну вершину (разом з суміжними їй ребрами), щоб в результаті з'явилося дерево.

12. Задано зважений неорієнтований граф з двома вершинами непарного степеня. Знайти цикл найменшої довжини, який проходить через кожне ребро графа, принаймні, один раз.

13. Заданий список ребер простого графа, який має не більше шести вершин. Побудувати гамільтонів цикл або виявити, що граф його не має.

14. Задача з розфарбування карти полягає у приписуванні кожній країні на карті одного з чотирьох заданих кольорів з таким розрахунком, щоб жодна пара сусідніх країн не виявилася пофарбованою в один колір. Розв'язати задачу методом пошуку вглиб.

15. Розв'язати задачу про переправлення через річку вовка, кози та капусти методом пошуку вишир. (Разом з людиною у човні

поміщається лише один об'єкт, а людині потрібно оберігати козу від вовка та капусту від кози.)

16. Розв'язати задачу комівояжера методом пошуку вглиб.

17. Розв'язати головоломку "Гра у вісім".

18. Розробити власну експертну систему на основі прикладу, описаного у розділі 3.9.

ПРЕДМЕТНИЙ ВКАЗІВНИК

А

Алгоритм уніфікації 23

Анонімні змінні 37

Аргументи 11, 12, 40, 47

Арність 12, 13, 36

Арифметичні операції 91

Атомарне висловлювання 12

Атомарне речення 13, 14

Б

База даних:

- зовнішня 75, 77

- ланцюгова структура 77

База знань 21, 135

База фактів:

- внутрішня 71, 72

Бінарні пошукові дерева 60

В

Виведення інформації 93

Висловлювання 7, 8, 9:

- атомарне 12

Відношення 4, 6, 9, 10, 12, 35

Відсікання 46

- зелене 47

- червоне 47

Г

Гамільтонів цикл 119, 120

Голова списку 63

Д

Дані:

- об'єкти складені 53

Дерева 43, 44, 45

- обхід 57

- пошук вглиб 128

- пошук вшир 131, 132

- цільове 43, 44, 45

- В+ дерево 75, 86

Детермінізм:

- відсікання 46

Диз'юнкція 6, 7, 13

Домени 40:

- зовнішніх баз даних 74

- складені 55, 56, 94

- стандартні 40, 78

Е

Експертна система 133

З

Заголовок правила 38, 44

Задачі:

- гамільтонів цикл 119, 120

- експертна система 133

- каркасне дерево 125, 129

- пошук вглиб 128

- пошук вшир 131

- пошук у лабіринті 117

- Ханойські вежі 112

- N ферзів 113
 Запити 39
 Заперечення 6, 7, 8, 13, 16
 Змінна 13, 16, 18, 37:
 - анонімна 37
 Значення істинності 6, 7, 8, 13, 16

I

Імплікація 6, 13, 14
 Інстанціювання 22
 Інтерпретація 7, 8, 15, 16, 19, 20
 Інтелект штучний 6, 26

K

Квантори 13:
 - всезагальності 13, 17, 22, 23
 - змінних 13
 Константи 10, 12
 Кон'юнкція 6, 7, 13, 22

L

Ланцюжки 75, 77, 78, 81
 Логічна програма 34

M

Машина логічного виведення 135
 Мови програмування 4, 5
 - LISP 5
 - PROLOG 5
 Модус поненс 21, 22, 23, 25
 Модус толленс 22

N

Невластиві символи 10
 Недетерміноване звернення 47
 Наявності квантор 13

O

Обхід дерева 57:
 - вглиб 58
 Об'єднання списків 68
 Об'єкти 36
 Оболонка експертної системи 135

Оголошення:

- предикатів 40
 - розділу фактів 40
 - складених доменів 53
 - списків 62
 Оператор
 - відсікання 46, 47, 48, 52, 53, 62

Операції

- арифметичні 91
 Оптимізована хвостова
 рекурсія 51

P

Повернення
 - точка повернення 46
 Пошук з поверненням 42, 46, 119
 Пошук:
 - вглиб 128
 - вшир 131, 132
 - з поверненням 26, 42, 43
 - всіх цілей 70
 Правила виведення 9, 19, 20, 22
 Правила 37, 38:
 - заголовок 38
 - тіло 38

Предикат 9, 12, 23, 36:
 - детермінований 47, 48
 - недетермінований 47
 - інфіксийний 54

Переривання пошуку з
 поверненням 47

P

Ребро 120, 123, 126
 Рекурсія 49
 - хвостова 51, 52
 Речення 6, 7, 12, 13, 18
 - атомарне 13
 Рівні інтелектуальних систем 3, 4:
 - абстракції 3, 35

- алгоритмів 4
 - знань 3, 4
 - мови 4, 5
 - символів 3, 4, 7
 - структур даних 4
 Розділ програми 40:
 - доменів 40
 - предикатів 40
 - речень 40
 - фактів 71
 - бази фактів 72, 73, 74
 - цілі 40

Рядки символів 103

C

Селектори 76, 88
 Символи 6, 10, 11, 12, 94
 - істинності 10, 11
 Список 26, 55, 62:
 - голова 63
 - хвіст 63
 - об'єднання 68
 - оголошення 63
 - пошук всіх цілей 70
 - кількість елементів 64
 Списки суміжностей 128, 131
 Список вершин та ребер 120
 Сколемізація 23
 Сортування на основі бінарного
 дерева 62
 Структури даних 4, 53:
 - складені об'єкти 53
 - рекурсивні 56
 - списки 62

T

Таблиця істинності 8
 Теорія першого порядку 13, 17, 34
 Терми 11, 81, 86
 Точка повернення 42, 45
 Тіло правила 38, 44, 45

Y

Уніфікатор 25, 26
 Уніфікація 23, 26, 28, 42

F

Файли 74, 98
 - закриття 100
 - відкриття 99
 Факторіал 50
 Факти 36, 37, 39, 40:
 - внутрішня база 71
 - додавання 72
 - розділ 67
 - збереження 74
 - вилучення 73
 Формула:
 - атомарна 12
 Функтори 55, 63
 Функціональний вираз 11

X

Ханойські вежі 112
 Хвостова рекурсія 51, 52, 65
 Хвіст списку 63

C

Цикл
 - гамільтонів 113
 Цілі 37, 39
 Цільове дерево 41, 42, 43

Ч

Числення:
 - висловлювань 6, 7, 8, 9
 - предикатів 6, 9, 10, 11, 13, 14

Ш

Шлях:
 - обходу дерева 54
 - гамільтонів 119, 120
 - найкоротший 123

SUBJECT INDEX

A

abs 92
 accessmode 78
 append 68
 arctan 92
 arity 11
 assert 72
 asserta 71
 assertz 72

B

B+ дерево 75, 86, 87
 - bt_copyselector 88
 - внутрішній вказівник 89
 - подвійні ключі 87
 - довжина ключа 88
 - закриття 88
 - оновлення 88
 - відкриття 87
 - створення 87
 - статистика 88
 - сторінки 87
 - видалення 88
 backtracking 26
 bt_close 88
 bt_copyselector 88
 bt_create 80, 87, 89
 bt_delete 88
 bt_open 87, 89
 bt_selector 78, 87
 bt_statistics 88
 byte 41

C

concat 105
 consult 74
 copyfile 102
 cos 92
 clauses 40
 closefile 100
 chain 75, 77
 chain_delete 76, 80, 81, 82
 chain_first 82
 chain_inserta 80, 81
 chain_insertafter 80, 82
 chain_insertz 80, 81
 chain_last 81, 82
 chain_next 81, 82
 chain_prev 81, 82
 chain_terms 81, 82
 char 41
 char_int 106
 cut 46

D

db_btrees 80
 db_chains 81
 db_close 79, 80
 db_copy 80, 81
 db_create 79
 db_delete 76, 79, 80
 db_flush 80
 db_open 79
 db_openinvalid 80
 db_selector 76, 78, 82
 db_statistics 81

deletefile 102
 denycode 78
 disk 102
 div 91, 92
 domains 40
 dword 41

E

eof 101
 existfile 102
 exp 92

F

facts 71, 73
 fail 26, 46
 file 76, 98, 99
 filemode 99
 file_str 97
 filepos 99
 findall 70, 71
 flush 76
 format 105
 frontchar 103
 frontstr 104, 105
 fronttoken 104, 107

G

goal 38, 39

I

improper symbol 10
 integer 41
 inference rule 20
 in_cms 79
 in_file 79
 in_memory 79
 isname 105, 107

K

key_current 89
 key_delete 88, 89
 key_first 88

key_insert 88
 key_last 88
 key_next 89
 key_prev 89
 key_search 88
 knowledge level 3

L

levels 3
 - symbol 3
 - knowledge 3
 LISP 5
 list 26
 ln 93
 log 92
 logical inference 19
 long 41

M

mod 91, 92
 modus ponens 9, 20, 21, 22
 modus tollens 22

N

nl 93
 not 48

O

objects 36
 occurs checks 24
 openappend 99
 open 102
 openfile 102
 openmodify 99
 openread 99
 openwrite 99

P

place 78
 predicates 40
 PROLOG 5, 34, 35
 proof procedure 21

R

random 92
 readchar 97
 readdevice 100
 readint 96
 readln 96
 readreal 96
 readterm 97
 real 41
 ref 78
 ref_term 82
 relations 36
 renamefile 102
 resolution 9
 retract 72
 retractall 73, 74
 round 92

S

save 74
 searchfile 102
 searchchar 105
 searchstring 106
 sin 92
 skolemization 23
 short 41
 sqrt 92
 stdin 99, 100
 stdout 99, 100
 str_int 106

str_len 105
 str_real 106
 string 41
 subchar 105
 substring 105
 symbol level 3

T

tail recursion 51
 tan 92
 term_delete 76, 80, 82
 term_replace 80, 82
 term_str 107
 trunc 92

U

ulong 41
 unify 26, 28
 upper_lower 107
 ushort 41

V

val 92
 variable quantifier 13

W

word 41
 write 93, 97
 writedevise 100
 writef 93, 94

СПИСОК ЛІТЕРАТУРИ

1. Адаменко А., Кучуков А. Логическое программирование и Visual Prolog. – СПб.: БХВ-Петербург, 2003.
2. Братко И. Программирование на языке Пролог для искусственного интеллекта. – М.: Мир, 1990.
3. Глибовец М.М., Олецький О.В. Штучний інтелект. – К.: Академія, 2002.
4. Клоксин У., Меллишин К. Программирование на языке Пролог. – М.: Мир, 1987.
5. Люггер Д. Искусственный интеллект. – СПб.: Вильямс, 2003.
6. Нікольський Ю.В., Пасічник В.В., Щербина Ю.М. Дискретна математика: Підручник. – Львів: Магнолія 2006, 2007.
7. Припула М., Щербина Ю. Алгоритми дискретної математики та обчислювальна складність: Навч. посібник – Львів: Видавничий центр ЛНУ ім. Івана Франка, 2002.
8. Стеллинг Л., Шатиро Э. Искусство программирования на языке Пролог. – М.: Мир, 1990.

ЗМІСТ

Вступ	3
1. Числення предикатів	6
1.1. Числення висловлювань.....	6
1.1.1. Символи та речення.....	6
1.1.2. Семантика числення висловлювань.....	7
1.2. Основи числення предикатів.....	9
1.2.1. Синтаксис предикатів та речень.....	10
1.2.2. Семантика числення предикатів.....	15
1.2.3. Значення семантики на прикладі "світ блоків".....	18
1.3. Правила виведення в теорії числення предикатів.....	19
1.3.1. Правила виведення.....	19
1.3.2. Уніфікація.....	23
1.3.3. Приклад уніфікації.....	28
1.4. Задачі.....	31
2. Основи мови Prolog	34
2.1. Факти і правила.....	36
2.1.1. Факти.....	36
2.1.2. Змінні.....	37
2.1.3. Правила.....	37
2.1.4. Запити.....	39
2.2. Основні розділи Visual Prolog-програм.....	40
2.3. Уніфікація та пошук з поверненням.....	42
2.4. Керування пошуком рішень.....	46
2.4.1. Використання відсікань.....	48
2.4.2. Предикат not.....	48
2.5. Рекурсія.....	49
2.6. Структури даних.....	53
2.6.1. Складені об'єкти даних.....	53
2.6.2. Деревя.....	56
2.6.3. Обхід дерева.....	57
2.6.4. Створення дерева.....	59
2.6.5. Бінарні пошукові дерева.....	60
2.6.6. Сортуння на основі бінарного дерева.....	62
2.6.7. Списки.....	62

2.7. Пошук всіх цілей відразу.....	70
2.8. Внутрішня база фактів.....	71
2.9. Зовнішні бази даних.....	74
2.9.1. Ланцюгова структура баз даних.....	77
2.9.2. Предикати для роботи з базами даних.....	79
2.9.3. В+ дерева.....	86
2.10. Арифметичні операції.....	91
2.11. Введення та виведення інформації.....	93
2.12. Файлова система.....	98
2.13. Робота з рядками символів.....	103
2.14. Комп'ютерні проекти.....	108
3. Приклади	112
3.1. Задача про Ханойські вежі.....	112
3.2. Задача про N ферзів.....	113
3.3. Пошук у лабіринті.....	117
3.4. Пошук гамільтонового циклу в графі.....	119
3.5. Пошук найкоротшого шляху.....	123
3.6. Побудова каркасного дерева зв'язного графу.....	125
3.7. Пошук вглиб.....	128
3.8. Пошук вшир.....	131
3.9. Побудова експертної системи.....	133
3.10. Комп'ютерні проекти.....	138
Предметний вказівник	141
Список літератури	147

Навчальне видання

ТРУШЕВСЬКИЙ Валерій Миколайович

**МОВИ ПРОГРАМУВАННЯ
ДЛЯ ШТУЧНОГО ІНТЕЛЕКТУ
ПРОГРАМУВАННЯ МОВОЮ PROLOG**

Навчальний посібник

*Рекомендовано
Міністерством освіти і науки України*

Редактор *Л. М. Макітринська*
Технічний редактор *С. З. Сенік*
Комп'ютерне верстання *Н. В. Буряк*

НБ ПНУС



754483

Формат 60x84/16. Ум. друк. арк. 8,7. Тираж 100 прим. Зам. №

Видавничий центр Львівського національного університету
імені Івана Франка. 79000 Львів, вул. Дорошенка, 41

Свідомство про внесення суб'єкта видавничої справи до Державного реєстру видавців,
виговівників і розповсюджувачів видавничої продукції:
серія ДК №3059 від 13.12.2007р.