В. І. Месюра, Н. В. Лисак, О. І. Суприган

ФУНКЦІОНАЛЬНЕ ТА ЛОГІЧНЕ ПРОГРАМУВАННЯ

Частина 1
Логічне програмування мовою Пролог

Міністерство освіти і науки, молоді та спорту України
Вінницький національний технічний університет

В. І. Месюра, Н. В. Лисак, О. І. Суприган

ФУНКЦІОНАЛЬНЕ ТА ЛОГІЧНЕ ПРОГРАМУВАННЯ

Частина1
Логічне програмування мовою Пролог
Лабораторний практикум

Вінниця
ВНТУ
2011

УДК 004.43.(075)
ББК 32.973-018.1я73
М53

Рекомендовано до друку Вченою радою Вінницького національного технічного університету Міністерства освіти і науки, молоді та спорту України (протокол №10 від 15.06.2009 р.)

Рецензенти:
В. П. Кожемяко, доктор технічних наук, професор
А. М. Петух, доктор технічних наук, професор
В. М. Лисогор, доктор технічних наук, професор

Месюра, В. І.

М53 Функціональне та логічне програмування. Частина 1. Логічне програмування мовою Пролог : лабораторний практикум / В. І. Месюра, Н. В. Лисак, О. І. Суприган - Вінниця : ВНТУ, 2011. - 106 с.

В практикумі розглянуті основні теоретичні відомості та практичні прийоми програмування мовою логічного програмування Турбо-Пролог (версії 2.0), приклади створення логічних програм, завдання для лабораторних робіт та рекомендована література. Лабораторний практикум розроблений відповідно до плану кафедри та програми дисципліни "Функціональне та логічне програмування".

УДК 004.43.(075)
ББК 32.973- 018.1я73

ЗМІСТ

ВСТУП

Бурхливий розвиток інформаційних технологій забезпечує створення все нових методів і засобів розв'язування складних логічних задач, що моделюють розумову діяльність людини, включаючи створення систем штучного інтелекту, асоціативних баз даних та знань, систем підтримки прийняття рішень на основі нечіткої логіки і т. ін. Одним з основних напрямків досліджень при цьому є розробка формалізованих моделей та потужних мов подання знань, що забезпечують достатньо виразні засоби роботи із ними.

Сучасні універсальні мови програмування (Сі, Java, Паскаль) безумовно здатні вирішувати усі ці проблеми, але їх використання у даній сфері потребує від програміста дуже високої кваліфікації та значних витрат часу. При цьому, безпосередня логічна сутність цих задач відходить на другий план, і основні зусилля витрачаються не на розв'язання задачі по-суті, а на часовитратне кодування наборами процедур, параметрів і викликів. Це ускладнює розуміння і відображення логіки задач, що розв'язуються, і значно зменшує оперативність її модифікації при зміні зовнішніх умов, а програміст зосереджується на трудомісткому маніпулюванні засобами конкретної універсальної мови програмування.

Тому дуже важливим є створення і розвиток таких засобів програмування, які дозволили би програмісту зосередитися саме на логічній та структурній сутності задачі і максимально звільнили би програміста від процедурного програмування дій комп'ютера з розв'язування задачі. Фактично йдеться про створення засобів підвищення рівня абстракції комп'ютера, коли задача формулюється в термінах конкретної проблемної області, а не в термінах апаратних можливостей комп'ютера або універсальної мови програмування. Це дозволяє якісно розширити коло задач і коло спеціалістів, здатних якісно розв'язувати вказані задачі у стислі терміни.

Саме поява логічного програмування, і безпосередньо мови Пролог (Програмування засобами Логіки), дозволили відобразити математичні підходи до розв'язання задач подання даних і зв'язків між ними за допомогою відношень (предикатів) і використання одноманітних процедур для здійснення на цих даних і зв'язках логічного виведення.

Творцями логічного програмування є Роберт Ковальський, який розробив процедурну інтерпретацію хорновських диз'юнктів, та Ален Колмероє, під керівництвом якого на початку 70-х років у Марсельському університеті (Франція) було створено спеціальну програму, написану на Фортрані, яка призначалася для доавтоматичного доведення теорем. Ця, названа Прологом програма, містила інтерпретатор Р. Ковальського.

Значний розвиток Пролог отримав у проектах, реалізованих у подальшому в Единбургському університеті [15].

Якщо традиційні мови програмування є процедурно-орієнтованими [14], то Пролог основано на декларативній (описувальній) парадигмі програмування. Це означає, що до комп'ютера треба ввести не закодований алгоритм розв'язання задачі, а лише формальний опис предметної області та задачі у вигляді аксіоматичної системи, а алгоритм розв'язку задачі здійснить «зашитий» у систему програмування інтерпретатор, що автоматично здійснює логічне виведення з опису задачі за певною стратегією. Отже, головною задачею програміста є не кодування алгоритму, а вдала аксіоматизація, тобто опис предметної області у вигляді системи логічних формул і такої множини відношень на неї, яка з достатнім ступенем повноти опише задачу.

Мова програмування Пролог базується на дуже обмеженому наборі механізмів, таких як, зіставлення зразків, деревоподібне подання структур даних і автоматичне повернення [1,3,11,12]. Пролог є надзвичайно корисним при моделюванні міркувань у задачах штучного інтелекту [2,3]. Зокрема, при розробці експертних систем [8], обробці природної мови [5,13], і т. ін., але дуже мало придатний в інших областях, наприклад, таких як графіка або числові алгоритми.

Останнім часом Пролог переживає бурний розквіт і отримав широке розповсюдження у найбільш розвинених країнах у таких галузях, які потребують розв'язування складних логічних задач управління, наприклад, літальними апаратами, електротехнічними системами, високотехнологічними виробництвами і т. ін. Разом із Ліспом він є головним інструментом створення сучасних систем штучного інтелекту, зокрема, експертних систем.

Реалізації Прологу існують для всіх сучасних програмних і апаратних платформ. Останніми роками він отримав значного розвитку в плані побудови серверних застосувань і програмування баз даних та Інтернету.

Даний навчальний посібник призначений для підготовки і виконання лабораторних робіт з дисципліни "Функціональне та логічне програмування" студентами напрямів підготовки 6.050101 «Комп'ютерні науки» та 6.050102 «Програмна інженерія».

Вміст посібника відповідає програмі дисципліни, розробленій на кафедрі інтелектуальних систем в межах загального циклу дисциплін поглибленої підготовки спеціалістів з програмування в області штучного інтелекту. Найбільшу увагу приділено розгляду таких важливих питань, як рекурсивні та ітераційні алгоритми, подання й обробка складних асоціативних структур даних, методи пошуку у просторах станів і методи логічного виведення.

1 ОСНОВИ ПРОГРАМУВАННЯ НА ТУРБО-ПРОЛОЗІ З ВИКОРИСТАННЯМ ФАКТІВ ТА ПРАВИЛ

1.1 Концептуальне моделювання предметної області

На відміну від процедурного програмування, де програміст повинен задати комп'ютеру покроковий алгоритм розв'язування конкретної задачі (тобто, спочатку знайти її розв'язок самостійно, в усякому випадку для окремих конкретних випадків), програмісту, що пише на Пролозі, треба зосередитися не на процедурі розв'язування задачі, а на створені формалізованого опису її предметної області. Отож першим етапом створення Пролог-програми є розробка концептуальної моделі (conceptual model), яка використовується для структурування предметної області (subject domain) задачі, забезпечуючи чітке виділення її окремих елементів: інформаційних об'єктів та зв'язків (відношень), що існують між ними. Саме тому, мова логічного програмування Пролог забезпечує широкі можливості щодо подання і використання знань про певну предметну область.

На рис. 1 наведений спрощений приклад подання предметної області «Футбольний клуб», серед інформаційних об'єктів якої можна виділити: футболістів, тренерів, суддів, лікарів, стадіон, уболівальників і т. ін. (рис.1).

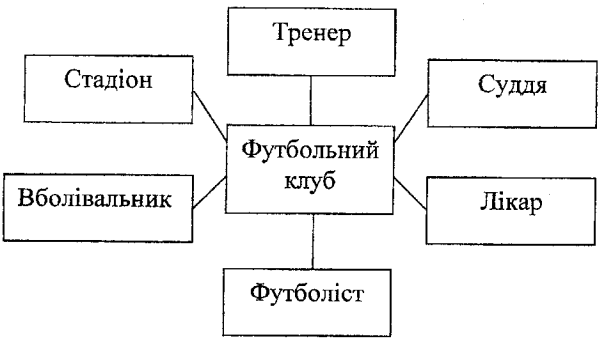

Рисунок 1 – Графічне подання фрагмента предметної області «Футбольний клуб»

Конкретний набір об'єктів визначається необхідним ступенем деталізації предметної області, потрібним для розв'язання задачі. Будемо розрізняти класи об'єктів (objects class) та екземпляри об'єктів (objects instance). Об'єкти визначають поіменованими характеристиками – властивостями, які мають назву атрибутів (attribute). Кожен клас об'єктів (рис.1) характеризується власною унікальною множиною властивостей, що

відрізняють його від інших класів об'єктів. Наприклад, клас об'єктів «Футболіст»» може характеризуватися атрибутами (властивостями) з назвами: «прізвище», «рік народження», «команда, за яку грає футболіст», «кількість забитих (воротарем – пропущених) м'ячів», «ріст», «найкращий результат виступів» і. т. ін. При цьому сам клас об'єктів «Футболіст» не характеризує жодного конкретного футболіста, а подає лише сукупність властивостей (атрибутів), притаманних кожному конкретному екземпляру даного класу об'єктів.

Назва кожного атрибуту одного класу об'єктів має бути унікальною, хоча вона може повторюватися для інших класів. Наприклад, назва «прізвище» може використовуватися лише для одного атрибута класу об'єктів «Футболіст», але вона може також використовуватися для одного з атрибутів будь-яких інших класів об'єктів, наприклад, таких як: «Тренер», «Суддя», «Лікар» і т. ін.

Поняття «екземпляр класу об'єктів» відноситься безпосередньо до конкретного об'єкта. Наприклад, екземплярами класу «Футболіст» можуть бути конкретні об'єкти «Луцюк» та «Бойко».

Розрізняють тип атрибута та екземпляр атрибута. Тип атрибута задає властивість об'єкта, у той час як екземпляри атрибута відображають різні значення цієї властивості.

Наприклад:

тип атрибута - дата народження.

екземпляри атрибута - 20.05.1981, 5.06.1988.

У подальшому, при використанні терміну «об'єкт», будемо розуміти клас об'єктів (його назву), а при використанні терміну «атрибут» - тип атрибута (назву атрибута).

Зазначимо, що абсолютної відмінності між об'єктом і атрибутом не існує. Те, що в одному контексті виступає як атрибут обраного об'єкта, в іншому контексті може виступати як самостійний об'єкт. Наприклад, для об'єкта «Футболіст» - «команда» є лише одним з атрибутів. У той же час для об'єкта «Національна першість» предметної області «Футбол», «Команда» буде одним з об'єктів, який буде мати власні атрибути.

Кожна властивість пов'язана з класом об'єктів певним відношенням (relation). Відношення, як і об'єкти, іменуються і можуть мати дуже різноманітні характеристики. При цьому екземпляр кожного окремого відношення подається лінією між тими екземплярами об'єктів, які він пов'язує. Атрибут (або набір атрибутів), що використовується для ідентифікації об'єкта (тобто, атрибут або набір атрибутів, значення якого є унікальним для кожного екземпляру об'єкта), називається ключовим атрибутом [7].

Кожен екземпляр відношення визначається набором ключових атрибутів (key attribute) об'єктів, які поєднуються цим відношенням.

Наприклад, відношення «грає_за» може пов'язувати об'єкт «Футболіст» з об'єктом «Футбольний клуб». При цьому екземпляри відношення «грає_за» будуть пов'язувати екземпляри об'єктів, наприклад, такі як «Бойко» - «динамо» та «Луцюк» - «шахтар».

Найбільш поширеною формою графічного подання відношень між класами об'єктів та екземплярами об'єктів є так звані ER-діаграми (entity-relationship diagram) класів та ER-діаграми екземплярів, спрощені приклади яких наведені на рис. 2 та рис. 3, відповідно. На цих діаграмах класи об'єктів позначаються прямокутниками, класи відношень – ромбами, екземпляри об'єктів –точками, а екземпляри зв'язків –лініями.

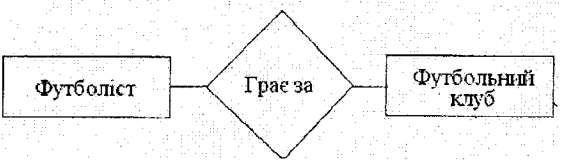

Рисунок 2 – Приклад діаграми класів

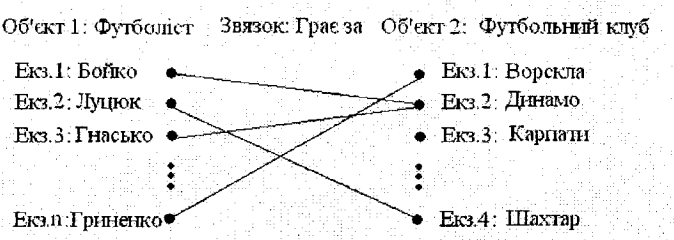

Рисунок 3 – Приклад діаграми екземплярів

Для подання будь-яких об'єктів та відношень предметної області у Пролозі використовується одна єдина структура – логічний терм (logical term). Терм - це або константа, або змінна, або функція.

Будь-який об'єкт та будь-яке відношення задачі, яку необхідно формалізувати, подаються термами одного або іншого вигляду.

Константи використовуються для подання конкретних значень екземпляра об'єкта. Константа подається символічним іменем (наприклад, бойко), або числом (наприклад, 32). Символічні імена констант називають атомами (atom). Атоми є нерозірваними ланцюжками букв і цифр, а також символів підкреслювання, які починаються з маленької букви. Вони можуть також подаватися у Пролозі рядками (string) – послідовностями будь-яких символів розміщених у подвійних лапках (наприклад "кількість м'ячів", "Прізвище").

Отже, атоми в Пролозі можуть бути, наприклад, такими:

луцюк; шахтар; гравець_10; “Луцюк”; “_воротар”; “рік народження”.

Атом також може складатися цілком зі спеціальних символів, до яких відносяться:

+ - * / ^ = : : ? @ $ &

Іншим видом константи є числова константа або просто число. Числа можуть бути цілими або дійсними.

Змінна у Пролозі використовується для позначення класу об’єктів або об’єкта, на який не можна послатися за іменем Змінна повинна починатися з великої літери або символу підкреслювання і вміщувати тільки символи літер, цифр та підкреслювання, наприклад:

X, Футболіст, Місце_проживання.

У випадках, коли ім’я змінної не є важливим, використовують так звану анонімну змінну, яка позначається знаком підкреслювання «_».

Відношення (зв’язки) у логічному програмуванні подаються логічними функціями, які називаються предикатами (predicate). Предикат відрізняється від звичайних функцій тільки тим, що може набувати лише двох значень - "істина" (true) або "помилка" чи "хибність" (false). Предикат (зв’язок, відношення, функція) записується у вигляді символьного імені, слідом за яким, в дужках, ідуть аргументи, кожен з яких є термом. Прикладами предикатів можуть бути такі:

грає_за (”Бойко”, “Динамо”). /*Бойко грає за Динамо*/;
грає_за (луцюк, шахтар). /*Луцюк грає за Шахтар*/;
грає_за (_, шахтар). /*Будь-хто, хто грає за Шахтар*/;
грає_за (X,Y). /* Будь-хто, хто грає за будь-яку команду*/;
має_вік (X, 28). /* Будь-хто, хто має вік 28 років*/.

Такі структуровані терми часто називають просто структурами. Структура складається з функтора, який подає ім’я відношення, та послідовності компонентів, які подають об’єкти у відношенні.

Кількість компонентів називається розмірністю (або арністю) структури.

1.2 Логічна програма

Логічна програма є множиною фактів (аксіом) та правил (rule), які виражають знання (knowledge) про задачу, яка розв’язується, внаслідок чого її іноді навіть називають базою даних (date-base) або базою знань (knowledge-base). Знання задаються відношеннями між об’єктами. Обчисленням логічної програми є виведення з неї наслідків (consequence).

Програма на Пролозі складається з множини речень, які називаються у Пролозі твердженнями (clauses). Кожне твердження закінчується крапкою і описує певне відношення, властивість, об’єкт або

закономірність. Множина речень може розглядатися як мережа відношень, що існують між термами, кожен з яких відображає деяку сутність відповідної предметної області.

Твердження має дуже просту структуру, яка, у загальному випадку, складається з голови (head) та тіла (tail):

<голова>:-<тіло>.

Голова твердження є предикатом, а тіло – логічним виразом. Тіло твердження складається з одного або більше предикатів, які поєднуються логічними операціями «і», «або» чи «ні», що позначаються, відповідно, символами «,», «;», «not». Символ «:-» відповідає слову «якщо». Таким чином, у загальному випадку, твердження є продукційним правилом вигляду «Якщо тіло – То голова»:

голова (Н)← тіло (Т),

і читається таким чином:

«Відношення, що є головою твердження, буде істинним,
якщо істинним буде логічний вираз, що є тілом твердження».

Сукупність тверджень і подає логічну програму, для запису якої у Пролозі відводиться спеціальний розділ – CLAUSES.

Усі твердження у Пролозі поділяються на факти, правила та запитання.

Факти відображають поточний стан предметної області. Вони містять конкретну інформацію і завжди мають значення «істина». Факти відповідають безумовним висловлюванням, тому твердження, що подає факт, містить лише голову (тіло твердження, яке описує умови істинності голови, у цьому випадку є пустим, оскільки факт є істиним за визначенням). Прикладами фактів для предметної області «Футбольний клуб» можуть бути такі:

```
футболіст("Бойко"),          /*Бойко є футболістом */;
команда (динамо),            /*Динамо є командою*/;
грає_за ("Бойко", "Динамо"), /*Бойко грає за Динамо*/;
має_вік (бойко, 28),         /*Вік Бойко складає 28 років*/.
```

Отже, факт є твердження про те, що у предметній області має місце певне конкретне відношення.

Нагадаємо, що згідно із синтаксисом Турбо-Пролога крапка вказує на завершення деякої порції інформації, що подається окремим твердженням. Між символами «/*» та «*/» у програмі розміщуються коментарі. Коментар також подається у вигляді рядка, що починається з символу «%».

Таким чином, факт завжди має структуру, наведену на рис. 4.

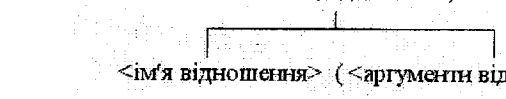

Рисунок 4 – Синтаксична структура предиката, що описує факт

Зауважимо важливість правильного порядку розташування аргументів у факті. Цей порядок задається автором і повинен виконуватись у всій програмі. Тобто, те, що вік Бойка складає 28 років може бути задане не тільки як *має_вік (бойко, 28)*, але й як *має_вік (28, бойко)*, але це будуть різні предикати.

З використанням відношень, можна описувати не тільки конкретні факти, але й правила (закономірності), які існують у проблемній області. Наприклад, на основі фактів:

нападаючий (бойко),
захисник (луцюк),
півзахисник (гнасько),
воротор (стеценко).

які свідчать про те, що різні гравці команди мають різні амплуа, можна визначити відношення «амплуа»:

амплуа (X) :- нападаючий (X); захисник (X);півзахисник (X); воротор (X).

Нагадаємо, що символ «;» позначає логічну операцію «або», отже наведене твердження можна прочитати так:

«X є амплуа (футболіста у даній предметній області),
якщо X – нападаючий, захисник, півзахисник або воротар».

Диз'юнкція (логічна операція «або») у тілі правила може задаватися й іншим чином, який є більш розповсюдженим у логічному програмуванні:

амплуа (X) :- нападаючий (X),
амплуа (X) :- захисник (X),
амплуа (X) :- півзахисник (X),
амплуа (X) :- воротор (X).

Даний фрагмент програми на Пролозі буде читатися, як:

«X є амплуа, якщо X – нападаючий або
X є амплуа, якщо X – захисник або
X є амплуа, якщо X – півзахисник або
X є амплуа, якщо X – воротар».

Чому така форма запису є більш зручною ми пояснимо при розгляді внутрішнього механізму розв'язування задачі, що існує у Пролозі.

Нагадаємо, шо правило є фактом (голова правила), істинність значення якого залежить від значення істинності інших фактів (тіло правила). Тобто, правило описує залежність деякого відношення від групи інших відношень (залежність значення істинності одного предиката від значення істинності інших предикатів), які називаються умовами (або цільовими твердженнями). Таким чином, правило відповідає умовному твердженню (імплікації [1,11,14]) і має структуру, наведену на рис. 5.

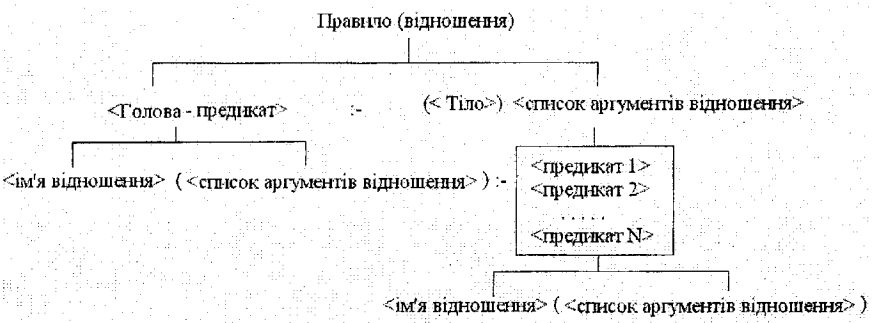

Рисунок 5 – Синтаксична структура предиката, що описує правило

Тіло правила, у більшості випадків, є кон'юнкцією предикатів, які називаються цільовими твердженнями (goal statement). Для доведення істинності голови правила необхідно послідовно довести можливість істинності кожного з цільових тверджень, шляхом зіставлення (comparison) їх з вмістом бази знань, яку подає логічна програма. Предикати (цілі) у тілі правила розділяються комами, які (як вже згадувалося) позначають логічну операцію кон'юнкції.

Правило є узагальненим твердженням, яке описує загальну закономірність, притаманну (у проблемній області задачі) не якомусь конкретному об'єкту, а цілому класу об'єктів. Тому до аргументів правила входять змінні. У загальному випадку, правило може розглядатися і як певний механізм редукції задачі, тобто зведення складної задачі (цілі) до множини більш простих підзадач (підцілей).

Наприклад, правило:

одноклубники (X,Y):- грає_за (X,Z), грає_за (Y,Z), X<>Y.

описує просту закономірність:

«X і Y є одноклубниками , якщо X грає за команду Z і Y грає за команду Z та X і Y не є однією і тією ж самою особою».

Отже, програма на Турбо-Пролозі має містити деякі факти і правила про певну предметну область. Наприклад, програма про національну

футбольну прем'єр лігу, може мати такий вигляд:

/*НАЦІОНАЛЬНА ФУТБОЛЬНА ПРЕМ'ЄР ЛІГА*/

```
DOMAINS                                  /*розділ опису типів даних*/
name, team = symbol.                     /*символьний тип даних */
old = integer.                           /*числовий тип даних */
PREDICATES                               /*розділ опису предикатів*/
команда(team).                           % вказується кожен з предикатів, що
футболіст(name).                         % будуть використані в програмі,
грає_за(name, team).                     % а саме, назва та типи даних кожного
має_вік(name,integer).                   з % аргументів кожного предиката
нападаючий(name).                        % програми
півзахисник(name).
захисник (name).
воротор(name).
одноклубники(name,name).
CLAUSES                                  /*розділ опису програми (тверджень
команда (динамо).                        бази знань) */
.....                                    /* опис фактів: */
команда (шахтар).                        % перелік команд предметної області
футболіст(бойко).                        % перелік футболістів прем'єр ліги
футболіст(луцюк).
.....
футболіст(дяченко).
грає_за (бойко,динамо).                  % за які команди грають футболісти
грає_за (луцюк, шахтар).
.....
грає_за (дяченко,динамо).                % опис віку футболістів
має_вік (луцюк, 23).
має_вік (дяченко, 19).
має_вік (бойко, 28).                     % опис амплуа футболістів
нападаючий (луцюк).                      % (Луцюк грає у двох різних %
півзахисник (луцюк).                     амплуа)
півзахисник (бойко).
захисник (дяченко)
амплуа (X) :- нападаючий (X).
амплуа (X) :- захисник (X).              /* опис правил: */
амплуа (X) :- півзахисник (X).           % правила визначення амплуа
амплуа (X) :- воротор (X).               % футболістів
одноклубники (X,Y):- грає_за (X,Z),
грає_за(Y,Z), X<>Y.                      % правило визначення одноклубників
```

Програма може описувати одне й те ж саме відношення кількома різними твердженнями, визначаючи таким чином різні його варіанти.Сукупність тверджень з однаковими головами (тобто, з однаковими іменами і кількістю аргументів), називається визначенням відношення або процедурою (procedure). Твердження одного визначення відношення мають групуватися разом. Всі твердження про предметну область утворюють програму, яку часто називають базою знань.

1.3 Запуск логічної програми на виконання

Ініціалізація логічної програми на виконання здійснюється введеням до неї запитання, відповідь на яке хоче отримати користувач. Запитання може бути введено з середовища Турбо-Прологу, або записано безпосередньо у програмі в спеціальному розділі GOAL. Запитання, або цільове твердження, за допомогою предиката описує відношення (гіпотезу), істинність якої для даного предметного середовища невідома і має бути визначена в процесі виконання програми, шляхом зіставлення запитання з твердженнями бази знань. Синтаксис запитання має збігатися з синтаксисом предиката.

Запитання є третім типом тверджень Пролога. Воно містить перелік певних умов, які цікавлять користувача, тобто подається консеквентом Т імплікації

$$H \leftarrow T.$$

Отже, запитання є твердженням, що не має голови і складається лише з тіла.

Введемо запитання до програми: «Хто з футболістів є одноклубниками?»:

GOAL

одноклубники (X, Y).

При отриманні запитання (цільового твердження), програма починає виконуватися. Сутність виконання полягає у пошуку в базі знань голови твердження, що відповідає запитанню (цільового твердження). Якщо таке твердження відсутнє, програма поверне відповідь «No». Якщо ж твердження присутнє, програма буде намагатися знайти такі значення змінних, при яких твердження стане істинним, і поверне ці значення як результат своєї роботи. Якщо ж таких значень знайти не вдасться, програма поверне відповідь «No».

Якщо цільове твердження складається з кількох підцілей (кон'юнкції предикатів), то Пролог по черзі, зліва направо, буде шукати такі значення змінних, при яких дані підцілі стануть істинними. Перехід до встановлення істинності наступної підцілі здійснюється лише після доведення істинності попередньої підцілі. Якщо довести істинність

попередньої підцілі не вдалося, пошук припиняється і програма повертає відповідь «No».

Таким чином, отримавши запитання

одноклубники (X,Y),

програма послідовно перегляне голови усіх тверджень з розділу CLAUSE, внаслідок чого знайде правило:

одноклубники (X,Y):-
грає_за (X,Z), грає_за (Y,Z),X<>Y.

Отже, для доведення істинності предиката *одноклубники(X,Y)* з голови правила, програмі необхідно буде знайти прізвища двох різних футболістів ($X \neq Y$), при підстановці яких на місця змінних X та Y, цей предикат стане істинним. Але він буде істинним тільки у тому випадку, якщо істинними будуть предикати *грає_за (X,Z), грає_за (Y,Z)* та *X<>Y*, розміщені у хвості правила. Таким чином, для доведенні істинності цільового твердження програма вимушена буде знайти у своїй базі знань не менш за два предикати *грає_за* з однаковими значеннями другого аргументу, та різними значеннями першого аргументу (у загальному випадку одноклубників може бути достатньо багато, і програма виведе усі їх пари, хоча у нашому прикладі міститься лише одна пара одноклубників – Бойко та Дяченко, які грають за команду «Динамо»).

Отож, отримавши на вході запитання *одноклубники (X,Y),* програма здійснить виконання таких дій:

1) пошук у розділі CLAUSE програми (базі знань) твердження, голова якого збігається з предикатом запитання:

одноклубники (X,Y).

Пошук здійснюється шляхом почергового зіставлення предикатних символів запитання і кожного твердження бази знань, в разі збігу яких здійснюється подальша перевірка запитання та твердження на однакову кількість та однакові типи їх аргументів. За результатами зіставлення виконуються такі дії:

- якщо у базі знань відсутні предикати з шуканою головою, програма поверне відповідь "No";
- якщо знайдене у базі знань твердження є фактом, наприклад,

одноклубники (петренко,гуцалюк),

прогама поверне відповідь: X= петренко, Y=Гуцалюк;

- якщо у базі знань існує правило, голова якого збігається із запитанням (як це є у нашому прикладі), програма розпочне процес визначення істинності кожного з предикатів (підцілей) хвоста правила, у порядку зліва направо;

У нашому прикладі, по черзі, переглядаючі усі твердження бази

знань, програма знайде правило:

одноклубники (X,Y):- грає_за (X,Z), грає_за(Y,Z), X<>Z,

структура голови якого повністю збігається зі структурою запитання.

2) оскільки знайденне твердження є правилом, програма розпочне процес доведення істинності першої з трьох його підцілей, для чого перейде до пошуку в базі знань твердження *грає_за (X,Z),* яке, у свою чергу, може бути як фактом (як це має місце у нашому прикладі), так і головою іншого правила (в цьому випадку програма вимушена була б перейти до перевірки істинності підцілей цього правила).

3) першим знайденим програмою шуканим твердженням стане факт:

грає_за (бойко,динамо),

аргументами якого є дві константи: перша - «бойко», та друга - «динамо». Програма здійснить зіставлення даного факта з підціллю

грає_за (X,Z),

аргументами якого є дві змінні: перша - «X», та друга - «Y». Зіставлення виявиться успішним (більш детально правила зіставлення будуть розглянуті пізніше) і за його результатами змінні набудуть значень:

X=бойко, Z=динамо.

4) програма перейде до доведення істинності другої підцілі, для чого знов з самого початку бази знань розпочне пошук твердження

грає_за (Y,динамо),

першим аргументом якого є змінна Y, а другим, внаслідок здійсненого на попередньому кроці означування змінної Z, константа «динамо».

5) першим знайденим програмою шуканим твердженням знов стане факт:

грає_за (бойко,динамо),

який успішно зіставиться з підціллю *грає_за(Y,динамо),* оскільки предикатні символи обох тверджень збігаються, другі аргументи мають константи з однаковими значеннями «динамо», а змінна Y набуде значення «бойко».

6) при доведенні третьої підцілі

X<>Z,

програма зазнає невдачі, оскільки у процесі доведення істинності підцілей змінні набули значень:

X = Y = «бойко».

7) оскільки заданий набір значень змінних не задовольняє умови істинності третьої підцілі (третього предиката хвоста) правила, Пролог здійснить спробу знайти інший предикат для зіставлення з другою підціллю

грає_за (Y,динамо).

Наступним у базі знань програми предикатом, предикатний символ і кількість аргументів якого збігаються з аналогічними параметрами другої підцілі, виявиться предикат

грає_за (луцюк, шахтар).

Але зіставлення цього предиката з другою підціллю буде невдалим, оскільки значення констант, що є другими аргументами двох тверджень, не збігаються:

«динамо» ≠ «шахтар».

8) внаслідок невдачі у доведенні істинності другої підцілі Пролог продовжить спроби знайти у базі знань інше твердження для зіставлення з нею, результатом чого стане виявлення факта:

грає_за (дяченко,динамо).

У даному випадку зіставлення буде успішним, оскільки предикатні символи обох тверджень збігаються, другі аргументи мають константи з однаковими значеннями «динамо», а змінна Y набуде значення «дяченко».

9) доведення третьої підцілі

«бойко» = X ≠ Y = «дяченко»,

також закінчиться вдало. Отож Прологу вдалося виявити у базі знань набір значень змінних, який задовольняє умови істинності усіх трьох підцілей цільового твердження, тобто виконання програми завершилося успіхом.

10) програма повертає отримані значення прізвищ одноклубників:

X = бойко, Y = дяченко.

Пролог автоматично зівставляє всі наявні в програмі факти з усіма правилами, за допомогою яких можно вивести нові факти, і таким шляхом сам визначає, що в наведеному правилі замість Y треба підставити «дяченко».

Якщо задати нашій програмі запитання

одноклубники (бойко, дяченко).

вона поверне відповідь "Yes".

1.4 Моделювання процесу доведення цільового твердження з використанням І/АБО-дерев

Програма на Пролозі являє собою базу знань. Її виконання ініціюється введеням запитання (запиту до бази знань), яке є одиничним цільовим твердженням, або кон'юнкцією цілей.

Виконання програми здійснюється внутрішнім інтерпретатором Прологу шляхом доведення цільових тверджень запиту за допомогою їх зіставленням з твердженнями бази знань. Зіставлення з фактом може привести до миттєвого доведення цільового твердження, оскільки факт завжди є істинним. Зіставлення з правилом лише зводить задачу (ціль) до сукупності більш простих підзадач, які описуються кон'юнкцією предикатів-підцілей. Зустрівши кон'юнкцію предикатів-підцілей (у запиті або у правилі) Пролог намагається узгодити їх з базою знань у тому порядку, в якому вони записані у запиті або хвості правила (зліва направо). Це значить, що жодне цільове твердження не буде оброблятися доти, доки не буде доведенна істинність його сусіда зліва. А сусід справа буде розглядатися лише після доведення поточного цільового твердження. Запит може містити змінні, а може і не містити їх. У першому випадку здійснюється пошук таких наборів значень змінних (тобто екземплярів об'єктів), для яких цільове твердження є істинним. У другому – розв'язується задача доведення (перевірки) істинності заданого твердження (Yes або No) з використанням фактів і правил бази знань.

Однією з основних відмінностей обчислювальної моделі логічного програмування від моделей традиційного програмування є недетермінізм (nondeterministic) [12]. В загальному випадку для кожного твердження бази знань логічної програми може існувати нескінченно багато варіантів визначення (правил з однаковими головами але різними тілами). Вибір «правильного» твердження здійснюється недетерміновано. Поняття недетермінованого вибору ефективно застосовується у визначенні багатьох моделей обчислень, зокрема у кінцевих автоматах та машинах Тьюрінга. Недетермінований вибір полягає у невизначеному виборі з деякої кількості альтернатив, який здійснюється методом «передбачення»: якщо тільки деяка з альтернатив приводить до успішного обчислення (у нашому випадку до знаходження доведення істинності цільового твердження), то саме її і буде вибрано. Формально поняття недетермінізму визначається таким чином: обчислення, яке містить недетермінований вибір за означенням, закінчується успішно, якщо існує послідовність недетермінованих виборів, що приводять до успіху. Зрозуміло, що жодна з реальних машин не може безпосередньо реалізувати це визначення. Але можна реалізувати ефективні наближення до даного поняття, що й зроблено у Пролозі за допомогою механізму пошуку в ширину з поверненнями.

Відображення недетермінованого вибору в ході виконання Пролог-програми зручно здійснювати з використанням формального апарату І/АБО-графів [3,15], а точніше - І/АБО-деревами (рис. 6). При цьому вершини І/АБО-дерев подають стани програми, а дуги – переходи (кроки) між можливими станами прогами. Головною особливістю І/ АБО-дерев (AND/OR-tree) є наявність у їх складі вершин та дуг двох видів: І та АБО.

Дуга (arc) І-типу (І-дуга) подає перехід від голови до тіла правила (від цільового твердження до його підцілей) і відображається гіпердугою, складові якої поєднуються між собою міткою у вигляді напівкруглої дужки.

Дуга АБО-типу (АБО-дуга) подає перехід від окремого твердження з тіла правила (підцілі запиту) до усіх можливих конкретних варіантів його визначення.

Вершину (vertex), з якої виходять І-дуги називають І-вершиною (AND-vertex), а вершину, з якої виходять тільке АБО-дуги – АБО вершиною (OR-vertex).

І-вершини зручно помічати іменами предикатних символів голів правил або запитань (цілей), які вони відображають, що, у загальному випадку, складаються з кон'юнкцій предикатів (підцілей).

АБО-вершини також зручно помічати іменами предикатних символів голів відповідних тверджень, з доданням порядкового номера варіанта визначення твердження згідно з його розташуванням в програмі.

Вершини розташовуються за рівнями І/АБО-дерева, які чередуються: один ярус складається з вершин лише типу І, а наступний – лише з вершин типу АБО, і навпаки.

Дерево будується зверху вниз. Запит до програми подається кореневою АБО чи І-вершиною в залежності від того, складається запитання, відповідно, з одного цільового твердження чи з кон'юнкції тверджень.

Після цього, у першому випадку, у базі знань здійснюється пошук визначень з іменами цільового твердження, і до кореневої вершини типу АБО приєднується така кількість вершин типу І, що відповідає кількості варіантів визначення тверджень відношення, яке подає запит. Вершинам приписуються імена визначення з номерами, що відповідають порядку розташування варіантів тверджень у програмі.

У другому випадку, до вершини типу І приєднується така кількість вершин типу АБО, скільки предикатів (підцілей) містить тіло правила (запит). Вершинам приписуються імена відповідних предикатних символів у тому самому порядку зліва направо, в якому вони розташовані в тілі правила (запити).

Процес повторюється доти, доки усі листові вершини не будуть відображати факти.

На рис. 6 наведено приклад І/АБО-дерева виконання фрагмента Пролог- програми:

CLAUSE

нападаючий (луцюк).
нападаючий (бойко).
грає_за (бойко,динамо).
грає_за (луцюк, шахтар).
місто (динамо,київ).
місто (шахтар,донецьк).
живе_у(X,Z):-
грає_за(X,Y),
місто(Y,Z).

GOAL

нападаючий(X), живе_у(X,Київ).

Відповідно до запиту, необхідно знайти будь-якого нападаючого, який живе у Києві.

На рис. 6 І-вершини подані чорними колами, АБО-вершини – білими колами. Вершини мають маркери, перший символ якого збігається з першим символом відповідного предикатного символу, а другий – визначає порядковий номер цього предиката в програмі.

Запит складається з двох підцілей, отож першою обробляється ціль

нападаючий (X).

У базі знань є два твердження з предикатом *нападаючий(X).* Аналіз починається з першого з них (н1) за порядком розташування предикатів у базі знань:

нападаючий (луцюк).

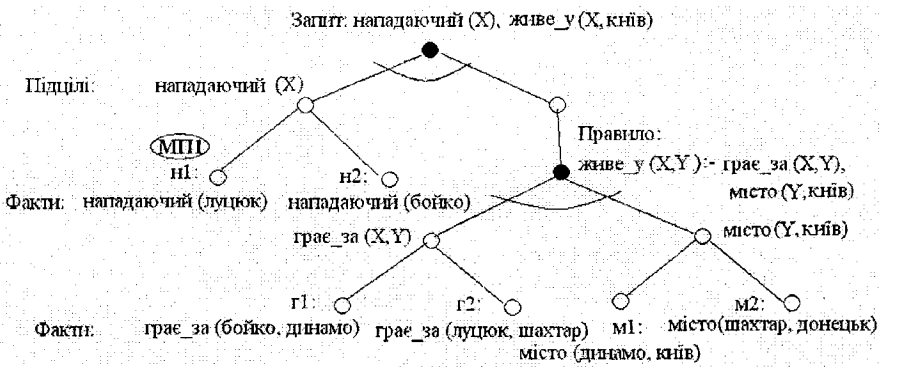

Рисунок 6 – Модель процесу виконання програми у вигляді І/АБО-дерева

Оскільки аргументом цілі є неконкретизована змінна X, а аргументом

факту є константа, уніфікація (unification) [1,7] відбувається успішно і змінна X набуває конкретного значення об'єкта «луцюк»:

X = луцюк.

Знайдене твердження відмічається маркером повернення МП1.

Знайдене твердження є фактом, отже, доведення цілі *нападаючий (X)* закінчується успішно і здійснюється перехід до другої підцілі запиту:

живе_у(X,київ).

Дана підціль є правилом і, в свою чергу, потребує доведення двох нових підцілей, першою з яких) є

грає_за(луцюк, Y).

(оскільки змінна X вже отримала значення «луцюк», а областю її дії є весь запит).

У базі знань є два твердження з предикатом *грає_за(X,Y).* Аналіз починається з першого з них (г1), згідно з порядком розташування предикатів у базі знань:

грає_за (бойко,динамо).

Оскільки константи «луцюк» та «бойко» не уніфікуються, здійснюється спроба уніфікації цілі з другим твердженням (г2)

грає_за (луцюк, шахтар).

У даному випадку уніфікація проходить успішно, оскільки предикатні символи і перші аргументи цілі і факту збігаються між собою, а другі аргументи цілі і факту є, відповідно, неконкретизованою змінною Y і константою «шахтар». При цьому змінна Y набуває значення «шахтар»:

Y = шахтар.

Оскільки усі твердження відношення *грає_за(X,Y)* вичерпалися, маркер повернення не встановлюється.

Знайдене твердження є фактом, отже, доведення цілі *грає_за(луцюк,Y)* закінчується успішно і здійснюється перехід до другої підцілі запиту, яка, на цю мить, у зв'язку з конкретизацією змінної Y=«шахтар», та заданим у запиті значенням змінної Z= «київ», набуває вигляду :

місто(шахтар,київ).

Спроби зіставлення даної цілі з відповідними фактами з бази даних виявляються невдалами, оскільки у першому з двох відповідних фактів (м1) з ціллю не збігається значення першого аргументу, а другого (м2) – значення другого аргументу.

Оскільки процес доведення цільового твердження (запиту) виявився невдалим (fail) (не вдалося знайти у базі знань такий набір значень змінних

X, Y, Z, який би задовольняв умови запиту), інтерпретатор Прологу здійснює спробу знайти інший шлях доведення цільового твердження. Для цього змінна Y звільняється від знайденого значення. Але, оскільки усі наявні у базі знань факти *грає_за(X,Y)* вже були переглянуті, і жодне з них не задовольняло цільові умови при знайденому раніше значенні X=«луцюк», розконкретизується і змінна X, а інтерпретатор починає пошук нового варіанта доведення з наступного за маркером повернення МП1 твердежння *нападаючий (X),* яким стає факт:

нападаючий (бойко).

Зіставлення з цільовим твердженням завершується успішно і X отримує значення

X = «бойко».

Перша підціль запиту задоволена і інтерпретатор переходить до другої підцілі, яка є правилом і, в свою чергу, потребує доведення двох цілей:

грає_за (бойко,Y) та *місто(Y,Київ).*

Перша ціль успішно узгоджується з фактом г1:

грає_за (бойко,динамо),

внаслідок чого змінна Y конкретизується значенням «динамо»

Y= «динамо».

На це твердження ставиться також маркер повернення МП2.

Далі інтерпретатор Пролога переходить до доведення другої підцілі, яка на цю мить набуває вигляду

місто(шахтар,київ).

Друга підціль узгоджується, оскільки у базі знань знаходиться саме такий факт м1.

Отже, інтерпретатору вдалося підібрати набір змінних

X = бойко, Y= динамо, Z= київ,

на якому цільове твердження стає істинним.

Пролог зробить спробу знайти ще й інші рішення, намагаючись зіставити другу підціль з ще не випробуваним в даному циклі доведення фактом м2, а потім з фактом р2. Але ці спроби будть невдалими і Пролог поверне результати пошуку і повідомлення 1 Solution, інформуючи про завершення роботи програми:

X = Бойко.
Y= динамо.
Z= київ.
1 Solution.

1.5 Алгоритм роботи внутрішнього інтерпретатора Прологу

Виконання Пролог-програми можна подати як покроковий процес обробки цільових тверджень, кожен крок якого ділиться на 4 фази:

1) зіставлення зі зразком;
2) уніфікація аргументів;
3) перевірка типу твердження: факт або правило;
4) процес повернення (backtracking).

Проілюструємо виконання кожної фази за допомогою І/АБО-дерева.

1.5.1 Зіставлення зі зразком

Виконання Пролог-програми здійснюється внутрішнім інтерпретатором Прологу шляхом доведення істинності цільових тверджень запиту або пошуку таких наборів значень змінних цільового твердження, на яких воно стане істинним. Доведення істинності цільових тверджень здійснюється на основі їх зіставленням з твердженнями, які утворюють базу знань програми. Цільове твердження, як і будь-який інший предикат, повністю визначається своїм іменем (предикатним символом) і кортежем (кількістю та порядком розташування) аргументів, які й утворюють зразок (pattern) для зіставлення.

Зіставлення з фактом (fact recognition) може привести до миттєвого доведення цільового твердження, оскільки факт завжди є істинним. Зіставлення з правилом лише зводить задачу (ціль) до сукупності більш простих підзадач, які описуються кон'юнкцією предикатів-підцілей. Зустрівши кон'юнкцію предикатів-підцілей (у запиті або у правилі) Пролог намагається узгодити їх з базою знань у тому порядку, в якому вони записані у запиті або хвості правила (зліва направо). Це значить, що жодне цільове твердження не буде оброблятися доти, доки не буде доведенна істинність його сусіда зліва. А сусід справа буде розглядатися лише після доведення поточного цільового твердження. Запит може містити змінні, а може і не містити їх. У першому випадку здійснюється пошук таких наборів значень змінних (тобто екземплярів об'єктів), для яких цільове твердження є істинним. У другому – розв'язується задача доведення (перевірки) істинності заданого твердження ("Yes" або "No") з використанням фактів і правил бази знань.

Якщо у базі знань відсутнє відношення, яке може бути зіставлене зі зразком, то програма повертає результат "No" і завершує своє виконання. Якщо у базі виявляється твердження із заданими предикатним символом та кількістю аргументів, то здійснюється перехід до другої фази – уніфікації аргументів.

1.5.2 Уніфікація аргументів

У процесі уніфікації здійснюється послідовний перегляд тверджень, знайдених у першій фазі, у тому порядку, в якому вони розташовані в базі знань. При цьому здійснюється послідовне зіставлення аргументів

цільового твердження (предиката) з аргументами голови досліджуваного твердження у порядку розташування термів у кортежі.

Нехай програма має такий набір фактів:

уболіває (андрій,динамо).
уболіває (марко,карпати).
уболіває (марко,фк_львів).
уболіває (марко,динамо).

а цільовим є твердження:

уболіває (марко,динамо).

Турбо-Пролог починає переглядати програму зверху вниз, намагаючись знайти факт, який зіставиться з цільовим твердженням.

Предикатні символи всіх фактів будуть однакові, але в першому твердженні не зіставиться перший аргумент, а в другому та третьому - другі аргументи. Нарешті, четверте твердження буде успішно зіставлено, ціль успішно доведена, а програма поверне значення "Yes". Таким чином, механізм внутрішньої уніфікації Пролога обробляє всі факти та правила бази знань (у порядку зверху вниз, зліва направо), які мають відповідний предикатний символ (functor). При виявлені потрібного предикатного символу, аргументи тверджень зіставляються зліва направо, доки це зіставлення не виявиться успішним або неуспішним. У випадку неуспішного зіставлення, здійснюється перевірка наступних фактів або правил доти, доки одне із зіставлень не закінчиться успішно, або доки всі релевантні факти та правила не будуть випробувані та не виявляться неуспішними.

Зауважимо, що у випадку зіставлення двох констант результат буде успішним тільки при їх збігу.

Задамо тепер системі запитання: «За яку команду вболіває Марко?»:

уболіває (марко,X).

У даному цільовому твердженні X є змінною (нагадаємо, що змінні завжди записуються з великої літери). У початковий момент часу змінна X не має ні якого значення, оскільки вона неозначена або неконкретизована. Неконкретизована змінна називається також вільною. При зіставленні неконкретизованої змінної X з об'єктом «динамо», вона перестає бути вільною і набуває значеня X = «динамо». Кожного разу, при зіставленні неозначеної змінної з константою змінна отримує значення цієї константи.

У прикладі, що розглядається, цільове твердження може бути успішно зіставлене з трьома фактами. Внутрішні уніфікаційні підпрограми Пролога знайдуть усі три рішення, які задовольняють цілі. При цьому змінна X буде послідовно отримувати значення «карпати», «фк_львів» та «динамо», а програма поверне такий результат:

X1 = карпати.
X2 = фк_львів.
X3 = динамо.

Змінна буде звільнюватися кожного разу, коли зіставлення виявиться неуспішним або ціль виявиться успішно обчисленою.

Процес означування змінної можна проілюструвати також на прикладі оператора « = », який може використовуватись у Турбо-Пролозі як оператор порівнювання або як оператор присвоювання, в залежності від того, чи є значення термів вільними або означеними.

Якщо обидва терми означені, то оператор = є оператором порівнювання, і повертає значення «істина» при збігу термів або значення «помилка» в протилежному випадку.

Якщо відомо значення тільки одного терма, то воно буде присвоєно іншому терму, при цьому неважливо, який з термів знаходиться зліва, а який справа від оператора « = ».

Якщо обидва терми є змінними (variable), то відбувається їх «зчеплення», тобто в подальшому вони сприймаються уніфікаційними підпрограмами як одна змінна.

Розглянемо детальніше поняття анонімної змінної, яка позначається символом підкреслювання «_», використовується у разі відсутності необхідності знання конкретного значення змінної, і не здійснює виведення значення на екран. Вона не потребує імені, яке не відіграє ніякої ролі в процедурі. Твердження може містити будь-яку кількість однаково позначених (символом підкреслювання) анонімних змінних, але всі вони будуть окремими змінними, оскількі кожного разу при зустрічі анонімної змінної Пролог надає її окремий індекс (_1, _2, _3 і т. ін.). Анонімні змінні використовують в запитах у тих випадках, коли значення, що підставляються на їх місце, не мають значення.

Наприклад, якщо ми схочемо узнати, чи є Марко футбольним уболівальником, та на запит:

уболівальник (марко, Y).

для останнього прикладу отримаємо таку надлишкову відповідь :

Y= карпати; Y=фк_львів;Y=динамо.

Відповіддю ж на запит

уболівальник (марко, _).

буде саме те, що потрібне, одне слово

Yes.

Сформулюємо основні правила уніфікації термів у Турбо-Пролозі:

• константи або конкретизовані змінні уніфікуються тільки якщо вони однакові;

• константа або конкретизована змінна уніфікується з неконкретизованою змінною, при цьому неконкретизована змінна отримує значення відповідної константи або конкретизованої змінної;

• структури уніфікуються тільки якщо збігаються їх функтори (предикатні символи), арність (кількість аргументів) та типи відповідних аргументів (компонентів);

• неконкретизовані змінні уніфікуються (зчеплюються) між собою і набувають одне й те ж саме неконкретизоване значення (стають однією і тією ж самою змінною);

• анонімна змінна уніфікується з чим завгодно, приписуючи програмі ігнорувати значення аргумента.

У разі успішного узгодження аргументів знайдене у означенні предиката твердження бази знань помічається маркером і відбувається перехід до наступної, третьої фази. Кожна ціль має свій маркер. Якщо узгодження аргументів цілі і поточного твердження бази знань закінчується невдачею, то здійснюється перехід до наступного за порядком розташування в програмі твердження у означенні. Це продовжується доти, доки не буде досягнено успіху, або доки в базі знань не залишиться жодного не переглянутого твердження із заданим іменем. Останній випадок свідчить про хибність цільового твердження, тому здійснюється спроба знайти інший шлях доведення його істинності, для чого вмикається механізм повернення (перехід до фази 4).

1.5.3 Перевірка типу твердження: факт або правило

У третій фазі з'ясовується, чи є досліджуване твердження фактом, чи правилом.

Якщо твердження не містить символу « :- », то воно є фактом. Оскільки факт завжди є істинним, то доведення поточної цілі завершується успішно. При цьому здійснюється перехід до наступної цілі, і для неї весь процес повторюється з першої фази. В І/АБО-дереві знайдений факт помічається маркером М1 і здійснюється перехід зліва направо по вершинах поточного рівня І/АБО-дерева.

Якщо зіставлення відбувалося не з фактом, а з головою правила, то починається процедура «розкриття» правила, тобто, послідовного доведення кожного цільового твердження з тіла правила зліва направо, тому поточною ціллю стає перший недосліджуваний предикат у тілі правила.

1.5.4 Механізм повернення

Механізм повернення (backtracking) використовується Прологом для знаходження додаткових фактів та правил необхідних при обчисленні цілі у випадках, коли поточна спроба обчислити ціль виявиться

неуспішною. Пошук здійснюється за І/АБО-деревом відповідно до алгоритму пошуку на графах в глибину.

Процес повернення полягає у перегляді результатів раніше виконаної інтерпретатором роботи (прегляді набору отриманих змінними значень) і спробах передовести (по іншому узгодити) цільові твердження з твердженнями бази знань за рахунок пошуку альтернативних шляхів доведення. При цьому здійснюється повернення до попередньої цілі, що знаходиться зліва від тієї, що виявилася невдалою, і ця попередня ціль знов стає поточною. Відповідно усі змінні, які на попередньому кроці доведення отримали значення конкретних об'єктів, звільняються (стають вільними). Обробка попередньої цілі починається не від самого початку бази знань, а продовжується від раніше досягненого інтерпретатором рубіжу: від твердження, поміченої маркером цілі. Для цього береться наступне після відміченого маркером твердження у визначенні відношення (наступний варіант), і починається процедура уніфікації аргументів його голови з аргументами цілі.

При знаходженні у означенні нового твердження, відповідного цільовому, воно буде помічене, поточна ціль вважатиметься доведеною, і здійснитиметься перехід до наступної цілі. Якщо інше відповідне твердження у визначенні відсутнє, поточна ціль вважатиметься недоказовою і повернення до передоведення раніше пройдених цілей продовжиться. При першому проході доведення розгляд тверджень завжди починається з самого початку бази знань, а при поверненні, з твердження бази знань, яке помічене маркером.

Проілюструємо механізми повернення та внутрішньої уніфікації на прикладах про улюблені команди футбольних фанів Марка та Андрійка:

	уболіває (марко,динамо).	/*у1*/
II	*уболіває (марко,будівельник).*	/*у2*/
IV	*уболіває (марко,карпати).*	/*у3*/
	уболіває (андрійко,Х):-	/*у4*/
	уболіває (марко,Х),	/*у6*/
	команда (Х),	/*к3*/
	місто (Х, Львів).	/*м4*/
I	*уболіває (андрійко,Х):-*	/*у5*/
	уболіває (марко,Х),	/*у7*/
	Х= будівельник.	/*б1*/
	команда (динамо).	/*к1*/
III	*команда (карпати).*	/*к2*/
	місто (динамо,київ).	/*м1*/
	місто (шахтар,донецьк).	/*м2*/
	місто(карпати,львів).	/*м3*/

В коментарях до кожного рядка вказані коди, які ми присвоюємо вершинам І/АБО-дерева. Перша позиція коду збігається з першою буквою голови відповідного твердження, а друга позиція – з порядковим номером цього варіанта визначення твердження у програмі в порядку зверху вниз.

Задамо цільове твердження: "За які команди уболіває Андрійко"?:

уболіває (андрійко,Х).

Пролог розпочне роботу з обробки цілі

уболіває (андрійко,Х).

та пошуку в програмі тверджень, що мають предикатний символ «уболіває», який збігається з предикатним символом цільової умови. Першими з них в програмі розташовані три факти у1-у3 про те, за які команди уболіває Марко. Зіставлення їх з цільовим твердженням не буде успішним, оскільки константа «андрійко» не зіставиться з константою «марко». Тоді буде здійснено спробу зіставлення цільової умови з головою правила у4. Оскільки змінні в голові правила та в цілі неконкретизовані, то голова правила і ціль зіставляться.

При цьому факти у6, к3, м4 тіла правила у4 стануть підцілями, а голова правила у4 буде помічена маркером повернення м1.

Доведення нової підцілі розпочнеться з самого початку бази знань. Тому, при зіставленні першої підцілі у6

уболіває (марко,Х),

з фактом у1

уболіває (марко,динамо),

змінна Х конкретизується значенням об'єкта «динамо», а сам факт у1 буде помічений маркером м2. Якщо тепер наступна підціль виявиться невдалою, механізм повернення буде мати точку для пошуку іншого кандидата на обчислення (доведення істинності) підцілі.

Наступною підціллю, істинність якої треба довести, є підціль к3:

команда (Х).

Оскільки в даний момент змінна Х має значення «динамо», то зіставлення здійснюється успішно. Але оскільки існує ще один факт (к2), предикатний символ якого збігається з предикатним символом поточної підцілі, то на к1 встановлюється маркер повернення м3. Остання відмічена точка завжди є точкою, з якої буде продовжено пошук альтернативного рішення.

Останньою підціллю правила у4 є твердження м4

місто (Х, львів),

яке, з урахуванням поточного значення змінної Х, перетворюється до

вигляду

місто (динамо, львів).

Але серед фактів бази даних м1-м3 відсутній такий факт, який можна було б зіставити з даною підціллю і всі спроби обчислити цю підціль (тобто довести її істинність) терплять поразку. Наслідком невдачі стає повернення до останнього показчика м3, який встановлений на факті к1

команда (динамо).

Зіставлення з наступним фактом к2

команда (карпати).

також виявляється неуспішним, що викликає повернення до покажчика м2, встановленого на факті

уболіває (марко,динамо).

У цей момент змінна Х знову стає вільною в зв'язку з неуспіхом спроби обчислення підцілі шляхом зіставлення Х із значенням «динамо». У наступному за маркером м2 твердженні Пролог знаходить факт у2

уболіває (марко,будівельник),

на який встановлюється маркер повернення м4, та у наслідок зіставлення з яким цільового предиката відбувається конкретизація змінної Х значенням об'єкта «будівельник».

Підціль

команда (будівельник),

не може бути доведена за допомогою фактів і правил, що зберігаються у базі знань. В результаті змінна Х знову звільняється, а внутрішні уніфікаційні підпрограми виконують повернення до твердження, наступного за тим, на якому встановлено показчик (pointer) м4:

уболіває (марко,карапати).

Тепер підціль

команда (Х),

зіставляється з фактом

уболіває (марко,карпати),

а змінна Х набуває значення "карпати". Після невдалої спроби зіставлення підцілі з фактом бази знань к1 *команда (динамо),* підціль, тим не менш, успішно зіставляється з фактом к2

команда (карпати).

Отже, залишається зіставити останню підціль правила у4 з фактами про «місто» (м1-м3). І таке зіставлення, потерпівши дві невдачі на

твердженнях м1 і м2, виявляється успішним для факта м3:

місто(карпати, львів).

З успішним зіставленням останньої підцілі правило доведено.

Змінна Х, набувши значення «Карпати», тим самим довела істинність тіла правила, внаслідок чого Пролог повертає знайдене значення

Х1= карпати.

Але Пролог продовжує пошук інших варіантів рішень. Оскільки ціль була успішною, то змінна Х звільняється і може знов бути конкретизована підпрограмами внутрішньої уніфікації.

Пошук розв'зання продовжується з маркера повернення м1, який є тепер останнім (маркер повернення м2 знищується, оскільки наприкінці шляху було знайдено розв'язок задачі).

Тепер пошук починається з правила у5. Знов здійснюється зіставлення першої підцілі у7 з групою фактів у1-у3.

Твердження у1

уболіває (марко,динамо),

зіставляється з підціллю

уболіває (марко,Х),

змінна Х набуває значення «динамо», а на твердеження у1 встановлюється маркер повернення.

Спроба обчислити другу підціль б1

Х = будівельник.

при значенні

Х = динамо,

закінчується невдачою, оскільки результат операції порівняння

динамо = будівельник.

є хибним.

Змінна Х звільняється і здійснюється повернення до твердження у1. Маркер повернення встановлюється на у2, а змінна Х набуває значення «будівельник». Зіставлення з останньою підцілью (б1) проходить успішно (будівельник = будівельник) і Пролог повертає значення

Х2 = кукурудза.

Тепер знайдені два рішення, а маркер повернення встановлений на твердження у2. Оскільки існує ще одно твердження (у3) Пролог повертається до нього, встановлює на ньому маркер повернення та намагається знайти ще один розв'язок. Змінна Х знову звільнена, вона отримує значення «карпати» і перша підціль виявляється успішною.

Звернувшись до наступної підцілі Пролог порівнює

карпати = будівельник,

внаслідок чого підціль зазнає невдачу.

Знову виконується повернення, тепер до голови правила у4

уболіває (андрійко,X).

Ця голова незіставима з підцілью

уболіває (марко,X),

тому уніфікаційний механізм перевіряє можливість зіставлення з головою наступного правила у5. Знову зіставлення виявляється невдалим. Тобто, на цей раз правило не змогло дати розв'язок і ціль виявилась неуспішною. Раніше були знайдені два рішення. Щоб показати, що процес пошуку рішення завершений, Пролог повертає повідомлення

Two Solution.

1.6 Реалізація операцій реляційної алгебри засобами Прологу

Мови логічного програмування можна розглядати як потужне розширення моделі реляційної бази даних (знань) [6,15]. Факти програми утворюють відношення бази знань, а правила реалізовують запити.

Розглянемо, як реалізуються у логічній програмі операції реляційної алгебри.

Операція об'єднання (UNION) будує одне n-арне відношення R з двох n-арних відношень A і B і позначається таким чином:

$$R=A \cup B.$$

Вона задається логічною програмою, що складається з двох правил:

a_union_b $(X_1, X_2,...X_n)$*:- a* $(X_1, X_2,...X_n)$*.*
a_union_b $(X_1, X_2,...X_n)$*:- b* $(X_1, X_2,...X_n)$*.*

Наприклад:

амплуа (X) :- нападаючий (X).
амплуа (X) :- півзахисник (X).

Зауважимо, що об'єднання відношень дозволяє здійснювати узагальнення знань про предметну область.

Операція симетричної різниці (DIFFERENCE):

$$R = A \oplus B = (A \cap \neg B) \cup (\neg A \cap B),$$

реалізовується двома такими правилами:

a_diff_b $(X_1, X_2,...X_n)$*:- a* $(X_1, X_2,...X_n)$*, not b* $(X_1, X_2,...X_n)$*.*
a_diff_b $(X_1, X_2,...X_n)$*:- b* $(X_1, X_2,...X_n)$*, not a* $(X_1, X_2,...X_n)$*.*

Наприклад:

футболіст_або_волейболіст (X):- футболіст (X), not волейболіст (X).
футболіст_або_волейболіст (X):- волейболіст (X), not футболіст (X).

Операція декартова добутка (CARTESIAN PRODUCT):

$$R=A\times B,$$

може бути реалізована таким правилом:

$$a\times b\ (X_1, X_2,...X_n, Y_1, Y_2,...Y_m):-\ a\ (X_1, X_2,...X_n),\ b\ (Y_1, Y_2,...Y_m).$$

Наприклад:

круговий турнір (X,Y):-команда(X), команда (Y).

Операція проекції (PROJECT) забезпечує побудову відношення, яке містить вибірку значень окремих атрибутів вихідного відношення, що входять до заданого списку A, і з якого вилучені рядки, що повторюються:

$$R=R[A].$$

Така операція може бути реалізована за допомогою правила:

$$r135(X_1, X_3, X_5):-r(X_1, X_2, X_3, X_4, X_5).$$

Наприклад:

футболіст (Прізвище, Ім'я, Вік, Зріст, Вага, Амплуа, Команда, Адреса, Телефон, E-mail).
контакт_футболіста:- футболіст (Прізвище, Ім'я, _ _ _ _ _ Адреса, Телефон, E-mail).

Операції вибірки (SELECT) забезпечує отримання відношення з тієї ж самої схеми, як і вихідне, яке містить лише підмножину тих кортежів, що задовольняють задану умову:

$$R=R[A\theta v].$$

Операція реалізовується з використанням правила:

$$rv(X_1, X_3):-r(X_1, X_2, X_3),\ X_2.> X_1.$$

Наприклад:

футболіст (Прізвище, Ім'я, Вік, Зріст, Вага, Адреса, Команда, Амплуа, Телефон, E-mail).
Нападник (Прізвище, Ім'я) :- футболіст (Прізвище, Ім'я, _ _ _ _ _ _ _ _ _), Амплуа=нападник..

Операція з'єднання відношень (JOIN) B і C схожа на декартовий добуток, але з додатковим обмеженням на входження до результуючого

відношення R лише тих рядків, що задовольняють певне співвідношення між атрибутами з'єднання $[A_1,A_2]$ відношень B і C:

$$R = B[A_1 \theta A_2]C .$$

Наприклад:

футболіст (Прізвище, Ім'я, Вік, Зріст, Вага, Адреса, Команда, Амплуа, Телефон, E-mail).

тренер (ПрізвищеТ, Ім'яТ, ВікТ, АдресаТ, КомандаТ, Амплуа, ТелефонТ, E-mailT).

тренер_футболіста (Прізвище, Ім'я, ПрізвищеТ, Ім'яТ):-

футболіст (Прізвище, Ім'я, _, _, _, _, _, _, _, _), тренер (ПрізвищеТ, Ім'яТ, _, _, _, _, _, _).

1.7 Структура програми на Турбо-Пролозі

Синтаксис та структура програм на Турбо-Пролозі відображають концепції логіки предикатів. В цілях спрощення організації фактів та правил Турбо-Пролог підтримує складові доменні структури, цеглинками для створення яких служать базисні типи доменів Турбо-Пролога.

Програма на мові Турбо-Пролог складається з кількох розділів. Кожний розділ програми ідентифікується ключовим словом, як показано в таблиці 1.

У програмі не обов'язково одночасно мають бути присутніми всі наведені в таблиці розділи. Наприклад, якщо пропущений розділ goal, то цілі можуть задаватися з вікна середовища Прологу безпосередньо під час виконання програми.

Таблиця 1 – Основні розділи програми на мові Турбо-Пролог

Розділ	Зміст
Опції компілятора	Опції компілятора, які задані на початку програми
Розділ constants	Нуль або більше констант
Розділ domains	Нуль або більше об'яв доменів
Розділ database	Нуль або більше предикатів бази данних
Розділ predicates	Нуль або більше об'яв предикатів
Розділ goal	Нуль або одна ціль
Розділ clauses	Нуль або більше речень (фактів та правил).

З іншого боку, вся програма може складатися з єдиного розділу goal. Наприклад,

```
goal
readint(X), Y=X+3, write("X+3=",Y).
```

Як правило, в програмі необхідні принаймні розділи predicates та clauses. Для більшості програм розділ domains є необхідним лише для

оголошеня списків, складних структур та власних імен для основних доменів.

1.8 Опис доменів та предикатів

Турбо-Пролог потребує вказати типи об'єктів для кожного предиката програми. Деякі з цих об'єктів можуть бути числовими даними, інші - символьними рядками. Наприклад, для того, щоб предикат ***likes*** («подобається») можна було використовувати в програмі, необхідно об'явити його:

```
predicates
likes(symbol,symbol).
```

Таке оголошення означає, що обидва об'єкти предиката ***likes*** відносяться до типу symbol, одного з базисних типів Турбо-Пролога, який буде описано нижче.

У деяких випадках бажано мати можливість трохи більшої конкретизації типу, який використовується предикатом об'єкта. Наприклад, в предикаті ***likes*** об'єкти мають такий зміст «той, хто любить» і «той, кого люблять». Турбо-Пролог дозволяє конструювати свої власні типи об'єктів з базисних типів доменів. Наприклад, ми можемо забажати присвоїти об'єктам предиката ***likes*** імена *person* (персона) та *thing* (річ). Для цього в розділі програми domains повинні з'явитися такі описи:

```
domains
    person,thing = symbol.
predicates
    likes(person,thing).
```

Імена *person* та *thing* будуть означати при цьому деякі сукупності (домени) значень. Будь-яке значення домена, наприклад person, може займати в твердженнях місце об'єкта person з відповідного предиката.

1.8.1 Опис доменів

Розділ *domains* містить оголошення доменів.

Турбо-Пролог має шість вбудованих стандартних типів доменів, опис яких наведено в таблиці 2.

Опис домена вигляду:

```
name = d.
```

оголошує домен name, який складається з елементів стандартного типу домену d; тип домену d повинен бути *integer, char, real, ref, string або symbol.*

Таке оголошення використовується для об'єктів, які синтаксично схожі, але відрізняються семантично.

Оголошення таким чином різних доменів дозволяє Турбо-Прологу контролювати домени, щоб уникнути випадкового їх змішування, наприклад, apples та mary. Але можна використати і знак рівності для перетворення між apples та mary.

Таблиця 2 - Опис стандартних типів доменів Турбо-Пролога

Тип домену	Опис стандартного типу домену
Char	Символ (8-бітовий ASCII-символ, який взятий в одиничні апострофи) належить домену char. Символ ASCII позначується символом \, за яким слідує код ASCII цього символу. \n та \t означають відповідно символи нового рядка та табуляції. Символ \, який слідує за будь-яким іншим символом, створює цей символ ('\\' створює \, а '\'' створює ')
Integer	Ціле число належить домену integer та знаходиться в межах від -32,768 до 32,767
Real	Дійсне число належить домену real та знаходиться в межах від ± 1e -307 до ± 1e +308. Дійсні числа записуються у вигляді знака, мантиси, десяткової точки, дробової частини, символу e, знака та показника - без пробілів. Наприклад, дійсне значення - $12345.6789 \times 10^{14}$ може бути записано як -1.23456789e+10. Знак, дробова частина та показник необов'язкові (хоча, якщо опускаєте дробову частину, то десяткову точку теж треба забрати). Турбо Пролог автоматично перетворює цілі числа в дійсні, коли це необхідно
String	Рядок (послідовність символов між парою подвійних лапок) належить домену string. Рядок може вміщувати символи, які створені послідовністю escape (як для char), довжина рядка не повинна перевищувати 250 символів
Symbol	Символьна константа (атом з іменем, який починається з маленької літери) належить домену типу symbol. Рядки теж сприймаються як елемени типу symbol, але елементи symbol зберігаються у внутрішній таблиці для найшвидшого зіставлення. Symbol-таблиця займає деяку область пам'яті, тому потребується деякий час, щоб війти в таблицю. Однак, якщо однакові елементи типу symbol часто порівнюються, їх варто туда помістити
File	Символічне ім'я файла належить домену file; це може бути як ім'я, яке починається з маленької літери та зустрічається в правій частині об'яви домену file, так і одне з наперед визначених символічних імен файлів: *printer, screen, keyboard, coml, stdin, stdout ma stderr*

Один предикат може використовувати домени різних типів:

payroll(employee_name, pay_rate, weekly_hours).

Цей предикат потребує такого опису доменів:

employee_name=symbol.

pay_rate=integer.
weekly_hours=real.

Прикладом коректного твердження, яке використовує предикат *payroll* може, при цьому, служити такий:

payroll("Artur Berman",16,45.25).

В Турбо-Пролозі є можливість стислої об'яви доменів. Наприклад, для об'яви стандартного домену *name = d*, ліва частина об'яви доменів може складатися зі списку імен:

mydom1, mydom2, ..., mydomN = ...

Така особливість дозволяє об'являти декілька доменів одночасно:

firstname, lastname, address = string.

1.8.2 Опис предикатів

В Турбо-Пролозі розділи, які озаглавлені ключовим словом *predicates,* містять оголошення предикатів. Предикат оголошується за допомогою його імені та доменів його аргументів:

predicates
predname(domain1, domain2, ..., domainN).

У цьому прикладі *predname* являє собою ім'я нового предиката, а *domain1, domain2, ..., domainN* позначують стандартні домени або домени, визначені користувачем.

Наприклад:

predicates
likes(symbol,symbol).
book(symbol,symbol,integer). та т.ін.

Оскільки тут використовуються стандартні базисні типи доменів, то відсутня необхідність описувати окремо домени об'єктів цих тверджень. Але можна явно описати всі використовувані домени об'єктів, наприклад, у предикаті *likes*. Тоді в розділах *domains* та *predicates* повинні з'явитися такі речення:

domains
name, fruit=symbol.
predicates
likes(name,fruit).

Для одного предиката дозволяються декілька оголошень. Наприклад, можно оголосити, що предикат *member* працює як з числами, так і з іменами, задавши таке оголошення:

predicates
member(name, namelist).
member(number, namelist).

У цьому прикладі аргументи *name, namelist, number* та *numberlist* є визначеними користувачем. Альтернативні оголошення для *member* повинні мати однакову кількість аргументів.

Можна оголошувати предикати з різними арностями:

Hanoi & вибирає за замовчуванням 10 дисків.
hanoi (integer) & пересуває N дисків.

У разі оголошення одного імені кілька разів, усі оголошення повинні йти одне за одним.

1.9 Використання зовнішніх та внутрішніх цілей

Програма на Турбо-Пролозі може містити в собі опис своєї цілі, але часто така ціль задається користувачем в процесі роботи програми, тобто є зовнішньою відносно програми.

Пикладом програми з внутрішньою цілью може служити закінчена програма з використанням предикатів та тверджень. Цілью програми є пошук синоніму деякого слова. Наприклад, синонімом слова *brave* є слово *daring*. На синонімічність цих слів вказує таке твердження:

synonym(brave,daring).

Опис предиката для цього твердженя буде мати такий вигляд:

synonym(word,syn),

де *word* та *syn* - об'єкти описаного предиката.

Наведемо повний лістинг програми:

```
domains
    word,syn,ant=symbol.
predicates
    synonym(word,syn).
    antonym(word,ant).
goal
    synonym(brave,X), write("A synonym for 'brave' is"),
    nl, write("",X,""), nl.
clauses
    synonym(honest,truthful).
    synonym(modern,new).
    synonym(brave,daring).
    antonym(honest,dishonest).
    antonym(modern,dismodern).
    antonym(brave,cowardly).
```

Оскільки ціль задана у самій програмі, вона є внутрішньою. Ціль

складається з п'яти підцілей, які розділені комами. Перша з них:

synonym(brave,X),

де X є вільною змінною, значення якої неконкретизовано (нагадаємо, що ім'я змінної об'єктів обов'язково повинно починатися з великої літери).

Друга підціль полягає у виведенні на екран рядка символів. Підціль створена за допомогою одного з вбудованих предикатів Турбо-Прологу *write*. Вбудовані предикати не потребують спеціального опису в програмі. Подвійні лапки використовуються для обмеження символьного рядка. Для імен змінних лапки не потрібні, як це видно з опису четвертої підцілі. Третя та п'ята підцілі є іншим вбудованим предикатом *nl*, який забезпечує перехід курсора на початок наступного рядка.

При запуску програми, в якій відсутній опис цілі, Турбо-Пролог пропонує ввести ціль з екрана. Наприклад, якщо з наведеного вище лістингу програми видалити розділ goal, то на екрані у вікні Dialog з'явиться запрошення Goal:. У відповідь треба сформулювати запит, наприклад:

synonym(modern,Y).

Внаслідок такого запиту буде отримано відповідь:

Y = new.
1 Solution.

Використання зовнішніх цілей буває корисним при записуванні коротких цільових формулювань, а також для отримання всього набору дозволених значень. Іншою перевагою є можливість адресувати базі даних цілком довільні запити.

1.10 Трасування програм

Використання на початку програми вбудованого предиката *trace*, або *shorttrace*, приводить до того, що всі предикати програми трасуються. Якщо за *trace* або *shorttrace* йде список імен предикатів, трасуються тільки вказані предикати. Під час трасування Турбо-Пролог виводить інформацію, наведену в таблиці 3.

Наприклад, для програми:

```
trace
domains
        list = integer*.
predicates
        eg(integer, integer).
        member(integer, list).
clauses
        member(X, [X\_]).
```

```
                    member(X, [_\L] :- member(X, L).
                    eg(X, X).
          goal
                    member(X, [1, 2], eg(X, 2).
```

буде отримано таку трасу:

```
CALL:     goal().
CALL:     member(_, [1, 2]).
RETURN: *member(1, [1, 2]).
CALL:     eg(1, 2).
FAIL:     eg(1, 2).
REDO:     member(_, [1, 2]).
CALL:     member(_, [2]).
RETURN: *member(2, [2]).
CALL:     eg(2, 2).
RETURN:   eg(2, 2).
RETURN:   goal().
```

Таблиця 3 - Повідомлення Турбо-Прологу в режимі трасування програм

Повідомлення	Зміст повідомлення
CALL	Кожен раз при виклику предиката, його ім'я та значення його параметрів відображаються у вікні трасування
RETURN	При виконанні речення у вікні трасування відображається результати конкретизації змінних. При наявності інших речень, які можна зіставити з вихідними параметрами, відображається зірочка (*), яка повідомляє, що це речення є точкою повернення
FAIL <ім'я_ предиката>	Означає невдале завершення предиката. Відображає ім'я предиката, який закінчився неуспішно
REDO	Показує, що мало місце повернення. У вікні трасування відображаються ім'я предиката, що викликається повторно, та значення його параметрів,

Контрольні запитання

1. Що означає термін «процедурне програмування»? Які мови процедурного програмування вам відомі?
2. Що таке «декларативне програмування»? В чому його основні відмінності від «процедурного програмування»?
3. Що таке концептуальна модель предметної області і для чого вона використовується?
4. Поясніть поняття «відношення»? Для чого відношення використову-

ються в логічному програмуванні?

5. Що таке «предметна область» задачі? З яких основних компонентів складається її опис?
6. Що таке «клас об'єктів» та «екземпляр об'єкта»? Для чого вони використовуються у логічному програмуванні?
7. Які типи тверджень Пролога ви знаєте?
8. Що таке терм?
9. Як логічна програма запускається на виконання? Що є обчисленням логічної програми?
10. Поясніть поняття І/АБО-дерева. Для чого воно використовується у Пролозі? Наведіть відповідний приклад використання І/АБО-дерева.
11. Чим відрізняються вершини типу «І» та вершини типу «АБО» у І/АБО-дереві?
12. Що означає термін «недетерміноване програмування»? Як реалізовано механізм недетрімнованого програмування у Пролозі?
13. Стисло охарактеризуйте механізм повернення Прологу. Наведіть приклад його дії.
14. Перелічіть основні фази роботи внутрішнього інтерпретатора Прологу. Дайте їх стислу характеристику.
15. Сформулюйте основні правила уніфікації термів у Турбо-Пролозі:
16. Що таке анонімна змінна, в яких випадках і для чого вона використовується?
17. Як реалізовується у Пролозі реляційна операція об'єднання?
18. Як реалізовується у Пролозі реляційна операція різниці?
19. Як реалізовується у Пролозі реляційна операція одекартова добутку?
20. Як реалізовується у Пролозі реляційна операція проекції?
21. Як реалізовується у Пролозі реляційна операція вибірки?
22. Як реалізовується у Пролозі реляційна операція з'єднання відношень?
23. Назвіть та стисло охарактеризуйте основні розділи програми на Пролозі.
24. Назвіть та стисло охарактеризуйте стандартні типи доменів.
25. Що таке «внутрішні» та «зовнішні» цілі Прологу? У чому полягає основна відмінність їх використання?
26. Який предикат забезпечує трасування логічної програми? Назвіть та стисло охарактеризуйте основні повідомлення Турбо-Пролога в режимі трасування.

Лабораторна робота №1
ПРОГРАМУВАННЯ НА ТУРБО-ПРОЛОЗІ З ВИКОРИСТАННЯМ ФАКТІВ ТА ПРАВИЛ

Мета роботи

Вивчення основ програмування на Турбо-Пролозі. Набуття практичних навичок програмування мовою Турбо-Пролог.

Завдання на підготовку

Студент повинен знати:

- основні теоретичні відомості про логіку предикатів;
- основні поняття мови Турбо-Пролог;
- типи даних в Турбо-Пролозі;
- порядок обробки фактів та правил Турбо-Прологом;
- правила уніфікації та конкретизації змінних;
- структуру програм Турбо-Пролога.

Студент повинен вміти:

- подавати дані за допомогою фактів та правил;
- відображати І/АБО-дерева обробки програм Турбо-Прологом.

Для допуску до виконання роботи необхідно:

- вміти відповісти на теоретичені запитання по ходу виконання роботи;
- надати викладачу заготівку звіту про лабораторну роботу, яка містить: титульний лист, опис вибраної предметної області (у вигляді графа) та попередній текст програми.

Завдання на лабораторну роботу

1. Обрати предметну область, для якої буде створюватись база даних.
2. Для обраної предметної області:

- нарисувати граф, вершини якого відображають об'єкти вибраної предметної області, а ребра - відношеня, які існують між об'єктами;
- написати програму з внутрішніми цілями, яка містить не менше як десять фактів та п'ять правил і дозволяє отримати відповідь не менше як на шість запитань;
- виконати трасування програми та записати фрагмент протоколу трасування у звіті з лабораторної роботи;
- описати роботу програми, помітивши точки розміщення маркерів повернення та порядок виконання тверджень програми;
- навести І/АБО-дерево підцілей для розробленої програми.

Приблизний перелік предметних областей

- родинні відношення;
- бібліотека;

- розклад (занять, руху тощо);
- студенти групи;
- кулінарія;
- карта України;
- міжміські переговори;
- світ тварин;
- світ рослин;
- військовослужбовці;
- постачання матеріалів та ін.

Оформлення роботи та порядок захисту

1. Наведіть у звіті:
- опис предметної області у вигляді графа;
- текст розробленої програми;
- ілюстрацію одного із запитів з використанням І/АБО-дерева підцілей з розставленими маркерами поверненнями;
- фрагмент трасування запиту, що містить усі види повідомлень, які виводяться при трасуванні.

2. При захисті роботи необхідно вміти пояснити кожен запис, відповісти на теоретичні запитання та продемонструвати практичні навички зі створення простих програм на Турбо-Пролозі.
3. Прикладом фактів та правил предметної області можуть бути такі:
   - відношення-факти:
   <група> (<шифр>,<факультет>,<кількість_студентів>,<староста>).
   <дисципліна> (<шифр>,<назва>,<кількість_годин>).
   <викладача> (<ПІБ>,<кафедра>,<телефон>).
   <розклад занять> (<шифр_групи>,<шифр_дисципліни>,<ПІБ>
   <номер_аудіторії>, <день_тижня>,<час>,
   <чисельник/знаменник >).
   - відношення-правила: які аудиторії зайняті у даний час, які аудиторії вільні у даний час, розклад занять на даний час, де знаходиться викладач у даний час, де знаходиться студент у даний час, і т. ін.

Запитання для самоконтролю

1. Назвіть основні типи речень Пролога.
2. Як називається речення, що містить твердження, яке є завжди безумовно правильним?
3. Як називється речення Пролога, яке вміщує твердження, правдивість якого залежить від деяких умов?
4. Що називають головою та тілом речення в Пролозі?
5. Які речення в Пролозі мають:
   а) пусте тіло? б) тільки тіло? в) голову та непусте тіло?
6. Що розуміється під конкретизацією змінної?
7. Відтранслюйте такі твердження в правила на Пролозі:

- всякий хто має дитину - щасливий (введіть одноаргументне відношення ***щасливий***);
- всяка особа що має дитину, у якої є сестра, має двох дітей (введіть відношення ***мати_двох_дітей***).

8. Визначіть відношення ***онук,*** використавши відношення ***батько***.
9. Визначіть відношення ***тітка***(X,Y) через відношення ***батько*** та ***сестра***.
10. Назвіть об'єкти даних Пролога.
11. Які з таких речень є правильними об'єктами Пролога, та що це за об'єкти:
    (а) Діана; (б) діана; (в) "Діана"; (г) _діана; (д) "Діана їде на південь"; (е) їде(діана, південь); (є) 45; (ж) 5(X,Y); (з) +(північ, захід); (і) три(Чорних(Коти)).
12. Що таке функтор?
13. Які правила зіставлення (уніфікації) термів Ви знаєте?
14. Укажіть, які з наведених операцій зіставлення будуть успішними, а які ні. Для успішних зіставлень назвіть результуючі конкретизації змінних:
    (а) крапка (A,B) = крапка(1,2); (б) крапка (A,B)=крапка (X,Y,Z);
    (в) плюс (2,2)=4; (г) +(2,D)=+(E,2); (д) трикутник (крапка (-1,0), P2,P3) = трикутник (P1, крапка (1,0), крапка (0,Y).
15. Які з наведених пар термів уніфікуються:
    (а) джек(х) – джек (дюдина); (б) джек (особистість) – джек (людина);
    (в) джек (X,X) - джек (23,23); (г) джек (X,X) – джек (12,23);
    (д) джек (_,_) – джек (12,23); (е) f (Y,Z) - X (ж) X - Z.
16. Охарактеризуйте п'ять основних розділів програми на Турбо-Пролозі.
17. Наведіть приклади опису типів даних в Турбо-Пролозі.
18. Поясніть на прикладі функціонування механізму повернення в Турбо-Пролозі.
19. Виконайте вручну трасування запропонованого Вам фрагмента програми.
20. Дано лістинг програми:

```
clauses
        big (bear).
        big (elefant).
        small (cat).
        brown (bear).
        black (cat).
        gray (elefant).
        dark(Z):- black(Z).
        dark(Z):- brown (Z).
goal  dark(Z), brown (Z).
```

Опишіть процедуру пошуку відповіді Прологом на наведений запит. Порівняйте процедуру трасування даного запиту з трасуванням запиту brown (Z)б dark(Z).

У якому випадку Пролог потребує більше роботи для пошуку відповіді?

2 СКЛАДЕНІ ОБ'ЄКТИ ДАНИХ

2.1 Поняття складених об'єктів

Типи простих об'єктів у Турбо-Пролозі обмежено шістьма типами доменів (див. табл. 2).

Розглянемо приклад твердження:

має("Маша","Гарфілд"). /* Маша має Гарфілд*/

Перший об'єкт «Маша» має просту структуру, він подає сам себе. Те ж саме можна сказати і про другий об'єкт. Такі об'єкти називаються простими об'єктами.

Наведене твердження подає той факт, що Маша має Гарфілд, який може бути, наприклад, назвою книги або іменем свійської тварини. Для поділу цих випадків можна записати твердження у формі, що більш докладно описує об'єкт:

має("Маша",собака("Гарфілд").
має("Маша",книга("Гарфілд").

Об'єкт, що подає інший об'єкт або сукупність об'єктів, називається складеним об'єктом. Записані у такий спосіб предикати «має» називаються складеними структурами, оскільки вони скомпоновані зі складених об'єктів.

Наприклад, твердження про те, що Коля любить яблука, апельсини й банани можна записати у вигляді:

любить("Коля", яблука, апельсини, банани).

Усі ці три види фруктів можна об'єднати в одній окремій структурі:

Фрукти (яблука, апельсини, банани),

внаслідок чого з'явиться можливість використання складеного об'єкта:

любить ("Коля", фрукти (яблука, апельсини, банани)).

У даному твердженні є два функтори: *любить* і *фрукти*, причому функтор «любить» - називається головним.

Для полегшення написання тверджень і предикатів у даній формі Турбо-Пролог дозволяє оголошувати складені об'єкти в розділі domains. Для розглянутого прикладу описами будуть служити:

```
domains
        уподобання_особи = фрукти (mun1, mun2, mun3).
        mun1, mun2, mun3 = symbol.
```

Ім'я домена *уподобання_особи* є іменем складеного домена, утвореного за допомогою функтора (предикатного символу «фрукти»).

Структура називається однодоменною, якщо всі її об'єкти

належать до одного й того ж самого типу доменів. В іншому випадку структура називається багатодоменною.

Розглянемо приклад використання доменних структур на прикладі програми "Бібліотека", що містить відомості про книги з особистих колекцій:

```
        /* Бібліотека */
    domains
      personal_library=book (title,author,publication).
      publication=publication (publisher,year).
      collector,title,author,publisher=symbol.
      year=integer.
    predicates
      collection(collector,personal_library).
```

/* Як бачимо з наведеного опису функтор структури «personal_library» має ім'я «book». Використання доменної структури спрощує структуру предиката «collection».*/

```
    clauses
      collection ("Ivan",book ("Logical_programming","Bratko",
            publication("USA",1990))).
      collection ("Ivan",book ("Database","Majers",
            publication("Au",1986))).
      collection ("Boris",book ("Symbolic_Logic","Carroll",
            publication("GB",1987))).
      collection ("Boris",book ("Problem_solving","Nillson",
            publication("AW",1985))).
            /*  Кінець програми  */
```

Дана програма використовує зовнішню ціль. Для того, щоб довідатися які книги належать Івану достатньо ввести запит:

collection ("Ivan",Books).

Щоб знайти книги видані у 2005 році достатньо ввести запит:

collection (_,book (Title,_publication (_,2005))).

Використання доменних і предикатних структур надає більшої зручності при використанні складних баз даних. Використання більшої кількості функторів дає можливість задавати базі даних більш складні запитання.

Припустимо тепер, що треба написати програму, яка зберігає інформацію про країни, і здатна відповідати на запитання про ці країни, наприклад такі:

- яке населення Єгипту?

- яку назву має столиця Франції і яке її населення?
- у якому годинному поясі знаходиться Москва?
- яка щільність населення в Лівії?
- чи є такі мови на яких розмовляють і у Німеччині і у Румунії?
- чи є країни, які граничать як з Єгиптом, так і з Мароко?

Одним з аспектів даної задачі є подання інформації про кожну країну. У цьому випадку найпростішою формою подання інформації задається використання предиката розмірністю 8, який відповідає восьми інформаційним об'єктам:

країна (Назва,Населення,Площа,Назва_столиці,Населення_столиці, Часовий_пояс_столиці, Список_мов,Сисок_сусідніх_країн).

При цьому опис, наприклад, Єгипту, набуде такої форми:

країна (єгипет, 42, 1000, каїр, 7500, 3, [арабська, берберська, нубійська, англійська], [ізраіль, судан, лівія]).

Найбільш суттєвим недоліком такого подання є втрата сутності даних, що зберігаються. Дані складаються з унікального ключа (назва країни) і деякої приєднаної до нього інформації, яка, у свою чергу, повинна мати власну структуру: одна частина цієї інформації має відношення до всієї країни, інша – тільки до її столиці, третя – до мов, які використовуються в країні, четверта – до сусідніх країн.

Найкращим розв'язком у цьому випадку для опису, наприклад Єгипту, буде використання такої структури:

країна (назва (єгипет), дані (населення_в_млн (42), площа_в_тис_км (1000),
столиця (назва(каїр), населення_в_тис (7500),часовий_пояс (плюс,3)),
мови [арабська, берберська, нубійська, англійська],
сусідні_країни[ізраіль, судан, лівія]).

При цьому інформаційна структура задана у вигляді ключа і даних, відображається в процедури, які їх обробляють:

знайти_і_обробити (Країна):-
країна (Країна, Дані),
.....

Якщо надалі збережена інформація буде змінена, ніякі процедури, що обробляють базу даних на верхньому рівні, не будуть потребувати зміни. Зауважимо також, що функтор структури повинен містити зміст інформації вкладеної до його компонент, але не повинен сам містити цю інформацію. З цієї причини краще подання інформації про часовий пояс

надає структура часовий_пояс (плюс,3), а не структура часовий_пояс (плюс(3)) або часовий_пояс_плюс(3).

Подання у вигляді складених об'єктів поєднує дві основні переваги: воно добре структуровано і добре сприймається. Щоб відгородити користувача від зайвих деталей, пов'язаних з поданням інфомрації, предикатам надається здатність витягати окремі компоненти із загальної структури. Зокрема для відповіді на запит вигляду:

?-населення (єгипет,Населення).
Населення = населення_в_млн(42).

слід побудувати процедуру для предиката "населення":

населення (Країна,Населення):-
країна (назва(Країна),
дані (Населення,_,_)).

2.2 Предикати відбору

Предикат відбору [13] дозволяє програмісту маніпулювати об'єктом бази знань, навіть якщо структура бази прихована від нього. Метод програмування на основі предикатів відбору містить такі основні кроки:

- виявлення можливих інформаційних запитів, а також з'ясування того, які підмножини даних значимі для користувача при відповіді на ці запити;
- встановлення того, як користувач специфікує об'єкти бази знань, які необхідно знайти. Як правило задається значення ключа;
- для кожного передбачуваного запиту будується певний предикат відбору.

Якщо запит стосується інформації, яка зберігається у вигляді фактів, для витягання її зі струкутри предикат відбору використовує зіставлення. Якщо ж інформація зберігається у неявно вираженому вигляді, її обчислює процедура. При цьому програмісту, що використовує даний предикат, не помітна різниці між інформацією, що зберегається у явному вигляді, і інформацією, що виведена з використанням правил.

У розглянутому прикладі при відповіді на інші запити можуть використовуватися, наприклад, такі процедури для предикатів відбору:

```
столиця_і_її_населення (Країна, Столиця, Населення):-
   країна (назва(Країна),дані (_,_,столиця (Столиця,
      Населення,_))).
часовий_пояс (Місто, Пояс):-
   країна(_,дані(_,_,столиця (назва (Місто),_,Пояс))).
щільність_населення (Країна, щільність (D)):-
      країна (назва (Країна),дані (населення_в_млн(P)
         площа_в_тис_км(A),_)),
   ..... /*тут випливає обчислення щільності населення*/
```

2.3 Структурні діаграми

Для аналізу і наочного подання складених структур служать структурні діаграми [5]:

- доменні структурні діаграми (ДСД);
- предикатні структурні діаграми (ПСД).

Ці діаграми наочно демонструють структуру доменів і предикатів і є зручним засобом при написанні й документуванні програм.

Для розглянутого прикладу "Бібліотека" вони набудуть вигляду, наведеного на рис. 7 і 8, відповідно :

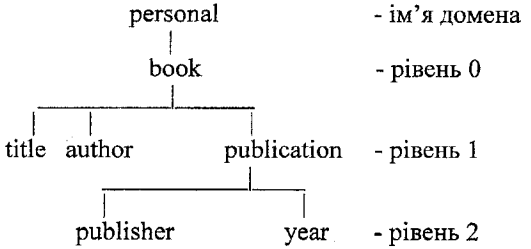

Рисунок 7 – Доменна структурна діаграма Пролог-програми «Бібліотека»

Рисунок 8 – Предикатна структурна діаграма Пролог-програми «Бібліотека»

2.4 Використання альтернативних доменів

Подання даних часто потребує наявності великої кількості структур, кожну з яких необхідно описати. Крім того, часто виникають труднощі з предикатами, що працюють з об'єктами цих доменів. Для усунення цього недоліку у Турбо-Прологі використовуються альтернативні домени.

Припустимо, необхідно створити структуру, що містить відомості про власність різних людей: особисті бібліотеки, фонотеки, спортивний інвентар. При цьому виникає необхідність опису предикатів. У традиційному стилі це можна зробити, наприклад, таким чином:

```
domains
  person,whatever,author,title,artist,album,=symbol.
  type=integer.
  misc_thing=misk_thing(whatever).
  book_library=book(author,title).
  record_library=record(artist,album,type).
predicates
  personal_things(person,misc_things).
  personal_books(person,book_library).
  personal_records(person,record_library).
clauses
  personal_things("Bill",misc_things("sail boat")).
  personal_books("Bill",book_library("Freid","Totem")).
  personal_records("Bill",record_library("Elton John,
          "Ice of Fire",25)).
```

При цьому доводиться використовувати три предикати, що описуються трьома доменними структурами: *misc_thing*, *book* і *record*.

Але існує можливість об'єднання всіх трьох структур в одну, під іменем *thing*. Така можливість надається використанням альтернативних доменів:

```
domains
   thing=misk_thing(whatever);
       book(author,title);
       record(artist,album,type).
   person,whatever,author,title,artist,album=symbol.
   type=integer.
```

Альтернативні домени розділяються знаком ";". При такому визначенні для забезпечення зв'язку між персоною і предметом її володіння достатньо ввести один єдиний предикат:

```
predicates
        owns(person,thing).
```

Тепер опис правил значно спроститься:

```
clauses
  owns ("Bill",misc_things ("sail boat")).
  owns ("Bill",book_library ("Freid","Totem")).
  owns ("Bill",record_library ("Elton John,"Ice of Fire",25)).
```

Відповідно спростяться й запити до програми.

2.5 Використання правил у запитах

У Турбо-Пролозі запити будуються із предикатів, що містять умови, які обмежують шляхи пошуку бажаних результатів. При цьому запит може

містити велику кількість умов, які необхідно щораз акуратно набирати у разі необхідності багаторазового повторення запитів. Уникнути такої важкої роботи можна шляхом відмови від використання тверджень бази знань при формулюванні запиту.

Цього можна досягти конструюванням правил, що не містять у собі даних, тобто правил нульової арності.

Наприклад, у раніше розглянутій лабораторній роботі n1 програмі про уподобання марка та андрійка спростити систему запитів, увівши в розділ clauses такі додаткові правила:

що_любить_коля:- like ("Коля",X),write ("Коля любить",X),nl.
за_кого_уболіває_марко:- like ("Марко",X),write ("Марко уболіває за",X),nl.
Назви_команд:- команда (X),write ("Назва команди",X),n,. тощо.

Тепер, для одержання відповіді на запитання, досить набрати запит, що містить лише одну відповідну букву. Зрозуміло, що в більших базах даних, які містять значні обсяги інформації, описаний підхід надає більші переваги для організації запитів за рахунок передчасного введення в розділ clauses значної кількості формулювань найбільш очікуваних запитів.

Контрольні запитання

1. Які типи об'єктів Прологу Вам відомі?
2. Які типи тверджень Прологу Ви знаєте?
3. Що таке складені об'єкти і які переваги надає їх використання?
4. Як описуються складені об'єкти у розділі доменів?
5. Що таке багатодоменна структура?
6. Поясніть поняття «доменна структура».
7. Поясніть поняття «предикатна структура».
8. Застосуйте І/АБО графи для опису доменної структури складеного об'єкта.
9. Поясніть поняття «предикат відбору».
10. Для чого використовуються предикати відбору?
11. З яких основних кроків складається метод програмування на основі предикатів відбору?
12. Наведіть власний приклад предиката відбору і поясніть, у чому полягають його преваги.
13. Наведіть приклад доменної структурної діаграми та поясніть її призначення та переваги, що надає її використання.
14. Наведіть приклад предикатної структурної діаграми та поясніть її призначення та переваги, що надає її використання.
15. Що таке альтернативні домени і для чого вони використовуються? Наведіть власний приклад.
16. Які переваги надає використання правила у запитах? Чи має використання правил у запитах недоліки? Якщо так, то які саме?

Лабораторна робота № 2
ВИКОРИСТАННЯ СКЛАДЕНИХ ОБ'ЄКТІВ У ТУРБО-ПРОЛОЗІ

Мета роботи

Набуття навичок використання складених об'єктів у Турбо-Пролозі.

Завдання на підготовку

Студент повинен знати:

- поняття та основні властивості складених предикатів;
- принципи використання складених об'єктів;
- правила побудови структурних діаграм;
- поняття альтернативних доменів;
- методи використання правил у запитах.

Студент повинен уміти:

- працювати в середовищі Турбо-Пролог;
- будувати складені предикати;
- відображувати складені об'єкти з використанням структурних діаграм;
- використовувати предикати відбору для оптимізації запитів.

Для допуску до виконання роботи необхідно:

- вміти відповісти на теоретичні запитання по ходу виконання роботи;
- подати викладачу заготівку звіту про лабораторну роботу, що містить: титульний аркуш і попередній текст програми, з розробленими, відповідно до завдання, складеними об'єктами та предикатами відбору.

Завдання на лабораторну роботу

1. Доповнити розроблену в попередній лабораторній роботі програму правилами, що включають складені об'єкти, та операторами відсікання і повернення.

2. Відобразити використані в роботі складені об'єкти за допомогою доменної і предикатної структурних діаграм.

Оформлення роботи та порядок захисту

1. Наведіть у звіті:

- текст додатково розроблених фрагментів програми;
- графічні ілюстрації роботи предикатів відсікання та повернення;
- доменні і предикатні структурні діаграми.

2. При захисті роботи необхідно вміти пояснити кожен запис, відповісти на теоретичні запитання та продемонструвати практичні

навички використання складених предикатів та операторів відсікання та повернення.

Запитання для самоконтролю

Розгляньте приклад Пролог-програми «Бібліотека»:

```
/* Програма: Бібліотека     Файл: DEMOLIB.PRO */
domains
personal_library= book(title, author, publication)
publication=publication(publisher, year)
collector,title,quthor,publisher=symbol
year=integer

predicates
collection(collector, personal_library)

clauses
collection ("Kahn",
        book("The Computer and the Brain",
              "von Neumann",
             publication("Yale University Press",
                    1958))).
    collection ("Kahn",
        book("Symbolic Logic",
              "Lewis Carroll",
             publication("Dower Publications",
                  1958))).
    collection ("Johnson",
        book("Database: A Primer",
              "C.J.Date",
             Publication("Addison Wesley",
                  1983))).
    collection ("Johnson",
        book("Problem-Solving Methods in AI",
              "Nils Nilsson",
             Publication("MgGrow Hill",
                  1971))).
    collection (smith,
        book("Alice in Wonderland",
              "Lewis Carroll",
             Publication("The New American Library",
                  1980))).
```

```
collection (smith,
    book("Fabkes of Aesop",
        "Aesop-Calder",
        Publication("Dower Pubications",
            1967))).
```

Дайте відповідь на такі запитання:

1. Зі скількох рівнів складається доменна структурна діаграма програми «Бібліотека»?
2. Зі скількох рівнів складається предикатна структурна діаграма програми «Бібліотека»?
3. Графічно відобразіть доменну структурну діаграму програми «Бібліотека».
4. Графічно відобразіть предикатну структурну діаграму програми «Бібліотека».
5. Чи буде виведено у відповідь на запит collection (smith,Books) хоча б одна книга із зібрання Kahn?
6. Якщо об'єкти title та author розмістити у підструктуру volume з таким означенням:

   volume= volume (author, title),

   і ввести ціль:

   collection (_,Book, Volume,_),

   чи видасть програма список усіх авторів та книжок?
7. Поясніть доцільність та переваги альтернативних доменів у програмі, створеній Вами під час виконання лабораторної роботи.
8. На прикладі створеної Вами під час виконання лабораторної роботи програми, поясніть переваги використання предикатів відбору.

3 ІТЕРАЦІЯ І РЕКУРСІЯ

3.1 Методи повторення й рекурсії

Часто в програмах виникає необхідність повторного виконання однієї і тієї ж самої операції кілька разів. У Турбо-Пролозі така задача реалізовується з використанням ітераційних (тобто таких, що базуються на механізмі повернення) або рекурсивних правил.

Ітераційне правило повторення операції на основі механізму повернення має такий узагальнений вигляд:

```
repetitive_rule:-                    /*правило повторення*/
        <предикати й правила>,
        fail.                        /*невдача*/
```

Останьою підціллю такого правила виступає вбудований предикат *fail*, який за своїм визначенням завжди закінчується невдачою. Таким чином, у разі успішного виконання правила (тобто, доведення істинності всіх підцілей правила, розташованих перед *fail*), програма викликає предикат *fail,* який ініціює перехід до останнього маркера повернення і продовження пошуку наступного розв'язку.

Основане на рекурсії правило має такий узагальнений вигляд:

```
recursive_rule:-   /*правило рекурсії*/
                   <предикати й правила>,
                   recursive rule.
```

Останнім твердженням у тілі даного правила розташовується саме правило *recursive_rule*, яке знов викликає само себе при досягненні узгодження усіх попередніх підцілей.

Правила повторення й рекурсії можуть забезпечувати однаковий результат, але алгоритми їх суттєво різняться. Рекурсія, як правило, потребує більших ресурсів, але більш компактно описує значну кількість достатньо складних задач і є основним механізмом обчислень у Пролозі.

3.2 Метод повернення після невдачі

При наявності у тексті Пролог-програми внутрішньої цілі, механізм уніфікації Турбо-Прологу зупиняє пошук розв'язку після першого ж успішного обчислення цілі. Інші набори значень змінних, які можуть служити розв'язками задачі, не будуть активізуватися доти, поки програма не змусить внутрішній механізм уніфікації повторно продовжити пошук додаткових наявних розв'язків. Для цього і використовується предикат fail, призначення якого полягає у завжди невдалому обчисленні. Отож, підціль з використання цього предиката ніколи не доводиться, а, відповідно, завжди включається механізм пошуку з поверненням,

внаслідок чого Турбо-Пролог виконує перехід до останнього маркера повернення і процес пошуку розв'язків буде повторюватися доти, поки не будуть оброблені всі твердження програми.

Насправді, для отримання такого ж самого ефекту можна записати, наприклад ціль 2 = 3. Ефект буде абсолютно аналогічним. Предикат fail використовується у тих випадках, коли у програмі є внутрішня ціль, і необхідно забезпечити виведення усіх розв'язків.

Розглянемо простий приклад використання вбудованого предиката *fail* для забезпечення повторів пошуку розв'язку задачі:

```
domains
  name,sex,department=symbol.
  pay_rate=real.
predicates
  employee(name,sex,department,pay_rate).
  show_male_part_time.
clauses
  employee("John","M","A1",3.5).
  employee("Nick","M","A5",4.5).
  employee("Irving","M","A1",4.0).
  employee("Diana","F","A1",2.5).
  employee("Kate","F","A5",4.5).
  employee("Mary","F","A5",5.0).
 show_male_part_time:-
  employee(Name,"M",Dept,Pay_rate),
  write(Name,Dept,"$",Pay_rate),
  nl,
  fail.

goal

show_all_part:-
                employee(Name,Sex,Dept,Pay_rate),
                write(Name," ",Sex," ",Dept," ",Pay_rate),nl,
                fail.
```

Твердження програми містять дані про службовців компанії. Предикат бази знань має вигляд:

службовець(ім'я,стать,відділ,погодинна_оплата).

Отримати вміст усієї бази даних дозволяє цільове твердження у розділі *goal:*

```
show_all_part:-
                employee(Name,Sex,Dept,Pay_rate),
                write(Name," ",Sex," ",Dept," ",Pay_rate),nl, fail.
```

Оскільки всі змінні в правилі не конкретизовані, то при уніфікації з аргументами першого предиката програми, що є фактом з константними значеннями аргументів

employee("John","M","A1",3.5),

усі вони конкретизуються відповідними значеннями (перша підціль правила).

При цьому маркер повернення буде встановлено на таке твердження програми:

employee("Nick","M","A5",4.5).

Виконання другої підцілі правила *show_male_part_time* забезпечить виведення на екран отриманих з бази знань відомостей. Якби в правилі була відсутня третя підціль (*fail*), то Турбо-Пролог завершив би роботу успішно довівши ціль. Але третя підціль ніколи не буває успішною, внаслідок чого Турбо-Пролог змушений буде продовжувати пошук розв'язків, переходячи до найближчого маркера повернення. Тобто, він перейде до обробки другого твердження програми, встановивши на нього маркер повернення. Процес буде тривати доти, поки не буде оброблено останнє твердження, тобто не будуть вичерпані всі маркери повернень.

Записане в програмі правило дозволяє здійснити вибіркове виведення на екран списку тільки службовців чоловічої статі (подумайте і поясніть як працює це правило).

Бачимо, що програма дійсно виведе перелік усіх службовців. Але слід зробити одне важливе зауваження. Хоча програма і виконала усі дії, які від неї вимогалися, сама робота програми закінчилася невдачею, оскільки ціль *show_all_part* з секції *goal* доведена не буде. Недоведення цілі *show_all_part* відбудеться тому, що коли мета *employee(Name,Sex,Dept,Pay_rate)* буде в останній раз успішно доведена фактом *employee("Mary","F","A5",5.0),* у дію знов вступить мета *fail.*

Але передовести ціль *employee(Name,Sex,Dept,Pay_rate)* більше неможливо, оскільки усі факти вже вичерпано. Отже, оскільки так і не вдалося довести ціль в тілі правила і головна ціль вважається недоведеною, тобто буде вважатися недоведеною ціль *show_all_part* з секції *goal.*

Насправді запобігти цьому недоліку дуже просто. Для цього достатньо лише додати у кінець програми ще один предикат *show_all_part* без аргументів, який завжди буде істинним. Наявність другого твердження для предиката *show_all_part* створить ще одну точку повернення. Коли ціль *show_all_part* у черговий раз своїм недоведенням запустить пошук з поверненням, а пере-довести мету *employee(Name,Sex,Dept,Pay_rate)* вже буде неможливо внаслідок вичерпання усіх фактів, тоді пошук з поверненням і повернется до передоведення цілі *show_all_part*, що

з успіхом виконається за допомогою завжди істинного другого твердження

show_all_part.

Наведемо ще один приклад управління пошуком з поверненням без використання предиката *fail*. Створимо програму, яка буде виводити додатні числа доти, пока не зустрінеться перше від'ємне число. При цьому програма має завершити роботу:

```
predicates
number (integer).
output
clauses
number (2).
number (1).
number (0).
number (-1).
number (-2).
output:- number (Positive_number), write (Positive_number),
nl, Positive_number<0.
output.
goal
output.
```

У даному випадку ціль *Positive_number<0* начебто відіграє роль умовного предиката *fail*. Доки ціль *number (Positive_number)* буде доводитися фактами, що містять додатні числа, ціль *Positive_number<0* доводитися не буде, і, відповідно, буде запускатися пошук з поверненням. Але як тільки змінна Positive_number буде конкретизована значенням –1, ціль *Positive_number<0* буде успішно доведена. Отже, доведення усього правила і цілі *output* у секції *goal* закінчиться успіхом. У цьому прикладі друге речення для предиката *output* потребується тільки на той випадок, якщо усі факти *number* будуть містити лише додатні числа.

3.3 Метод відсікання й повернення

Цей метод використовується для фільтрації даних, що вибираються з бази за багатьма умовами. Наприклад, метод відсікання і повернення дає можливість вибрати всі заявки, зафіксовані з 18 по 19 червня, що мають літеру В у номері накладної і отримані до того, як службовець Романюк заступив на чергування.

Які можливості надає спільне використання предиката *fail* з іншим вбудованим предикатом cut (відсікання), який позначається символом «!». На відміну від *fail*, виконання якого завжди закінчується невдачею і примушує програму переходити до останнього маркера повернення,

обчислення предиката *cut* завжди завершується успішно, змушуючи внутрішній механізм уніфікації Турбо-Прологу знищити усі маркери повернення, встановлені під час спроби обчислення поточної підцілі.

Іншими словами, *cut* "встановлює бар'єр", що забороняє виконувати повернення до всіх альтернативних розв'язків поточної підцілі. Але наступні підцілі можуть створювати нові маркери повернення і, тим самим, створювати умови для пошуку нових розв'язків.

Предикат *fail* використовується для імітації невдалого обчислення виконання наступного повернення доти, поки не буде виявлена певна умова. Предикат *cut* служить для заборони всіх наступних повернень.

Розглянемо принцип дії методу на прикладі бази знань, що містить список імен:

```
    /*Програма: список імен*/
domains
  name(symbol).
predicates
  person(name).
  show_some_of_names.
  make_cut(name).
  go_and_get_names.
clauses
  person("Tom").
  person("Bob").
  person("Alice").
  person("Pat").
  person("Tom").
  person("Lee").
  person("Beth").
  person("Tom").
  show_some_of_names:-person(Name),write(Name),nl,
              make_cut(Name),!.
  make_cut(Name):-Name="Lee".
  go_and_get_names:-
  person(Name),Name="Tom",write(Name),
        nl,fail.
```

Визначення цільовим правилом *show_some_of_names* забезпечує послідовний перегляд списку імен доти, поки не зустрінеться ім'я *Lee*. При цьому істинним стане предикат *make_cut* і в правилі *show_some_of_names* здійсниться перехід до підцілі *!*. Виконання предиката *cut* призведе до знищення всіх маркерів повернення і завершення роботи програми. Такий спосіб використання методу відсікання і повернення називається методом

ранжованого відсікання.

Запит go_and_get_names забезпечує виведення тільки імені Tom стільки раз, скільки це ім'я зустрічається в базі даних.

Зауважимо, що оскільки при поверненні Пролог анулює усі знайдені раніше значення змінних, для їх виведення обов'язково використовується предикат *write*. Без використання такого предиката, як однієї з підцілей правила, що генерує значення, які потім відкідаються предикатом *fail*, всі ці значення просто будуть втрачені.

Дослідимо більш детально роботу предиката відсікання *cut*.

Нехай є набір фактів, що описує деякі імена:

```
predicates
  first_name(symbol).
  second_name(symbol).
  last_name(symbol).
  show_full_name.
clauses
  first_name(marko).
  first_name(andriy).
  second_name(roman).
  second_name(ostap).
  last_name(boyko).
  last_name(lutsyuk).
  show_full_name:- first_name(X), second_name(Y),
     last_name(Z), write (X, " ", Y ," ", Z,";"," ", ), nl, fail.
  show_full_name.
goal
  show_full_name.
```

Результатом роботи програми буде генерування повних імен (у дужках наведені ідентифікатори маркерів, повернення до яких сприяє отриманню даного результату):

marko roman boyko; (МП3) marko roman lutsyuk;
(МП2) marko ostap boyko;(МП3) marko ostap lutsyuk;
(МП1) andriy roman boyko; (МП3) andriy roman lutsyuk;
(МП2) andriy ostap boyko; (МП3) andriy ostap lutsyuk;

У даному випадку в програмі є три маркери повернення МП1 – МП3, які і приводять до такого результату (рис. 9).

Отже, узгодивши першу підціль з фактом f1 програма залишає перший маркер МП1 повернення на факті f2, узгодивши другу підціль з фактом s1, вона залишає маркер повернення МП2 на факті s2, узгодивши третю підціль з фактом l1 - залишає маркер повернення МП3 на факті l2 (результат: *marko roman boyko)*. Далі, за рахунок переузгодження третьої

підцілі завдяки використанню предиката fail, здійснюється повернення до останнього маркера МП3 і, внаслідок переузгодження третьої підцілі, формується результат *marko roman lutsyuk.*

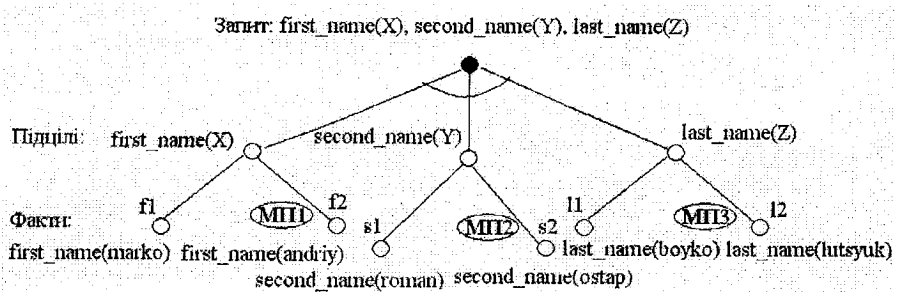

Рисунок 9 – І/АБО – дерево підцілей програми генерування повних імен

Оскількі факти для третьої підцілі вичерпано, Пролог повертається до маркера МП2, звільнюючи змінні X і Y. Переузгодження другої підцілі значенням Y2=ostap (і, відповідно, два повторні узгодження третьої підцілі), приводять до результату:

marko ostap boyko; marko ostap lutsyuk.

Накінець, при поверненні до маркера МП1 здійснюється звільнення усіх трьох змінних, а подальше переузгодження підцілей правила приводить до формування залишку реультату:

andriy roman boyko; andriy roman lutsyuk; andriy ostap boyko;
andriy ostap lutsyuk.

Модифікуємо цільове парвило, розмістивши після першої підцілі предикат відсікання:

show_full_name:- first_name(X), !, second_name(Y),
last_name(Z), write (*X*, " ", *Y*," ", *Z*,";"," ",), nl, fail.

При цьому буде знищено маркер повернення МП1, що призведе до відповдіної зміни результатів роботи програми:

marko roman boyko; (МП3) marko roman lutsyuk;
(МП2) marko ostap boyko;(МП3) marko ostap lutsyuk;

Перемістимо предикат відсікання на місце після другої підцілі:

show_full_name:- first_name(X), second_name(Y), !,
last_name(Z), write (*X*, " ", *Y*," ", *Z*,";"," ",), nl, fail,

що приведе до знищення двох маркерів повернення: МП1 і МП2. Результат роботи програми відповідно зміниться на такий:

marko roman boyko; (МП3) marko roman lutsyuk;

Накінець, при розташуванні предиката відсікання після третьої підцілі будуть знищені усі маркери повернення, внаслідок чого програма поверне один єдиний результат:

marko roman boyko.

Предикат відсікання використуовується для обмеження перебору, коли програміст впевнений у відсутності сенсу його виконання у певній частині простору пошуку розв'язків.

Розглянемо задачу класифікації, коли маємо декілька взаємовиключних тверджень. Припустимо, ми хочемо розділити спортсменів на три категорії: перша – ті, хто ніколи не програвав; друга – ті, хто мають і перемоги, і поразки; третя – ті, хто не досяг жодної перемоги. Запишемо задачі у вигляді програми на Пролозі, формалізувавши її у вигляді правил бази знань і фактів про результати трьох поєдинків чотирьох спортсменів:

predicates

category (symbol, integer).
winner (symbol, symbol).
clauses
winner (marko, andriy).
winner(petro, ostap).
winner(petro,marko).
category (X,2):- winner(X,_), winner(_,X).
category (X,1):- winner(X,_).
category (X,3):- winner(_,X).

Анонімна змінна використовується, оскільки нас не цікавіть, кого саме перемагав спортсмен, або від кого саме він зазнавав поразок.

Дослідимо програму. Цільове твердження

category (marko,X).

приведе до отримання трьох результатів:

X=2.
X=1.
X=3.
3 Solution.

Для цільового твердження

category (petro,X).

Пролог поверне два результати:

X=1.
X=1.

2 Solution.

Це відбувається тому, що дослідивши, завдяки механізму повернення, усі три варіанти узгодження цільового твердження *category (marko,Y),* Пролог встановить істинність усіх їх (відповідно, цільове твердження *category (petro,X)* буде узгоджено з двома фактами з бази знань). Але ж твердження є взаємовиключними. Тобто, отримавши одне рішення, нам не треба шукати інші, оскільки вони не є можливими! Проблему вирішує ввдення у перші два варіанти правила для визначення категорії спортсмена предикатів відсікання як останньої підцілі (для останнього варіанта правило відсікання вже не потрібне):

```
category (X,2):- winner(X,_), winner(_,X),!.
category (X,1):- winner(X,_),!.
category (X,3):- winner(_,X).
```

Знищення маркерів повернення приведе до отримання у відповідь на запит

category (marko,X).

єдиного правильного результату:

X=2.

Один результат буде повернено тепер і для запиту *category (petro,X).*

Загальним правилом для програміста має бути прибирання усіх непотрібних у програмі точок повернення якомога раніше.

3.4 Метод повторення, визначуваний користувачем

Отже, пошук з поверненням є можливим лише у тих випадках, коли у твердженнях є цілі, які можна передовести. Що ж робити, якщо для розв'язання задачі необхідний пошук з поверненням, але у програмі відсутні цілі, які можна передовести? У такому випадку точку повернення необхідно створити штучно! Для цього треба створити спеціальний предикат, метою якого є лише генерування точки повернення, без виконання жодних інших дій. Потрібний предикат можна визначити таким чином (предикат не є стандартним і його ім'я може бути вибрано абсолютно довільно):

```
repeat.
repeat:-repeat.
```

Оскільки перший *repeat* не створює підцілей і не має аргументів, він завжди є істинним. Але, при виконанні його як підцілі, маркер повернення викликає перехід програми на другий *repeat* (повернення до голови правила). Отже, правило *repeat* використовує само себе як компоненту. Його підціль завжди успішна, отже завжди успішним є і саме правило.

Предикат *repeat* завжди буде виконуватися успішно при спробі викликати його після повернення. Таким чином, *repeat* є рекурсивним правилом, яке ніколи не буває невдалим.

Прикладом використання методу повтору, визначуваного користувачем, може служити програма, що зчитує рядок, який вводиться з клавіатури, і дублює його на екрані:

```
domains
  name=symbol.
predicates
  repeat.
  do_echo.
  check(name).
clauses
  repeat.
  repeat:-repeat.
  do_echo:-
  repeat,readln(Name),write(Name),nl,check
(Name),!.
  check (stop):-nl,write ("OK!").
  check (_):-fail.
goal
  do_echo.
```

Якщо записати правило виведення без цілі *repeat*

do_echo:- readln (Name),write (Name),nl,check (name),!.

то буде виконано введення і виведення тільки першого і єдиного рядка, що не збігається зі «stop». Дійсно, після здіснення введення і виведення першого рядка буде виконано перевірку його відповідності зразку «stop», яка завершиться невдачею, внаслідок чого запуститься механізм пошуку з поверненням. Але оскільки жодну з цілей у правилі не можна передовести, програма не встановила жодного маркера повернення, отже, точки для початку повторних обчислень відсутні і виконання програми завершується невдачею.

Розглянемо тепер варіант правила з використанням цілі *repeat.* При першому доведенні цілі *repeat* вона успішно доводиться з використанням відповідного факту *repeat.* При цьому встановлюється маркер повернення. Отже, якщо у подальшому не буде доведено будь-яку ціль, то буде здійснено повернення до цілі *repeat* і передоведення її за допомогою правила *repeat:- repeat.*

Далі виконується послідовне успішне доведення усіх цілей правила, аж до цілі *check(Name).* У разі, якщо

$$Name \neq \text{stop},$$

варіант правила *check(_):-fail* генерує повернення до точки, визначеної маркером повернення. У даному прикладі цілі *readln(Name), write(Name), nl* – передовести неможливо, а предикат *repeat* визначено таким чином, що він завжди може бути передоведеним. Отже, ціль *repeat* успішно передоводиться (генеруючи точку повернення на саму себе) і відновляється природний порядок доведення цілей, зліва направо.

Зауважимо, що програма завершує роботу при введені рядка «stop». При виконанні умови *check*, повернення припиняються внаслідок використання предиката *cut* (*!*). Якщо значення введеного рядка відрізняється від «stop», кожного разу відбувається повернення до правила *repeat*. Завдяки тому, що правило *repeat* є компонентом правила *do_echo*, останнє стає правилом повтору.

Таким чином організовуються повторні дії за допомогою пошуку з поверненням у тому випадку, коли у базі знань не було цілей, які б забезпечували генерування точки повернення.

3.5 Проста рекурсія

В Турбо-Пролозі для виконання операцій, які повторюються, частіше за все використовують прийом, який має назву рекурсія (recursion). Рекурсивним називається правило, яке містить в собі само себе. Для того, щоб рекурсивний метод розв'язання задачі був результативним (тобто закінчувався), він повинен в кінцевому підсумку приводити до задачі, яка розв'язується безпосередньо. Відповідальність за забезпечення зупинки правила рекурсії покладається на програміста.

Прикладом правила рекурсії може служити таке:

```
write_string:-write("Hello!"),nl,
              write_string.
```

Така програма виводить повідомлення «Hello!», після чого здійснює виклик самої себе. Задля запобігання нескінченному зациклюванню до рекурсивного правила вводять предикат завершення, який містить умову виходу з рекурсії. У наведеному нижче прокладі програми, яка циклічно зчитує символи, що вводяться з клавіатури, таким предикатом є Char_data<>#:

```
Domains
  Char_data=char.
predicates
  read_a_character.
  write_prompt.
clauses
  write_prompt:-
      write("Введіть символ. Для завершення введіть #"),nl.
```

```
read_a_character:-
    readchar(Char_data),Char_data<>#,write(Char_data),
    read_a_character.
goal
  write_prompt,read_a_character.
```

Тут *readchar* є вбудованим предикатом Турбо-Прологу, що забезпечує зчитування одного символу. Наступне підправило перевіряє, чи не є введений символ решіткою «#». Якщо ні, то підціль успішна, символ виводиться на екран і здійснюється рекурсивний виклик *read_a_character*. Програма завершує свою роботу при введені символу «#».

3.6 Узагальнене правило рекурсії

У загальному вигляді правило рекурсії можна подати таким чином:

```
<ім'я правила рекурсії>:-
        <список предикатів>,                    (1)
        <предикат умови виходу>,                (2)
        <список предикатів>,                    (3)
        <ім'я правила рекурсії>,                (4)
         <список предикатів>.                   (5)
```

де,

1 - список предикатів, успіх або невдача яких не впливають на рекурсію;
2 - успіх або невдача предиката умови виходу дозволяє, відповідно, продовжити рекурсію або зупинити її;
3 - група предикатів аналогічних 1 (не впливає на рекурсію);
4 - саме рекурсивне правило, що викликає рекурсію;
5 - група предикатів, що отримують значення (за їх наявності), які розміщаються у стеку під час виконання рекурсії.

Розглянемо як приклад програму, що генерує цілі числа від 1 до 7:

```
domains
  number=integer.
predicates
  write_number(number).
clauses
  write_number(8).
  write_number(Number):-
        Number<8,
        write(Number),nl,
        Next_number=Number+1,
        write_number(Next_number).
goal
```

write("Here are the numbers:"), nl,write_number(1),nl.

write(""OK!).

Програма починається зі спроби обчислити цільове твердження *write_number(1)*. Спочатку вона зіставляє цільове твердження з першим правилом *write_number(8)*. Оскільки $1 \neq 8$, зіставлення виявляється неуспішним. Після цього відбувається успішне зіставлення з головою правила *write_number(Number)*, внаслідок чого змінна Number набуває значення 1. Далі відбувається порівняння 1 з 8 і, оскільки $1 \neq 8$, підціль виявляється успішною. Наступний предикат виводить значення Number на екран. Далі значення змінної Next_number збільшується на одиницю, після чого правило *write_number* викликає само себе з параметром рівним 2.

Зауважимо щодо можливості виклику правила з іменем змінної, що не збігається з тим, яке використовується в голові правила. Суттєвою при передачі значень є лише позиція в списку параметрів. Тепер цикл повторюється зі значенням параметра Number=2. Програма продовжує роботу доти, поки не виконається умова Number=8.

Часто для розв'язання задач використовують стратегію розбиття вихідної задачі на декілька більш простих підзадач. При цьому нова підзадача інколи буває зменшеним варіантом вихідної задачі і спосіб її подальшого розбиття і розв'язання ідентичний тому, який було застосовано до вихідної задачі.

3.7 Стратегія "розділяй та пануй"

Зі сказаного вище можна помітити, що рекурсія є одним з варіантів стратегії редукції задачі, тобто її розв'язування шляхом розбиття її на підзадачі. Така стратегія полягає в тому, щоб розбити початкову задачу на більш дрібні підзадачі, розв'язати їх, а потім об'єднати розв'язки підзадач для того, щоб отримати розв'язок початкової задачі. Це й є сутністю стратегії "розділяй і пануй". Нерідко доцільно продовжувати розбиття отриманих підзадач на ще більш дрібні з тим, щоб здійснити їх розв'язування по частинах.

У випадку, коли отримана підзадача є зменшеним варіантом вихідної задачі, спосіб її подальшого розбиття і розв'язування залишається ідентичним тому, що застосовувався до початкової задачі. Саме такий процес і називається рекурсією.

Слід особливо підкреслити, даний метод розв'язування задачі буде результативним лише у тому випадку, якщо він здатний привести до задачі, яка може бути розв'язана безпосередньо. Така задача називається "тривіальною" і описується твердженнями, які називаються граничними умовами.

Розглянемо приклад використання методу рекурсії для визначення родинного відношення *пращур*.

Визначимо в програмі в явному вигляді відношення *батько* між парами об'єктів. Крім того, будемо казати, що деякий X є пращуром деякого Z, якщо між X і Z існує ланцюжок осіб, які пов'язані між собою відношенням батько-дитина, наприклад так, як показано на рис. 10.

Це можно описати такими логічними виразами:

«Для будь-яких X і Z, X є пращуром Z, якщо існує Y такий, що X є батьком Y і Y є пращуром Z».

Відповідне речення Прологу буде мати такий вигляд:

пращур (X,Z):- батько (X,Y),пращур (Y,Z).

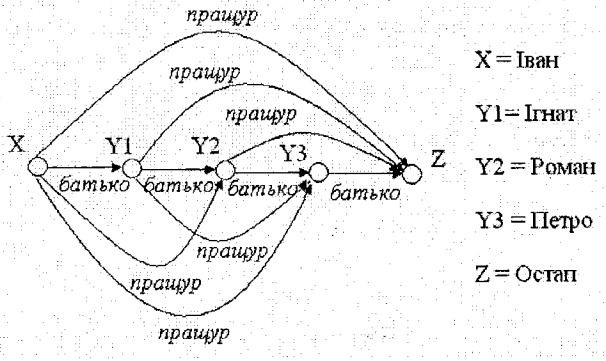

Рисунок 10 – Графічне подання відношень «батько» та «пращур»

Умовою зупинення тут буде поняття безпосереднього (найближчого) пращура, яке збігається з поняттям «батько». Тобто, правило виявлення безпосереднього пращура на Пролозі можно записати таким чином:

пращур (X,Z):-
батько (X,Z).

Умова зупинення виконається у тому випадку, якщо перший аргумент набуде значення імені батька, тобто найбільш близького пращура об'єкта, який визначає другий аргумент. Кожний крок рекурсії поступово наближатиме програму до умови зупинення, оскількі кожне рекурсивне звернення до відношення *пращур* видаляє один елемент з нашого «генеалогічного дерева». Таким чином рекурсивне правило для відношення *пращур* остаточно набуде такого вигляду:

```
пращур (X,Z):-
        батько (X,Z).      % умова зупинення
        пращур (X,Z):-
```

батько (X,Y),
пращур (Y,Z). % ім'я правила рекурсії

Нехай база знань містить такі факти:

батько (іван, ігнат),
батько (ігнат, роман),
батько (роман, петро),
батько (петро, остап).

Розглянемо порядок досягнення цілі Пролог-системою у відповідь на запит

пращур (Іван, Остап).

Перш за все програма спробує знайти в програмі речення, з якого безпосередньо випливає задана ціль. Речень, які описують відношення *пращур* в програмі, є два. Множина речень, які входять до складу одного і того ж відношення, називається процедурою.

Спочатку система випробовує речення, яке розміщено у програмі першим, що приведе до такої конкретизації змінних:

X1=Іван, Z1=Остап.

При цьому початкова ціль перетвориться на нову ціль:

батько (Іван,Остап).

Оскільки в базі знань відсутнє речення, голова якого може бути зіставлена з такою ціллю, ця ціль терпить невдачу і система здійснює перехід до другого твердження *пращур (X,Z)*, щоб спробувати інший варіант задовільнення цілі верхнього рівня *пращур (Іван, Остап).*

Знов змінним X і Z приписуються значення X2=Іван, Z2=Остап. Змінну Y ще не конкретизовано. Тепер верхня ціль *пращур(Іван, Остап)* замінюється на дві підцілі:

батько (Іван,Y),
пращур (Y,Остап).

Систама пробує досягнути ці цілі в тому порядку, в якому вони записані. Процес уніфікації (або зіставлення) першої підцілі з базою знань приводить до конкретизації змінної Y значенням

Y=Ігнат.

При цьому досягається перша підціль, а друга перетворюється на

пращур (Ігнат,Остап).

Для досягнення цієї цілі знов застосовується перше правило. Нове (чергове) застосування правила ніяк не пов'язано з його попереднім застосуванням. Тому система використовує нову множину змінних правила кожен раз, коли воно застосовується. Відобразим це,

переіменувавши змінні правила, після чого воно набуде вигляду:

працюр (X3, Z3):-
батько (X3, Z3).

Голова цього правила повинна відповідати нашій поточній цілі *працюр (Ігнат, Остап).* Поточна ціль замінюється на *батько (Ігнат,Остап).* Оскількі така ціль не задовольняється, виконується повернення до другого правила *працюр*, результатами чого стає конкретизація змінної Y2=Роман і рекурсивний виклик правила

працюр (Роман, Остап).

Цей крок рекурсії викликає такі конкретизації змінних:

X3=Роман, Y3=Петро, Z3=Остап.

Тобто, при виклику другого правила ціль *працюр (Роман. Остап)* замінюється на підціль *батько (Роман, Петро)* і

працюр (Петро, Остап).

При наступному рекурсивному виклику правила *працюр (X4,Z4)*, поточну ціль *працюр (Петро,Остап)* буде замінено на ціль

батько (Петро,Остап).

Така ціль негайно досягається, оскількі у базі знань знаходиться відповідний факт. На цьому кроці обчислення закінчується і на отриманий запит програма повертає позитивну відповідь -Yes.

3.8 Низхідна стратегія

Розглянемо приклад програми, що дозволяє обчислити значення n-го терма у такій послідовності:

1, 1, 2, 6, 24, 120, 720...

Проаналізувавши задачу, можна помітити, що значення її термів підкоряються такій закономірності:

- значення нульового терма дорівнює 1;
- значення першого терма можна отримати множенням 1× 1;
- значення другого терма можна отримати множенням 1× 1×2;
- значення третього терма можна отримати множенням 1× 1×2×3 ;

.....

- значення n-го терма можна отримати множенням 1×1×2×3 … × n;

З іншого боку, нульовий терм дорівнює 1, а n-й терм є (n-1)-м термом, помноженим на n. Дане визначення є рекурсивним, оскільки зводить розв'язання задачі обчислення значення n-го терма до розв'язання аналогічної, більш простої, задачі знаходження значення (n-1)-го терма з подальшим множенням його на n.

Подамо факт «N-й терм послідовності дорівнює V» предикатом

терм (N,V).

Тоді програма обчислення термів послідовності може бути такою:

```
терм (0,1),!.     % гранична умова - нульовий терм дорівнює 1,
терм (N,V):-  %  рекурсивна умова - якщо (N-1)-й терм дорівнює U,
M=N-1,            % то N-й терм дорівнює U× N.
терм (M,U),
V=U*N.
```

У відповідь на запитання, наприклад, *терм (3,X)*, програма поверне X=6.

Предикат відсікання *cut* служить першим твердженням для того, щоб виключити можливість зациклювання програми у випадку, якщо користувач помилково попросить її знайти інші можливі розв'язки.

Розглянемо докладніше роботу програми при обчисленні заданого в прикладі терма. Обчислення цього значення здійснюється програмою у два етапи: редукції задачі (розбивання задачі на підзадачі) і безпосереднього розв'язування задачі.

3.8.1 Фаза редукції задачі

При спробі задовольнити запит

терм (3,X).

Пролог намагається зіставити твердження запиту з першим твердженням програми

терм (0,1).

Ця спроба закінчується невдачею (оскільки перші аргументи тверджень є константами з різними значеннями) і Пролог переходить до зіставлення запиту із другим твердженням, головою правила

терм (N,V).

Перше рекурсивне звернення. Ця спроба зіставлення закінчується успіхом і змінні конкретизуються значеннями: N=3, V=X (V зв'язується з X). На наступному кроці виконується перша підціль правила: M набуває значення M=N-1=2. Друга підціль, яка набуває вигляду *терм(2,U)*, оскільки M набуло значення 2, викликає рекурсивне звернення до програми. При цьому розмірність другого запиту є меншою, у порівнянні з першим, оскільки перший аргумент зменшився на одиницю. Тому, що третя підціль виявилася незадоволеною, її узгодження відкладається доти, поки не буде погоджена друга підціль. На час очікування третя підціль заноситься до стека.

Друге рекурсивне звернення. Знову спроба погодити нове цільове

твердження

терм (2,U),

з першим твердженням *терм(0,1)* закінчується невдачею і Пролог зіставляє *терм(2,U)* з головою другого твердження

терм(N_2,V_2),

внаслідок чого змінні набувають значення N_2=2 і V_2=U (індекс 2 надамо змінним лише для того, щоб відрізнити друге звернення до твердження *терм* від першого; у загальному випадку індексом n змінної будемо позначати те, що вона використовується при n-ому зверненні, змінні при першому зверненні залишимо без індексів). Далі M_2 набуває значення M_2=1. Друга підціль *терм(1,U)* виклакає друге рекурсивне звернення. Узгодження третьої підцілі знов відкладається доти, поки не буде погоджена друга підціль (третя підціль знову заноситься до стека). Отже, при другому зверненні використовується те ж саме цільове твердження, що й при першому, але з меншим значенням аргументу N_2.

Третє рекурсивне звернення. Знову здійснюється спроба погодити цільове твердження, яке тепер набуло вигляду

терм(1,U_2).

з першим твердженням програми *терм(0,1)*. І знов спроба виявляється невдалою, але вдається погодити цільове твердження з головою другого твердження, що приводить до конкретизації змінних: N_3=1, V_3=U_2. Далі M_3 набуває значення M_3=0.

Зауважимо, що цього разу друга підціль

терм(0,U_3).

успішно погоджується з першим твердженням програми

терм (0,1).

При цьому U_3 набуває значення U_3=1, і Пролог накінець отримує змогу перейти до обчислення третьої підцілі.

На цьому редукція задачі завершується. Гранична умова дозволила успішно розв'язати останню (тривіальну) задачу. Зауважимо, що послідовністю рекурсивних викликів ми зменшили розмірність початкової задачі, звівши її до тривіальної задачі, розв'язок якої був нам відомий. З граничної умови ми отримали значення нульового терма, що дорівнює 1. Тепер Пролог послідовно намагатиметься узгодити треті підцілі, витягаючи їх у зворотному порядку зі стека. Якщо йому це вдасться, програма завершиться успіхом.

3.8.2 Фаза розв'язування задачі

Узгодження третьої підцілі, *терм(1,U_2)*. Оскільки на третьому рівні рекурсії змінні конкретизовані значеннями: V_3=U_2, U_3=1, N_3=1, третє цільове твердження

$$V_3=U_3\times N_3.$$

набуває вигляду:

$$U_2=1\times 1.$$

Отже, U_2 набуває значення $U_2=1$.

Узгодження третьої підцілі терм(2,U). У зв'язку з отриманою конкретизацією змінних на другому рівні рекурсії: $V_2=U$, $U_2=1$, $N_2=2$ Друге відкладене твердження

$$V_2=U_2\times N_2.$$

подає третю підціль при другому рекурсивному зверненні у вигляді:

$$U=1\times 2.$$

Отже, U набуває значення U=2.

Узгодження третьої підцілі терм(3,X). Внаслідок конкретизації змінних значеннями V=X, U=2, N=3, третя підціль

$$V=U\times N.$$

у першому рекурсивному зверненнні набуває вигляд:

$$X=2\times 3.$$

Таким чином, отримуємо остаточне значення X=6.

Оскільки цільові твердження зі стека вичерпані, Пролог виводить на екран результат роботи програми:

X=6.

3.8.3 Діаграма трасування запиту

Кращому розумінню механізму роботи рекурсивних запитів сприяє їх подання у графічному вигляді, яке здійснюється з використанням спеціальних діаграм трасування запитів (рис. 11). У лівій частині такої діаграми відображаєтьс фаза редукції задачі, а у правій - фаза її розв'язку. Цільові твердження розміщені у прямокутниках, а значення змінних, при яких узгоджуються цільові твердження, розміщаються у шестикутниках. Дуги, що виходять з цільового твердження, позначаються номером узгаджуваного твердження.

Таким чином, у рекурсивному описі дій необхідно звернути увагу на такі обставини:

- процедура повинна містити принаймні одну термінальну гілку й умову закінчення;
- коли процедура досягає рекурсивної гілки процес, що функціонує, призупиняється і запускається аналогічний новий процес, але вже на новому рівні. Параметри перерваного процесу запам'ятовуються у стеку, він переходить до режиму очікування і запускається знову лише після

закінчення нового процесу. У свою чергу і новий процес може призупинитися і перейти до очікування виконання ще більш нового процесу і. т. ін.

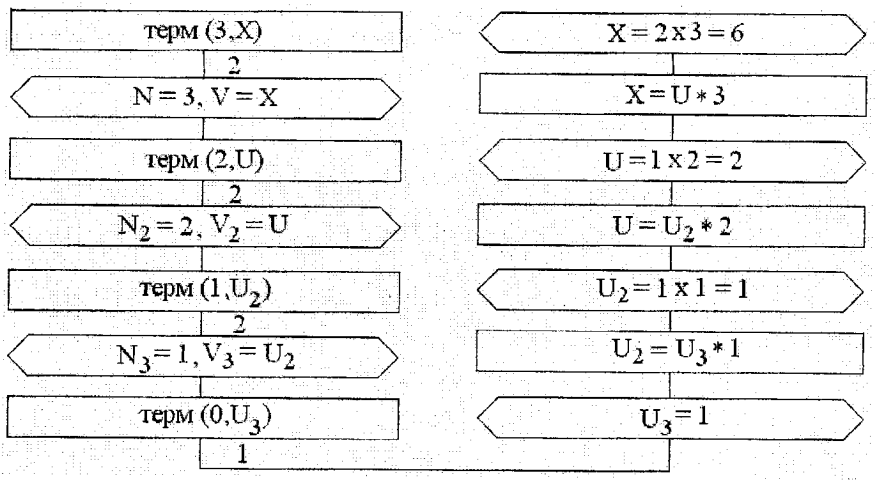

Рисунок 11 - Схема трасування рекурсивного запиту

Таким чином, при виконанні рекурсивної процедури утворюється стек перерваних процесів, з яких у дану мить часу виконується лише останній процес, після закінчення роботи якого, поновлює своє виконання попередній до нього процес. Повністю процес закінчується після того, як спорожніє стек, що означає завершення усіх перерваних процесів.

3.9 Висхідна стратегія

Ми розглянули процес, що полягає у розбитті задачі на підзадачі, з наступним їх розв'язанням. Наприклад, у розглянутому прикладі для доведення цільового твердження *терм (4,V)* необхідно виконати ланцюжок таких обчислень:

$$терм(3,V_2), терм(2,V_3), ..., терм(0,V_5).$$

і побудувати розв'язок, вибравши як базис граничну умову.

Інший підхід полягає в тому, щоб, почавши з граничної умови, будувати розв'язок доти, поки не буде вирішена поставлена задача.

Згідно з даним підходом, обчислення четвертого члена послідовності можна розпочати з граничної умови 1, потім обчислити значення першого терму 1×1, другого - 1×1×2, третього - 1×1×2×3, і, нарешті, четвертого 1×1×2×3×4. Така стратегія називається висхідною.

При використанні цього методу розв'язування цільове твердження повинне мати два додаткові параметри: один для визначення "розміру" розв'язаної на дану мить задачі, а другий - для запису проміжного результату. При виклику твердження параметри пов'язуються із граничною умовою.

Для розглянутого прикладу, програма, побудована за висхідною стратегією, може мати такий вигляд:

```
поточний_терм (N,V,N,V),!.
поточний_терм (NS,VS,N,V):-
    NS1=NS+1,
    VS1=VS*NS1,
поточний_терм (NS1,VS1,N,V).
```

Запит містить граничну умову, як розв'язану до теперішнього часу задачу. На запит

поточний_терм(0,1,7,V).

буде отримано відповідь V=5040.

Альтернативним варіантом є визначення ще одного твердження, яке маскує додаткові параметри у поточний_терм:

поточ_терм (N,V):- поточний_терм (0,1,N,V).

У цьому випадку на запит, наприклад,

поточний_терм (7,V),

буде отримано таку ж саму відповідь V=5040.

Контрольні запитання

1. У чому полягає принципова відмінність ітерації та рекурсії?
2. Наведіть узагальнений вигляд правила ітерації та дайте необхідні пояснення щодо його функціонування.
3. Які вбудовані предикати для управління ходом виконання логічної програми Вам відомі?
4. Що таке рекурсія?
5. Наведіть узагальнений вигляд правила рекурсії та дайте необхідні пояснення щодо його функціонування.
6. Що таке зовнішня та внутрішня цілі у Пролог-програмі? Які особливості виконання вони мають?
7. Що таке предикат *fail*, чи є він вбудованим (стандартним) предикатом Прологу і для чого він використовується?
8. Що таке предикат *cut*, чи є він вбудованим (стандартним) предикатом Прологу і для чого він використовується?
9. Що таке предикат *repeat*, чи є він вбудованим (стандартним) предикатом Прологу і для чого він використовується?

10. Що таке предикат *!*, чи є він вбудованим (стандартним) предикатом Прологу, для чого використовується і яке має інше позначення?
11. Який механізм Прологу і як саме забезпечує виконання функцій предикатом *cut*?
12. Який механізм Прологу і як саме забезпечує виконання функцій предикатом *fail*?
13. Наведіть приклад задачі, для розв'язування якої доцільно використати метод повернення після невдачі.
14. Які основні прийоми можуть використовуватися для забезпечення повернення після невдачі?
15. Поясніть сутність методу повтору, визначуваного користувачем? За рахунок чого забезпечується повернення?
16. Поясніть поняття «рівень рекурсії». Що відбувається на кожному новому рівні рекурсії?
17. Що таке «проста» рекурсія і якою ще може бути рекурсія?
18. Наведіть узагальнене правило рекурсії та поясніть функціональне призначення головних його компонентів.
19. Які основні дії виконуються рекурсивною програмою у фазі редукції задачі?
20. Які основні дії виконує рекурсивна програма у фазі розв'язування задачі?
21. Чи обов'язковою є фаза редукції задачі у рекурсивній програмі? Поясніть свою відповідь.
22. Чи обов'язковою є фаза розв'язування задачі у рекурсивній програмі? Поясніть свою відповідь.
23. Поясніть принципи побудови діаграми трасування рекурсивного запиту.
24. Що таке низхідна та висхідна стратегії побудови рекурсивної програми?
25. Наведіть порівняльний аналіз низхідної та висхідної стратегій побудови рекурсивної програми.
26. Які ознаки задачі визначають застосування низхідної або висхідної стратегії побудови рекурсивної програми?

Лабораторна робота № 3
СТВОРЕННЯ ІТЕРАЦІЙНИХ ТА РЕКУРСИВНИХ ПРОЦЕДУР У ТУРБО-ПРОЛОЗІ

Мета роботи

Набуття навичок побудови ітераційних та рекурсивних процедур мовою Турбо- Пролог.

Завдання на підготовку

Студент повинен знати:

- поняття та основні властивості ітерації та рекурсії;
- основні методи організації ітерацій;
- основні методи побудови рекурсивних процедур.

Студент повинен уміти:

- працювати в середовищі Турбо-Пролог;
- будувати діаграми трасування запитів при створенні рекурсивних процедур;
- виконувати трасування програми з використанням оператора trace.

Для допуску до виконання роботи необхідно:

- вміти відповісти на теоретичні питання по ходу виконання роботи;
- подати викладачу заготівку звіту про лабораторну роботу, що містить: титульний аркуш і попередній текст програми, з розробленими, відповідно до завдання, ітераційними та рекурсивними процедурами.

Завдання на лабораторну роботу

1. Доповнити розроблену в попередній лабораторній роботі програму правилами, що включають ітераційні (не менше двох) та рекурсивні (не менше трьох) визначення.

2. Отримати результати трасування двох рекурсивних запитів з використанням оператора trace та порівняти отримані результати із заздалегідь підготовленими двома діаграмами трасування запитів.

Оформлення роботи та порядок захисту

Наведіть у звіті:

- тексти розроблених ітераційних та рекурсивних визначень;
- графічне пояснення роботи ітераційних запитів;
- результати трасування рекурсивних запитів з використанням оператора trace;
- діаграмами трасування рекурсивних запитів.

При захисті роботи необхідно вміти пояснити кожен запис, відповіс-

ти на теоретичні питання та продемонструвати практичні навички з використання складених предикатів та операторів відсікання та повернення.

Запитання для самоконтролю

1. Модифікуйте програму про службовців компанії (С. 80) таким чином, щоб вона виводила перелік даних лише про службовців-чоловіків.
2. Модифікуйте програму про службовців компанії (С. 80) таким чином, щоб вона виводила перелік даних лише про робітників відділу AI.
3. Модифікуйте програму про службовців компанії (С. 80) таким чином, щоб вона виводила перелік даних лише про службовців з оплатою більшою за 5 доларів на годину.
4. Магазин взуття фіксує кількість взуття, проданого протягом дня. Дані про продажі містять відомості про індекс товару, його ціну, продану кількість екземплярів даного товару, розмір. Напишіть Пролог-програму, що формує звіт про продажі.
5. Поясніть механізм функціонування предиката відсікання.
6. Модифікуйте програму (С. 68), написану за методом повторення, визначеним користувачем, таким чином, щоб вона сприймала тільки цілі числа з клавіатури і дублювала їх на екрані. Програма має завершувати свою роботу при введені з клавіатури числа нуль (вбудованим предикатом Турбо-Прологу для зчитування цілих чисел з клавіатури є readin (Numder), де Number – ім'я змінної для цілих чисел).
7. Модифікуйте програму з попереднього завдання таким чином, щоб вона сприймала два десяткових числа з клавіатури та дублювала їх на екран. Потім примусьте програму обчислювати суму введених десяткових чисел і виводити цю суму на екран. Програма має завершувати свою роботу у разі, якщо одне з двох введених чисел виявиться нулем (вбудованим предикатом Турбо-Прологу для зчитування десяткових чисел з клавіатури є readreal (Numder), де Number – ім'я змінної для десяткових чисел).
8. Напишіть ітераційну та рекурсивну програму генерування усіх цілих чисел із заданого діапазону.
9. Напишіть рекурсивну програму генерування усіх цілих чисел із заданого діапазону в зворотному порядку.
10. Напишіть ітераційну та рекурсивну програму для додавання усіх чисел із заданого діапазону.
11. Напишіть рекурсивну програму для додавання усіх парних чисел із заданого діапазону.
12. Напишіть рекурсивну програму для обчислення факторіала введеного числа з використанням низхідної, а потім висхідної стратегії.
13. Напишіть рекурсивну програму для обчислення значень чисел Фібоначчі.

4 СПИСКОВІ СТРУКТУРИ

4.1 Нотація для списків

Списком (list) називається впорядкований набір об'єктів одного й того ж самого доменного типу, які розташовані один за одним. Об'єктами списку можуть бути цілі й дійсні числа, символи, символьні рядки й структури. Порядок розташування елементів є важливою характеристикою списку; одні і ті ж елементи, упорядковані іншим способом, є вже зовсім іншим списком. Об'єкти списку пов'язані між собою, тому з ними можна працювати і як з групою об'єктів (списком у цілому), і як з індивідуальними об'єктами (елементами списку).

Множина елементів списку розташовується у квадратних дужках, а один від одного елементи відділяються комами:

[1, 2, 3, 4],
[s, g, u, n, i],
["учора","сьогодні","завтра"].

Об'єкти списку називаються його елементами, а кількість елементів у списку - довжиною списку. Список може містити один елемент або навіть зовсім не містити жодного елемента (порожній список []).

Непорожній список прийнято розглядати як такий, що складається з двох частин:

- голови (перший елемент списку, який зазвичай позначають латинською буквою H, від англійського слова Head - голова);
- хвоста (весь список за винятком першого елемента, який зазвичай позначають латинською буквою T, від англійського слова Tail - хвіст).

Голова подає один елемент списку і є окремим неподільним значенням. Навпаки, хвіст є списоком, складеним з того, що залишилося від вихідного списку без його голови. Цей новий список, якщо він не є порожнім, можна далі й далі ділити на голову і хвіст, аж доки він не стане пустим. Навіть якщо список складається з одного елемента, його можна розділити на голову, яка й буде подаватися цим єдиним елементом, і хвіст, що буде в цьому випадку порожнім списком.

4.2 Графічне подання списків

Список можна подати графічно:

- лінійним графом;
- бінарним деревом.

Наприклад, твердження

number ([1, 2, 3, 4]),

об'єктом якого є список з чотирьох елементів, можна подати, відповідно, у вигляді ланцюжка

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5,$$

де напрямок стрілок показує порядок, в якому можна отримати доступ до відповідного елемента, або у вигляді дерева, наведеного на рис. 12.

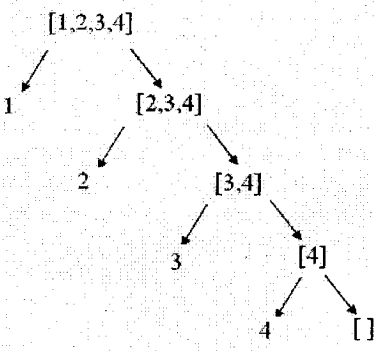

Рисунок 12 – Графічне подання списку у вигляді голови й хвоста

Внутрішні підпрограми уніфікації Прологу дотримуються задаваного цими графами напрямку впорядкування об'єктів. Бінарне дерево наочно інтерпретує процес повернення у вигляді перебору гілок дерева при пошуку заданого елемента списку.

4.3 Зіставлення списків

Елементи списків і самі списки як окремі елементи зіставляються відповідно до загальних правил зіставлення. У текстах програм список з головою H і хвостом T позначається, як правило, у вигляді [H|T]. На рис. 13 показано, що список, який має форму [H|T], зіставляється з будь-яким непустим списком, а також показані правила узгодження змінних H і T для різних ситуацій.

Зауважимо, що до голови списку можна розмістити більш ніж один елемент. Наприклад, запис [M, N|T] позначає будь-який список, що містить принаймні два елементи (якщо T - порожній список), а запис [M, N] - позначає будь-який список, що містить рівно два елементи.

4.4 Використання списків у програмі

Для використання списку в програмі необхідно описати предикат списку. Введення списків до програми відбивається на трьох її розділах:

domains, predicates, clauses або *goal.*

Розглянемо список, що подає назви птахів:

birds (["соловей", "зозуля", "ворона", "горобець")].

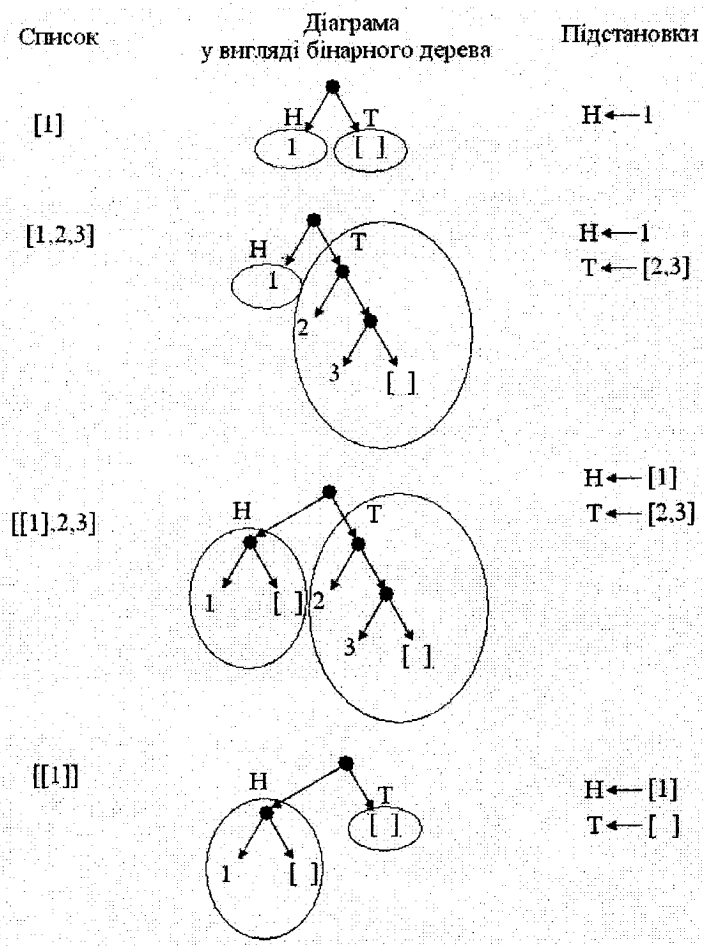

Рисунок 13 – Приклади зіставлення списків

Для опису списку в розділі доменів надамо йому ім'я *name_birds*. Позначення того, що даний номен належить до списків є наявність зірочки «*» після імені домена елементів.

Наприклад, запис

*bird_name**

вказує на те, що це домен списку, елементами якого є об'єкти типу *bird_name*. При цьому можна скористатися одним з двох таких описів у розділі domains:

*bird_list=bird_name**
bird_name=symbol.

або

*bird_list=symbol**

В останньому випадку домен *bird_list* є доменом списку елементів типу symbol (списку назв птахів).

У розділі *predicates* необхідно описати список з використанням імені предиката та взятого у круглі дужки імені домена:

birds (bird_list).

У розділі *clauses* список набуде такого вигляду:

birds(["соловей","зозуля","ворона","горобець")].

Розглянемо приклади запитів, за допомогою яких можна отримати доступ до елементів списку. Відповіддю на запит

birds(All).

буде

All= [соловей, зозуля, ворона, горобець).

У відповідь на запит:

birds([_,_,A,_]),

Пролог поверне значення змінної A:

A = ворона

Результатом запиту

birds([A,D,_,_]).

стане

A = соловей, D = зозуля.

Відзначимо, що змінна All подає весь список у цілому. На відміну від цього, другий і третій запити мають справу з окремими елементами списку.

4.5 Використання методу з поділом списку на голову й хвіст

Операція розбиття списку на голову H і хвіст T позначається за допомогою вертикальної риски [H|T]. Обробка списків зводиться до повторюваних кроків.

1. Здійснюється перевірка, чи не є список пустим. Якщо список пустий, роботу з ним закінчено. Якщо список не є пустим, здійснюється перехід до кроку 2.
2. Список розбивається на голову і хвіст.
3. Здійснюється необхідна обробка елемента, який подається головою хвоста.

4. Хвіст, отриманий у пункті 2, розглядається як новий список, який подається на крок 1.

Отже, за своєю природою список є рекурсивною структурою, а обробку списків доцільно здійснювати за допомогою рекурсивних процедур.

Рекурсивна процедура, що працює зі списками, мало чим відрізняється від «простої рекурсії». Основними вимогами до неї залишаються.

1. Необхідність визначення термінальної гілки, яка повертає результат, що стає основою для обчислення результатів «відкладених» у стеку рекурсивних викликів. У випадку рекурсії зі списками термінальною умовою у більшості випадків є пустий список, що свідчить про закінчення його обробки.
2. Визначення рекурсивної гілки, у якій процедура викликає саму себе. При цьому, внаслідок кожного рекурсивного виклику, обчислення повинні наближатися до термінальної умови. Ця вимога автоматично виконується при обробці на кожному кроці рекурсії голови списку, і рекурсивному виклику на наступному кроці його хвоста.

Зауважимо, що дуже часто буває надзвичайно важко детально уявити собі дію складної рекурсивної процедури. Але однією з перваг рекурсії є те, що їх можна писати на основі виконання загальних формальних правил, без чіткого уявлення порядку обислення окремої конкретної процедури.

Розглянемо приклади декількох нескладних рекурсивних програм, що працюють зі списками.

1. Програма друкування елементів списку:

```
/* Робота зі списком шляхом ділення його на голову й хвіст*/
                domains
                  number_list=integer*.
                predicates
                  print_list(number_list).
                clauses
                  print_list([]).
                  print_list([Head|tail]):-
                        write(Head),nl,
                        print_list(Tail).
        /*Кінець програми*/
```

Програма використовує для доступу до елементів списку рекурсивне правило print_list, другою підціллю якого є саме це правило.

Перше твердження print_list([]), є граничною умовою, яка означає, що рекурсивні виклики повинні бути припинені, коли список вичерпається (стане порожнім), тобто коли всі його елементи будуть вивe-

дені на друк.

Роботу процедури print_list розглянемо на прикладі запиту

print_list ([1, 2, 3]).

Спроба задовольнити запит зіставлення першого твердження завершиться невдачою, оскільки його об'єктом є порожній список. При спробі зіставлення з об'єктом другого твердження змінні конкретизуються такими значеннями:

Head=1, Tail=2, 3.

У результаті успішного виконання першої підцілі на друк буде виведене значення змінної Head - 1.

Далі, оскільки хвіст списку є сам по собі список, значення змінної Tail використовується як об'єкт рекурсивного виклику print_list (Tail). Оскільки Tail має значення [2, 3], зіставлення з першим твердженням знову не проходить, тоді як зіставлення з головою правила приведе до конкретизації змінних значеннями: Head = 2, Tail = 3. Процес буде повторюватися доти, поки хвіст списку не стане порожнім (змінна Tail не набуде значення []). При цьому, в результаті рекурсивного виклику *print_list (Tail)*, значення Tail буде відповідати об'єкту правила *print_list([])*. Оскільки цей варіант не має рекурсивних викликів, ціль вважається досягненою, і, таким чином, виконується умова закінчення рекурсії print_list.

2. Створимо процедуру для визначення належності елемента спискові.

Спочатку визначимо загальну ідею перевірки того, чи є деякий об'єкт елементом списку. Оскільки список подається головою (H) і хвостом (T), то перевірка належності елемента R списку L=[H|T] виконується дуже просто, відокремленням голови від списку і порівнянням її з потрібним елементом. Якщо порівняння виявиться невдалим, пошук продовжується у хвості списку. Ознакою наявності елемента у списку буде успішне доведення цільового твердження (при цьому Пролог поверне "Yes"). Якщо ж ціль доведено не буде (при цьому Пролог поверне "No"), значить шуканий елемент у списку відсутній. Отож, необхідно забезпечити перевірку двох умов.

Гранична умова (тривіальна задача, до якої може бути зведена початкова задача шляхом послідовного зменшення її розмірності) може бути сформульована таким чином:

« Об'єкт входить до складу списку, якщо він є головою списку»
(в одному з рекурсивних викликів),

або у вигляді твердження Прологу:

належить (Об'єкт, Список):-

Список=[Голова|Хвіст],
Голова=Об'єкт.

Відповідно рекурсивну умову можна подати таким твердженням: «Об'єкт входить до складу списку, якщо він знаходиться у хвості списку»

(за умов, що об'єкт не є головою списку),

або у вигляді твердження Прологу:

```
належить(R,L):-
    L=[H|T],
    належить (Об'єкт, Хвіст).
```

Поєднавши перевірку двох умов, отримуємо:

```
належить (R,L):-
   L=[H|T],
   H=R.
належить (R,L):-
   L=[H|T],
   належить (R,T).
```

Підціль L=[H|T] служить тут для поділу списку на голову і хвіст. Але, враховуючи, що Пролог спочатку зіставляє з ціллю голову твердження, а лише потім намагається погодити його тіло, дану процедуру можна переписати в спрощеному вигляді:

```
належить (R,[R|T]).   % гранична умова: якщо голова списку і елемент є
                      % одним об'єктом R, то ціль задовольняється,
                      % обчислення припиняються і Пролог повертає
                      % значення "Yes".
належить              % рекурсивний виклик хвоста списку (у разі
(R,[H|T]):-           % неузгодження першої підцілі) , для подальшої
належить (R,T).       % перевірки входження елемента R до списку
```

З урахуванням того, що у першому правилі нас не цікавіть значення хвоста, а у другому – значення голови списку, осточно можемо записати:

```
domains
lisl=symbol*.
predicates
member(symbol, list).
clauses
member (R,[R|_]).
member (R,[_|T]:- member (R, T)).
```

3. На основі предиката *належить/2* (де 2 - арність предиката), можна побудувати процедуру *загальний_елемент/3*. Викликана як мета вона

успішно узгодиться, якщо два списки мають загальний елемент, яким і буде третій аргумент даного предиката:

загальний_елемент (L1,L2,E):-
належить (E,L1),
належить (E,L2).

Перша підціль даного правила послідовно генерує елементи першого списку, а друга - перевіряє, чи втримуються вони в другому списку.

Отже, при побудові процедур ми виконували певну послідовність кроків:

- встановили, що існує лише два можливі співвідношення для даного об'єкта й списку;
- описали ці можливі стани за допомогою зразків аргументів: або R і [R|T], або R і [H|T];
- специфікували процеси, що забезпечують той або інший можливий стан: у першому випадку ціль узгодиться негайно, а в другому – в разі узгодження рекурсивного виклику.

4. Побудуємо процедуру *double(L,*M) яка, отримавши вхідний список, наприклад L=[a, b, c], поверне список M, в якому дублюється кожен елемент вхідного списку M=[a, a, b, b, c, c].

Сформулюємо граничну умову, тобто, визначимо, що має повернути програма при наявності на вході пустого списку. Очевидно, якщо продублювати елементи пустого списку, то ми й отримаємо пустий список:

double ([],[]).

Визначимо тепер рекурсивну гілку. Її задачею є відокремити від вхідного списку голову і додати в голову результуючого списку два елементи, що відповідають голові вхідного списку, після чого здійснити рекурсивний виклик процедури, подавши на її вхід хвіст списку з попереднього виклику:

double ([L, M]):-
L=[H|T],
M=[H,H|TM],
double ([TL], [TM]).

Отже, остаточно отримуємо:

double ([],[]).
double ([H|T],[H,H|TM]):-
double ([TL], [TM]).

В останньому варіанти використано «скорочений» запис процедури, коли на місцях аргументів замість імен списків одразу записуються їх

структури.

На рис.14 наведено діаграму трасування рекурсивного запиту *double ([a, b, c], M).*

З рисунка можна побачити, що фаза редукції задачі зводиться до послідовного відокремлення голів вхідного списку, доки він не залишиться пустим. У фазі розв'язання задачі, розміщені у стеку елементи послідовно двічі вставляються у голову результуючого списку, формуючи результат виконання програми.

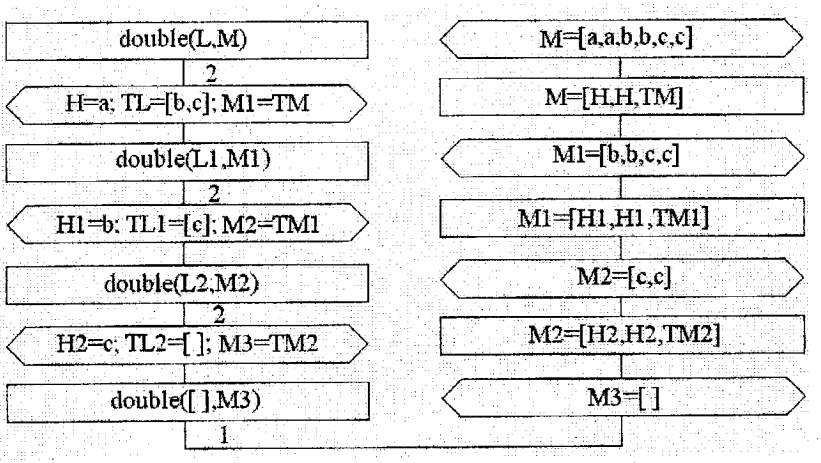

Рисунок 14- Діаграма трасування рекурсивного запиту *double ([a, b, c], M)*

5. Спробуємо тепер розв'язати задачу створення процедури розміщення заданого на вході елемента в останню позицію вхідного списку:

addlast (X, L, M).

Для побудови граничної умови знов з'ясовуємо, що ми отримуємо, якщо додамо об'єкт до останньої позиції пустого списку. Очевидно, що ми отримаємо список з одного цього об'єкта:

addlast (X, [ ], [ ]).

В основу рекурсивної гілки покладається найпростіша ідея: у кожному рекурсивному виклику треба просто відокремлювати від списку голову і передавати її у результуючий список!

addlast(X,[H|TL],[H|TM]):-.

addlast (X,TL,TM).

Таким чином, ми:

а) послідовно занесемо до стека всі елементи вхідного списку;

б) коли вхідний список стане пустим, занесемо до нього вхідний елемент;
в) послідовно занесемо в голову результуючого списку елементи зі стека.

Зауважимо, що елементи вхідного списку будуть забиратися зі стека і додаватися до голови результуючого списку у зворотному порядку (від останнього до першого), внаслідок чого порядок слідування елементів результуючого списку збережеться відносно порядку слідування елементів вхідного списку.

4.6 Метод аналізу станів

Сформулювати тепер загальну технологію розробки рекурсивної процедури, що отримала назву методу аналізу станів:

1) виявити можливі стани вхідних аргументів процедури. Зазвичай буде мати деяке твердження для кожного стану процедури (таких станів може бути й значно більше двох);

2) встановити, як саме розпізнається кожний стан:

- зразком у голові твердження, що подає даний стан;
- підціллю з тіла твердження, яка викликається в першу чергу;

3) специфікувати процес, що забезпечує даний стан. Сюди ж включається й специфікація форми результуючих аргументів.

Існують два великі класи станів, і кожна процедура повинна мати твердження принаймні для одного стану з кожного класу:

- Базовий стан (гранична умова).

У цьому стані відсутня наступна рекурсія. Якщо вихідні стани будуються за допомогою послідовних підстановок, то виходом для кожного з них служить константний терм, який завершує підстановку.

- Рекурсивний стан.

У цьому стані твердження специфікує деяку підстановку, яка повинна бути застосована до вихідного результуючого аргументу, і містить у собі рекурсивний виклик. Рекурсивний виклик повинен на кожному наступному кроці наближати базовий стан.

4) установити порядок розташування тверджень. Першими розмістити твердження, що відносяться до базового стану (граничних умов). Вони визначають завершення процесу, коли ніякі інші стани не можуть бути застосовані.

5) установити, чи не є стани взаємовиключними. Якщо визначення предиката забезпечує єдиний результат, необхідно переконатися, що кожний можливий вхід дає відмову тільки в одному зі станів, який необхідно виявити. В іншому випадку Пролог може дати при поверненні неправильні результати.

Доцільною є також перевірка того, чи наближає рекурсивний виклик досягнення базового стану, для чого слід порівняти аргументи, задані в голові твердження, з аргументами, які використовуються в рекурсивному

виклику. Наприклад, у другому твердженні процедури для предиката належить/2 другим аргументом голови є структура [H|T], а в рекурсивному виклику - змінна T. Тобто, при кожному рекурсивному виклику новий список буде коротшим за попередній. Рекурсивні виклики успішно узгоджуються при зіставленні підцілі з першим твердженням або дають відмови, якщо виникає порожній список.

Зазвичай стан з порожнім вхідним списком розглядається першим, оскільки він є базовим станом.

Розглянемо приклад побудови процедури для предиката conc/3, що визначає конкатенацію (об'єднання) двох списків. Списки, що об'єднуються, задаються у вигляді двох перших аргументів L і M, а результуючий список подається третім аргументом N. Вихідний список містить послідовність елементів першого вхідного списку, за якими послідовно розташовуються елементи другого вхідного списку.

З умови задачі можна визначити два основні стани, що залежать від того, чи має перший список L форму [HL|TL] або він є порожнім списком. Ці стани повинні бути подані, щонайменше двома відповідними твердженнями процедури.

Стан будемо виявляти за допомогою зразків у головах цих двох тверджень.

Базовий стан (гранична умова) описує приєднання порожнього списку до списку L, наслідком чого буде, очевидно, список L, тобто в базовому стані процедура просто повертає другий список:

conc ([], L, L).

Рекурсивний стан складається з послідовних підстановок, врахочуючи, що голова першого вхідного списку HL повинна бути головою результуючого списку HN = HL. Хвіст же TL першого списку передається рекурсивному виклику для побудови хвоста вихідного списку TN:

```
conc ([HL|TL],M1,[HN|TN]):-
conc (TL, M, TN).
```

Приєднання другого вхідного списку M до кінця першого вхідного списку L виконується таким чином:

а) перший список L поелементно пересилається до стека;

б) другий список M повністю передається у результуючий список N;

в) до голови третього списку N почергово додаються зі стека елементи першого списку L, як це мало місце у попередніх прикладах.

Для кращого розуміння принципу побудови процедури запишемо її друге твердження в деталізованому вигляді:

```
conc (([],L,L)).  % гранична умова
conc (L, M, N):-
L=[HL|TL],      % передача елементів списку L до стека
conc(TL,M,TN), % рекурсивний виклик з хвостом списку L
```

N=[HN|TN]. % приєднаня елементів зі стека у голову
% результуючого списку

Зауважимо, що при такому записі предикати, розташовані до рекурсивного виклику, забезпечують занесення даних до стека, а предикати, розташовані після рекурсивного виклику, будують розв'язок задачі з даних, що надходять зі стека.

Тепер принцип роботи процедури стає очевидним. У фазі редукції задачі, внаслідок виконання першої підцілі кожного рекурсивного виклику, відбувається послідовне відділення від вхідного списку L його голови (першого на дану мить часу елемента списку L), значення якої заноситься до стека. В результаті кожний наступний рекурсивний виклик процедури *conc* застосовується до зменшеного на один елемент вхідного списку. Це триває доти, поки вхідний список не стане порожнім. При цьому успішно узгоджується друга підціль правила внаслідок задоволення першого твердження процедури *conc([], L, L)* і вихідний список N набуде значення другого вхідного списку M. Після цього Пролог перейде до фази розв'язування задачі, що полягає в спробі задоволення третіх підцілей, шляхом послідовного отримання зі стека раніше занесених туди значень елементів першого списку L і приєднання їх як голови до результуючого списку N (який на початку фази розв'язання задачі повністю збігається з другим вхідним списком M).

На рис.15 наведено діаграму трасування запиту *conc([1, 2, 3],[4, 5], L2).*

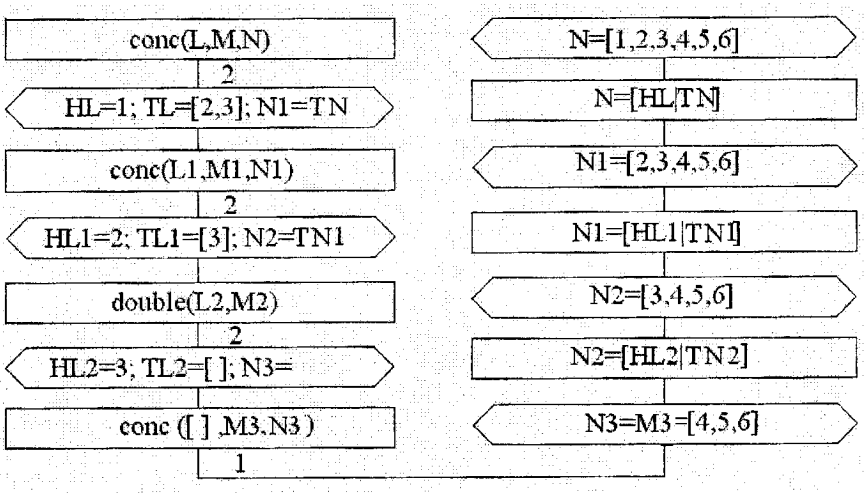

Рисунок 15 – Діаграма трасування запиту *conc ([1, 2, 3],[4, 5], L2)*

4.7 Метод вхідної рекурсії

У наведених прикладах ми розглядали роботу з елементами списку в напрямку від голови до хвоста (використовуючи послідовну підстановку). Але що робити якщо необхідно обробляти список у зворотному напрямку, наприклад, для побудови предиката реверс/2, який виглядає як другий аргумент об'єкта списку, яким є перший аргумент, розташований у зворотному напрямку?

У процедурних мовах програмування існує можливість використання допоміжних змінних, у яких можна зберігати проміжні результати. У логічному програмуванні змінні таким чином не використовуються. Проте відповідний механізм неважко здійснити, використовуючи допоміжний предикат, з додатковим аргументом.

Для цілей побудови рекурсивних структур, починаючи з константного терма і маркуючого кінця, використовується метод вхідної рекурсії. Він застосовується в тих випадках, коли для об'єктів структури запропонований особливий порядок, який не може бути забезпечений послідовною підстановкою, або коли в рекурсивних станах розрахунки залежать від наявного доступу до побудованої на теперішній момент структури.

Як приклад використання методу розглянемо побудову предиката реверс/2:

```
реверс (L1, L2):-
доп_реверс (L1, [], L2).
доп_реверс ([], L, L).
доп_реверс ([H|T], L1, L2):-
доп_реверс (T, [H|L1], L2.
```

Тут для виконання звертання списку використовується допоміжний предикат доп_реверс/3 з додатковим аргументом. При виклику предиката доп_реверс додатковий аргумент являє собою порожній список. На кожному рівні рекурсії один елемент першого списку міститься в цьому аргументі як голова. Третій аргумент просто передається на кожний новий рівень рекурсії доти, поки не буде досягнутий базовий стан, після досягнення якого його значенням стає значення другого аргументу.

Таким чином, метод вхідної рекурсії включає такі основні положення:

- константний терм є аргументом у викликах процедури;
- аргументом кожного рекурсивного виклику є нова структура, одним з компонентів якої служить попередньо побудована структура;
- змінна, призначена для розташування в неї остаточного результату, передається "внутрішнім образом" і до неї не застосовується ніяких підстановок, поки не буде досягнутий базовий стан;

- у базовому стані зазначена змінна зіставляється з аргументом, що означає побудовану структуру.

Контрольні запитання

1. Які графічні подання спискової літератури Ви знаєте. Наведіть власні приклади.
2. Як можна отримати доступ до конкретного (наприклад третього) елемента списка?
3. Як описується список у розділі доменів?
4. Які особливості має рекурсивна процедура, що використовує списки?
5. Наведіть діаграму трасування для довільної процедури, що використовує списки.
6. Охарактеризуйте основні положення методу аналізу станів для побудови рекурсивної процедури.
7. Наведіть приклади формування граничної умови для кількох рекурсивних процедур, що використовують списки.
8. Що таке вхідна рекурсія?

Лабораторна робота № 4
СТВОРЕННЯ РЕКУРСИВНИХ ПРОЦЕДУР ЗІ СПИСКАМИ В ТУРБО-ПРОЛОЗІ

Мета роботи

Вивчення методів організації даних у вигляді списків і розробки процедур обробки списків у Турбо-Пролозі. Набуття практичних навичок програмування операцій над списками.

Завдання на підготовку

Студент повинен знати:

- основні методи організації повторюваних операцій;
- методи організації рекурсії;
- списокову форму запису в Турбо-Пролозі;
- основні операції над списками.

Студент повинен уміти:

- працювати в середовищі Турбо-Пролог;
- будувати діаграми трасування запиту при використанні рекурсивних правил і виконанні операцій над списками;
- виконувати трасування програми з використанням оператора trace;
- пояснити роботу всіх програм і предикатів, наведених як приклади у теоретичному вступі до роботи.

Для допуску до виконання роботи необхідно:

- вміти відповісти на теоретичні питання по ходу виконання роботи;
- подати викладачу заготовку звіту про лабораторну роботу, що включає: титульний аркуш і попередній текст програми, з розробленими, відповідно до завдання, процедурами обробки списків.

Завдання на лабораторну роботу

1. Доповніть розроблену в попередній лабораторній роботі програму даними, організованими у вигляді списків, і процедурами обробки списків (не менше трьох процедур).

2. Розробіть програму, що реалізовує одну з операцій над списками (вибір операції здійснюється викладачем безпосередньо на лабораторній роботі).

Оформлення роботи та порядок захисту

Наведіть у звіті:

- тексти розроблених рекурсивних визначень;
- результати трасування рекурсивних запитів з використанням оператора trace;
- діаграми трасування рекурсивних запитів.

При захисті роботи необхідно вміти пояснити кожен запис, відповісти на теоретичні запитання та продемонструвати практичні навички з розробки рекурсивних процедур зі списками.

Деякі приклади рекурсивних процедур із списками:

- пошук елемента в списку;
- ділення списків;
- приєднання (конкатенація) списків;
- друк списку в зворотному порядку;
- визначення довжини списку;
- додання (видалення) елемента в список (із списку);
- перестановки елементів списку і так далі.

Запитання для самоконтролю

1. Дайте визначення рекурсивної процедури. Наведіть узагальнене правило рекурсії.
2. Що таке список? Наведіть форми подання списків.
3. Які з поданих нижче пар термів зіставляються? Для пар, що зіставляються, наведіть відповідні підстановки:
   1) [[сірий, зелений], чорний, голубий] і [H|T];
   2) [[георгій, марія]] і [H|[]];
   3) [[уільям, мері]|Інші] і [Перший,Другий];
   4) [дім, осел, кінь] і [H,T];
   5) [[брак (георгій, марія)]] і [A];
   6) [[1805], 1815] і [[A|B], C|Д];
   7) [[1805],1815] і [A|B,C|Д];
   8) [петро, іван] і [A,B|C];
   9) [[петро, іван]] і [[A],B|C].
4. Визначте відношення *останній (Елемент,Список)* таким чином, щоб *Елемент* був останнім елементом списку *Список*.
5. Визначте відношення *обернення (Список,Обернений Список)*, яке здійснює обернення списків. Наприклад, *обернення ([a,b,c,d],[d,c,b,a])*.
6. Визначте процедуру *паліндром(Список)*. *Список* називається паліндромом, якщо він читається однаково як зліва направо, так і справа наліво. Наприклад [мадам].

7. Визначте відношення *зсунути(Список1, Список2)* таким чином, щоб *Список2* являв собою *Список 1*, "циклічно зсунутий" вліво на один символ. Наприклад, запит *? – зсунути ([1, 2, 3, 4], L1)*, для відношення *зсунути (L1,L2)* має повернути *L1=[1,2,3,4] L2=[2,3,4,1]*.
8. Визначте відношення *переклад(Список1, Список2)* для перекладу списку чисел від 0 до 9 у список відповідних слів. Наприклад, *переклад([3,5,1,3],[три, п'ять, один, три])*. При написанні програми скористайтеся допоміжним відношенням *означає (0, нуль), означає (1, один)* і т. ін.

Нехай визначено предикат *ланцюжок_звань/2*:

наступне_звання (рядовий, сержант),
наступне_звання (рядовий, лейтенант),
наступне_звання (лейтенант, капітан),
наступне_звання (капітан, майор),
наступне_звання (майор, полковник),
наступне_звання (полковник, генерал).

9. Сформуйте у вигляді виклику предиката ланцюжок_звань/2 такі запити:
   - У яких військових звання є меншим не менше ніж на два чини відносно найвищого?
   - У яких військових звання є нижчим рівно на два чини порівняно з найвищим?

Задано базу даних на Пролозі:
військовий (прізвище (павлов), звання (генерал)),
військовий (прізвище (бойко), звання (полковник)),
військовий (прізвище (луцюк), звання (полковник)),
військовий (прізвище (риндюк), звання (сержант)),
військовий (прізвище (каськів), звання (сержант)),
військовий (прізвище (зінько), звання (капітан)),
військовий (прізвище (клічко), звання (лейтенант)),
військовий (прізвище (яровенко), звання (капітан)),
військовий (прізвище (поліщук), звання (майор)).

10. Визначте процедуру для предиката командує/2, першим аргументом якого є прізвище військового, та який повертає список звань в порядку убування тих, ким цей військовий може командувати.
11. Визначте процедуру для предиката відносне_звання/3, першими двома аргументами якого є прізвища двох військових, та яка повертає у третьому аргументі список чинів, які має отримати перший з них, щоб дослужитися до звання другого.
12. Визначте іншу процедуру для завдання з п.10, яка повертає список звань у порядку їх зростання.
13. Визначте процедуру для предиката *розбиття_списку (Список, Список1, Список2)*, який по черзі розподіляє елементи *Списку* між

двома списками *Список 1* і *Список 2*, і щоб довжина цих списків відрізнялася не більш ніж на одиницю. Наприклад розбиття_списку([a, b, c, d, e], [a, c, e], [b, d]).

14. Нехай задано список структур "клієнт":
[клієнт (a, 29, 3),клієнт (b, 29, 6), клієнт (C.40, 2)].
15. Першим аргументом кожної структури є ім'я клієнта, другим – добовий тариф, а третим – кількість днів, на яку взято на прокат автомобіль. Напишіть правило, яке обчислює загальну суму оплати всіма клієнтами, дані про яких містяться у списку. Визначте процедуру для предиката вилучити/3, першими двома аргументами якого є об'єкт та список. Процедура має перевірити, чи знаходиться заданий об'єкт у списку і повернути у третьому аргументі список з вилученими з нього об'єктами, заданими у першому аргументі.
16. Визначте процедуру для предиката реверс/2, першим аргументом якого є список, який повертає у другому аргументі обернений, відносно вхідного, список.
17. Визначте процедуру для предиката зглажування/2, яка повертає у другому аргументі список, з розміщеними на одному рівні з елементами всіх підсписків вхідного списку, отриманого у першому аргументі. Наприклад, *зглажування ([a, b, [c, d],[],[[[e]]], f], L)* → *L=[a, b, c, d, e,f]*.
18. Визначте процедуру для предиката максимум/2, який повертає у другому аргументі максимальний елемент списку, заданого у першому аргументі.
19. Визначте процедуру для предиката довжина/2, який повертає у другому аргументі довжину списку, заданого у першому аргументі.
20. Визначте процедуру для предиката *бульбашкове_сортування/2*, який повертає у другому аргументі відсортований у порядку зростання список, заданий у першому аргументі. Ідея бульбашкового сортування полягає у порівнянні на кожному наступному кроці двох сусідніх елементів списку. Якщо виявляється, що вони розташовані «неправильно», тобто попередній елемент є меншим за наступний, то вони міняються місцями. Процес продовжується доти, доки присутні «неправильно» розташовані пари, відсутність яких і зупиняє рекурсію. Для розв'язання задачі доцільно скористуватися допоміжним предикатом *перестановка/2*, який має порівнювати два перших елементи списку і, в разі необхідності, міняти їх місцями.
21. Визначте процедуру для предиката *сортування_вставленням/2*, який повертає у другому аргументі відсортований у порядку зростання список, заданий у першому аргументі. Ідея сортування вставленням полягає у тому, що коли хвіст списку вже відсортовано, достатньо розмістити перший елемент списку на відповідне місце у його хвості, і весь список буде відсортований. Для розв'язування задачі доцільно

скористуватися допоміжним предикатом *вставлення/2*, який, власне, розміщує перший елемент списку на відповідне місце у його хвості.

22. Визначте процедуру для предиката *сортування_вибором/2*, який повертає у другому аргументі відсортований у порядку зростання список, заданий у першому аргументі. Ідея сортування вибором полягає у тому, що у списку виявляється мінімальний елемент, який вилучається зі списку і після сортування списку, що залишився, вставляється в його голову. Для розв'язування задачі доцільно скористуватися допоміжними предикатами *мінімальний_елемент/2*, який виявляє мінімальний елемент списку, та *вилучити_елемент/2*, який вилучає заданий першим аргументом елемент зі списку, заданого другим елементом.
23. Визначте процедуру для предиката *швидке_сортування/2*, який повертає у другому аргументі відсортований у порядку зростання список, заданий у першому аргументі. Ідея швидкого сортування у виборі деякого «бар'єрного» елемента, відносно якого початковий список розбивається на два підсписки, в одному з яких розміщаються елементи менші за «бар'єр», а у другому – більші або рівні. Кожен з нових списків сортується так саме, після чого списку з елементів менших за бар'єрний приписують спочатку сам бар'єрний елемент, а потім – список елементів не менших за бар'єрний. Внаслідок цього отримуємо список з елементів, розташованих у «правильному» порядку. Для розв'язування задачі доцільно скористуватися допоміжними предикатами: *розподілом/4*- першим аргументом якого є бар'єрний елемент, другим – початковий список, третім та четвертим – результуючі списки, рекурсивними викликами яких початковий список розбивається на множину елементарних списків; та *конкатенацією/3*, першим аргументом якої є відсортований перший підсписок, другим – об'єднання бар'єрного елемента з другим відсортованим списком, а третім - результуючий список.
24. Визначте процедуру для предиката *сортування_злиттям/2*, який повертає у другому аргументі відсортований у порядку зростання список, заданий у першому аргументі. Ідея сортування злиттям полягає у розбитті початкового списку на два підсписки, після чого кожен з них впорядковується одним і тим самим способом, після чого отримані підсписки зливаються в один впорядкований список. Для розв'язування задачі доцільно скористуватися допоміжними предикатами: *розбиття/3*, який розбиває початковий список на два списки, *об'єднання_сортованих_списків/3*, який об'єднує два відсортованих списки у спільний відсортований список.
25. Визначте процедуру для предиката *відсортований_список/1*, який перевіряє, чи є поданий на вхід список відсортованим.

ГЛОСАРІЙ

AND/OR-tree – дерево І/АБО
anonymous variable - анонімна змінна
arc – дуга
atom – атом
attribute – атрибут
automated reasoning –автоматичне виведення
backtracking – механізм повернення
backward-chaining – зворотний ланцюжок виведення
body of clause – тіло речення
comparison – зіставлення, порівняння
conceptual model – концептуальна модель
conclusion – висновок
consequence – наслідок
constant – константа
constraint – обмеження
cut – пердикат відсікання
date-base – база даних
determinism – детермінізм
domain – домен
empty – пустий, порожній
external goal – зовнішня ціль
fact – факт
fail – невдача
false – хибний
functor – функтор
goal statement – цільове твердження
head of list – голова списку
implication – імплікація
inference rule – правило виведення
internal goal – внутрішня ціль
interpretation – інтерпретація
key attribute – ключовий атрибут
knowledge-base - база знань
list - список
logical term – логічний терм
nondeterminism – недетермінізм
object instance – екземпляр об'єкта
objects class – клас об'єктів
pattern comparison – зіставлення зі зразком
pointer – показчик
predicate – предикат

procedure – процедура
production rule – продукційне правило
recursion –рекурсія
refutation – спростування
relation - відношення
resolution inference rule – правило виведення резолюції
resolution proof procedure – доведення методом дозволу резолюції
recursive data structure – рекурсивна структура даних
rule - правило
state-space – простір станів
string – рядок
subject domain –предметна область
substitution – підстановка
tail of list – хвіст списку
term – терм
terminal vertex - термінальна вершина
true – істина
unification – уніфікація
union – об'єднання
variable – змінна
vertex – вершина

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Адаменко А. И., Кучуков А. М. Логическое програмированние и Visual Prolog. –Спб.: БХВ-Петербург, 2003.
2. Братко, Иван. Алгоритмы искусственного интеллекта на языке PROLOG, 3-е издание. : Пер. с англ. – М.: Издательский дом «Вильямс», 2004.
3. Братко Т. А. Программирование на языке Пролог для искусственного интеллекта: Пер. с англ. - М.: Мир, 1990. - 560 с.
4. Доорс Дж., Рейблейн А. Р., Вадера З. Пролог - язык программирования будущего: Пер. з англ. - М.: Финансы та статистика, 1990. - 144 с.
5. Ин Ц., Соломон Д. Использование Турбо-Пролога: Пер. с англ. - М.: Мир, 1993.- 608 с.
6. Клоксин У., Меллиш Д. Программирование на языке Пролог: Пер. с англ. - М.: Мир, 1987.- 336 с.
7. Малпас Дж. Реляционный язык программирования Пролог: Пер. с англ. – М.: Наука, 1990. - 463 с.
8. Марселлус Д. Программирование экспертных систем на Турбо-Прологе: Пер. с англ. – М.: Финанасы и статистика, 1994. – 238 с.
9. Месюра В. І. Функціональне та логічне програмування. Ч1. Основи функціонального програмування. – Вінниця: ВДТУ, 2001.
10. Методические указания по изучению языка Турбо-Пролог по курсам "Языки программирования" та "Диалоговые средства та машинная графика" для студентов всех форм обучения та слушателей МИПК/ Составители О. Ф. Цурин, В. Рот. - Киев: УМК МинВуза УССР, 1989. - 86с.
11. Основы программирования на языке Пролог : курс лекций : учеб. пособие для студентов вузов, обучающихся по специальностям в обл. информ. технологий / В. А. Шрайнер. – М.: Интернет – Ун-т Информ. Технологий, 2005. – 176 с.
12. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог: Пер. с англ. - М.: Мир, 1990. - 235 с.
13. Стобо Д. Ж. Язык программирования Пролог: Пер. з англ. - М.: Радио та связь, 1993. - 368 с.
14. Хоггер К. Введение в логическое программирование: Пер. с англ. - М.: Мир, 1983 - 384 с.
15. Цуканова Н. И., Дмитриева Т. А. Логическое программирование на языке Visual Prolog. Учебное пособие для вузов. – М.: Горячая линия. – Телеком, 2008. – 144 с.
16. Янсон А. Турбо-Пролог в сжатом изложении: Пер. с нем. - М.: Мир, 1991. - 94 с.

Додаток А

СИСТЕМНІ ВІКНА ТУРБО-ПРОЛОГУ

Система Турбо-Пролог включає головне меню з декількома підменю, що спускаються, чотири вікна та рядок допомоги, як це показано на рис. А.1

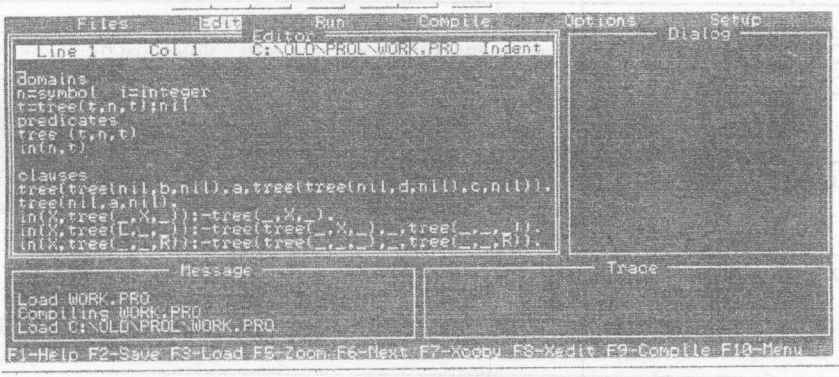

Рисунок А.1 - Системні вікна Турбо-Прологу

Системними вікнами Турбо-Прологу є: вікно редагування (Edit), вікно трасування (Trace), вікно повідомлень (Message) та діалогове вікно (Dialog). Крім цього є вікно допоміжного редактора (Auxiliary).

Вікно редагування

Для того, щоб виправити або створити файл в Турбо-Пролозі, необхідно "знаходитись" у вікні Edit (що аналогічно поняттю "знаходитись в редакторі"). Існує дві можливості "потрапити" в це вікно:

- обрати з головного меню пункт Edit;
- скористатись (практично з будь-якого місця системи Турбо-Пролога) "гарячою" клавішою Alt-E.

Потрапивши у вікно Edit, зверніть увагу на його рамки: Турбо-Пролог завжди фіксує поточне вікно подвійними рамками.

Якщо попередньо в редактор не був підвантажений якийсь файл, то Турбо-Пролог відкриває вікно редагування, створюючи при цьому новий (пустий) текстовий файл з іменем WORK.PRO.

Компоненти вікна редагування

Вікно редагування Турбо-Пролога складається безпосередньо з самого вікна, в якому можуть бачити та корегувати рядки вихідного тексту програми та татусні рядки, які надають інформацію про текст, що редагу-

ється.

Статусний рядок містить повне ім'я вихідного файла (будову, каталоги, ім'я файла та розширення), координати розташування курсора та поточні режими редагування (рисунок А.1).

Таблиця А.1 - Інформація, що міститься в рядку стану вікна редагування.

Статус	Опис
Line n (Рядок n)	Курсор знаходиться в рядку з номером n.
Col n (Колонка n)	Курсор знаходиться в колонці з номером n
C:path\filename.ext	Будова (C:), каталог (PATH\), ім'я (FILENAME) та розширення (.EXT) файла, який редагують в даний момент
Insert (Вставка)	Включений режим вставки; переключення цього режиму (включення/виключення) виконується за допомогою клавіші Insert або Ctrl+V.
Indent (Абзац)	Включений режим автоматичного формування відступу (абзацу). Переключення цього режиму виконується натиском послідовності клавіш Ctrl-Q I.
Text mode (Текстовий)	Включений текстовий режим (тобто, якщо поточний рядок виходить за межі екрана, весь текст пересувається в потрібний бік. Переключення цього режиму виконується натиском послідовності клавіш Ctrl-Q-W.

Рядок допомоги в останньому рядку екрана перелічує деякі можливі в редакторі команди та "гарячі" клавіші.

Для виходу з вікна редагування (для виклику головного меню), слід натиснути клавішу F10 або Esc. Редактор залишиться на екрані таким, що його можна бачити; все, що треба зробити для повернення в вихідний текст, - це натиснути в головному меню E (для Edit) або (практично, в будь-якому місці системи Турбо-Пролог) - Alt+E.

Вікно трасування

У вікні трасування (Trace) можно слідкувати за тим, як Турбо-Пролог виконує програму. Для цього необхідно відкомпілювати програму з включеною опцією трасування. Включити її можно або за допомогою меню Options/Compiler Directives, або включивши в вихідний текст директиви trace або shorttrace. В будь-який момент виконання програми можна перейти у режим трасування, натиснувши клавіші Ctrl+T.

Можна, натиснувши клавіші Alt+P, задати Турбо-Прологу режим

дублювання даних, які вводяться у вікно трасування, на принтері або в файлі (дана "гаряча" клавіша активізує спеціальне меню регистрації.)

Вікно повідомлень

В процесі роботи Турбо-Пролог виводить різні повідомлення у вікно під назвою Message (Повідомлення). Наприклад, Турбо-Пролог забезпечує:

- повідомлення про читання або зберігання файла;
- виведенення імен предикатів програми, що компілюється в даний момент;
- у випадку звернення до стандартного або глобального предиката з неприпустимим шаблоном потоку - відображення варіант потоку.

Діалогове вікно

Діалогове вікно є засобом середовища розробки, за допомогою якого ви передаєте в систему Турбо-Пролога свої запити. Для переходу в діалогове вікно просто натисніть Alt+R.

Якщо в даний момент в редакторі є текст програми, причому в ній відсутня секція goal, то Пролог спочатку скомпілює її (що необхідно для додавання в систему ваших предикатів), а потім активізує діалогове вікно.

Якщо ж в редакторі немає вихідного тексту програми, то Турбо-Пролог активізує діалогове вікно без будь-якої компіляції. Після цього надається можливість вводити по одному запиту (цілі) за один раз. Клавіша F8 використовується для виклику останньої з цілей, які були введені раніше.

Призначенням діалогового вікна, як засобу середовища розробки, є перевірка предикатів програми за допомогою відповідних запитів. Після забезпечення того, що введення потрібної цілі приводить до очікуваної реакції програми, можно включити цю ціль в програму та скомпілювати її у виконуваний файл.

Вікно допоміжного редагування (Xedit)

При знаходженні у допоміжному редакторі - (потрапити в нього можна з вікна редагування внаслідок натиснення клавіш F7 або F8), доступним стає цілковито інший набір команд редагування. У випадку Xcopy (копіювання ззовні) можна за допомогою клавіші F7 відмітити початок текстового блока, а з допомогою клавіші F10 (або знову ж F7) зафіксувати кінець блока та скопіювати його в попереднє вікно. У випадку Xedit (редагування допоміжного файла), клавіша F10 перед поверненням у вікно редагування запише на диск файл, який знаходиться у допоміжному редакторі (під іменем, яке ви самі задали), в той час, як клавіша Esc відмінить всі зроблені зміни та забезпечить повернення в попереднє вікно.

Вікно перегляду

Даний рядок допомоги з'являється при активізації редактора в режимі перегляду. Цей режим використовує вікно, яке з'являється у випадку, якщо вибрати з меню Editor Help (F1) варіант Show Help, якщо активізувати редактор, звертаючись до предиката display, або якщо викликати довгу форму предиката edit (з EditMode = 0). В режимі перегляду не можна корегувати текст, однак всі клавіші, які керуються курсором, працюють нормально.

Зміни розмірів вікон

При зміні розмірів вікон Турбо-Пролога, рядок допомоги підказує, які клавіші виконують такі функції. Інформація про зв'язок клавіш із функціями зміни розміру вікон наведена в таблиці 6.

Таблиця А. 2 - Клавіші змінення розмірів вікон

Функція	Клавіші
Стиснення/розширення обраного вікна з нормальною швидкістю	Клавіші зі стрілками
Стиснення/розширення обраного вікна з підвищеною швидкістю	Ctrl + стрілки
Пересування вікна вліво, вправо, вниз або вверх	Chift + стрілки
Розширення вікна до правого краю екрана	End
Розширення вікна до лівого краю екрана	Home
Пересування вікна до правого краю екрана	Chift + End
Пересування вікна до лівого краю екрана	Chift + Home
Пересування вікна до верхнього правого кута екрана	Ctrl + End
Пересування вікна до верхнього лівого кута екрана	Ctrl + Home
Вихід з режиму	F10

Існує декілька способів переходу до режиму змінення розмірів вікна:

- обрати в меню Setup команду Window Size;
- для роботи з поточним вікном - натиснути Chift-F10;
- натиснути клавішу F6, а потім використовувати спеціальні клавіші (після натиснення F6 в рядку допомоги буде ідентифіковуватися Next-наступний).

Додаток Б

ОСНОВНІ КОМАНДИ РЕДАКТОРА

Основні команди редагування програм Турбо-Прологу наведені в таблиці Б.1

Таблиця Б.1 - Основні команди редактора Турбо-Прологу

Команди редактора	Клавіші
Переміщення курсора	
В перший рядок вікна	Ctrl + Home
В перший рядок файла	Ctrl + PgUp або Ctrl + Q, R
На останій рядок вікна	Ctrl + End
На останій рядок файла	Ctrl + PgDn або Ctrl + Q, C
Вправо на одне слово	Ctrl + стрілка вправо або Ctrl + F
вправо в кінець рядка	End або Ctrl + Q, D
Вліво на одне слово	Ctrl + стрілка вліво або Ctrl + A
Вліво на початок рядка	Home або Ctrl + Q, S
Вставки та вилучення	
Вставка нового рядка	Ctrl + N
Вставка поміченого блока	Ctrl + F5 або Ctrl + K, C
Вилучення символа під курсором	Del або Ctrl + G
Вилучення символа зліва від курсора	Backspace або Ctrl + H
Вилучення символа вправо від курсора	Ctrl + T
Вилучення рядка під курсором	Ctrl + Backspace або Ctrl + Y
Вилучення тексту з початку рядка до курсора	Ctrl + Q, T
Вилучення тексту від курсора до кінця рядка	Ctrl + Q, Y
Маніпуляція блоками	
Фіксація початку блока	Ctrl + K, B
Фіксація кінця блока	Ctrl + K,
Фіксація (вставка) поміченого блока	Ctrl + F5 або Ctrl + K, C
Переміщення поміченого блока	Ctrl + K, V
Стирання поміченого блока	Ctrl + K, Y
Поновлення стертого блока	Ctrl + F7 або Ctrl + K, U
Запис поміченого блока в файл	Ctrl + K, W
Виведення поміченого блока на принтер	Ctrl + K, P
Пошук/Пошук та заміна	
Пошук	Ctrl + F3 або Ctrl + Q, F
Повторний пошук	Shift + F3 або Ctrl + O
Заміна	F4 або Ctrl + Q, L
Повторна заміна	Shift-F4 або Ctrl + L

Навчальне видання

Месюра Володимир Іванович
Лисак Наталія Володимирівна
Суприган Олена Іванівна

ФУНКЦІОНАЛЬНЕ ТА ЛОГІЧНЕ ПРОГРАМУВАННЯ

Частина 1

Логічне програмування мовою Пролог

Навчальний посібник

Редактор В. Дружиніна
Коректор З. Поліщук
Оригінал-макет підготовлено В. Месюрою

Підписано до друку 20.05.2011 р.
Формат 29,7×42¼ . Папір офсетний.
Гарнітура Times New Roman.
Друк різографічний. Ум. друк. арк. 6.6.
Наклад 75 прим. Зам. 2011-110

Вінницький національний технічний університет,
навчально-методичний відділ ВНТУ.
21021, м. Вінниця, Хмельницьке шосе, 95,
ВНТУ, к. 2201.
Тел. (0432) 59-87-36.
Свідоцтво субєкта видавничої справи
серія ДК № 3516 від 01.07.2009р.

Віддруковано у Вінницькому національному технічному університеті
В компютерному інформаційно-видавничому центрі.
21021, м. Вінниця, Хмельницьке шосе, 95,
ВНТУ, ГНК, к. 114.
Тел. (0432) 59-87-38.
Свідоцтво субєкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.