Міністерство освіти і науки України
Національний університет водного господарства та природокористування

В. М. Заяць, М. М. Заяць

ЛОГІЧНЕ І ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ. СИСТЕМНИЙ ПІДХІД

ПІДРУЧНИК

для студентів базового напряму підготовки
“Комп’ютерні науки”, “Комп’ютерна інженерія”
та “Програмна інженерія”

2-ге видання, виправлене та доповнене

Рівне

2018

УДК 681.142.2;004.432.42 (075.8)
З-411

Затверджено вченою радою Національного університету водного господарства та природокористування.
(*Протокол № 4 від 28 квітня 2017 р.*)

Рецензенти:
Мартинюк П.М., д-р. техн. наук, професор Національного університету водного господарства та природокористування, м. Рівне;
Пасічник В.В., д-р. техн. наук, професор Національного університету "Львівська політехніка";
Савула Я.Г., д-р. фіз.-мат. наук, професор Львівського Національного університету імені Івана Франка;
Яворський І.М., д-р. фіз.-мат. наук, професор Технологічно-природничого університету (UTP), м. Бидгощ (Польща).

Заяць В.М., Заяць М.М.
З-411 Логічне і функціональне програмування. Системний підхід. Підручник. – 2-ге видання, випр. та доповн. – Рівне : НУВГП, 2018. – 422 с.
ISBN 978-966-327-375-4

У підручнику подано теоретичні основи та програмні і технічні засоби логічного та функціонального програмування, ефективні методи і алгоритми побудови програм як мовою логіки, так в термінах функціоналів, підходи до реалізації найчастіше вживаних логічних та функціональних прикладних програм.

Викладений матеріалу проілюстровано достатньою кількістю прикладів та їх реалізацією в середовищі Турбо-Прологу та Авто-Ліспу, що сприятиме кращому засвоєнню матеріалу студентами та користувачами декларативних мов програмування.

Призначений для студентів, що навчаються за напрямом "Комп'ютерні науки", "Комп‘ютерна інженерія", "Програмна інженерія", "Прикладна математика та інформатика", "Автоматика, кібернетика та обчислювальна техніка" а також може бути корисним для фахівців у галузі розроблення та застосування декларативних мов програмування стосовно прикладних задач штучного інтелекту.

УДК 681.142.2;004.432.42 (075.8)

ISBN 978-966-327-375-4

СТИСЛИЙ ЗМІСТ

ЗМІСТ

ВСТУП

Підручник написано на основі курсів лекцій з логічного та функціонального програмування, які автори читали протягом тривалого періоду у Національному університеті "Львівська політехніка" на кафедрах програмного забезпечення, інформаційних систем і мереж; у Львівському державному інституті новітніх технологій та управління імені В.Чорновола на кафедрі інформаційно-комп‘ютерних технологій та систем і кафедрі загальної екології та екоінформаційних систем (м. Львів); в Українській академії друкарства на кафедрі автоматизації та інформаційних технологій (м. Львів); у Національному університеті водного господарства та природокористування на кафедрі обчислювальної техніки (м. Рівне); у Технологічно-Природничому Університеті в Інституті телекомунікацій, комп’ютерних наук та електротехніки (м. Бидгощ, Польща).

У підручнику висвітлені основи логічного та функціонального програмування, основні механізми логічних вислідів, методи та підходи до побудови ефективних логічних програм та наведені приклади їх реалізації в середовищі стандартного Прологу та Турбо Прологу, подані основні методи та підходи до побудови програм та їх реалізації в середовищі стандартного Ліспу та Авто Ліспу. Після викладення основної методики побудови логічних та функціональних програм, наводиться їх реалізація на сучасних комп’ютерах та розглядаються найбільш ефективні методи побудови елегантних логічних предикатів і функцій та прикладних програм в термінах їх реалізації.

В основу сучасних мов логічного програмування покладено мову Пролог [1-3], що з'явилася в 1970 році водночас з такими поширеними мовами як Паскаль і Сі. Орієнтація мов логічного програмування – це нетрадиційне застосування комп’ютерної техніки: розуміння природної мови, створення баз знань, інтелектуальних систем прийняття рішень [4], систем розпізнавання об’єктів, експертних систем [6], систем перекладу текстів, інформаційно-аналітичних систем та інших задач, які прийнято відносити до проблем штучного інтелекту [5]. Принципова відміна мови Пролог від традиційних мов в тому, що програма на мові логіки описує не процедуру розв’язання задачі, а створює логічну модель предметної області – деякі факти відносно властивостей даної предметної області і відношення між цими властивостями, а також правила висліду нових властивостей і відношень на основі заданих. Отже, Пролог є описовою мовою, а не процедурною. Основні поняття мови досвідченими програмістами сприймаються без особливих затруднень, а от практичне втілення цих знань в

корисні і ефективні програми викликає певні ускладнення.

В основу сучасних мов функціонального програмування покладено мову Лісп, яка запропонована Дж. Маккарті [12] в 1960 р. і орієнтована на розв'язання задач нечислового характеру. Англійська назва мови Lisp є скороченням виразу опрацювання списків і підкреслює область її застосування. У вигляді списків зручно відображати алгебраїчні вирази, множини, графи, правила виводу, граматика природних мов та інші складні об'єкти. Списки є однією з найбільш гнучких форм представлення інформації в пам'яті комп'ютера.

Після появи Ліспу [13-19] рядом авторів розроблено інші алгоритмічні мови, орієнтовані на розв'язання задач штучного інтелекту, серед яких слід відзначити Пленер, Снобол, Рефал, Пролог [5]. Тим не менше мови функціонального програмування успішно розвиваються і майже за сорокарічну історію існування з'явилося ряд діалектів функціональної мови: Common Lisp [14], MacLisp, R-Lisp [21], InterLisp, Standart Lisp, Reduce [22] та ряд інших.

Відзначимо, що відміни між ними не носять принципового характеру і зводяться, в основному, до несуттєвої різниці в формі запису програми та наборі вбудованих функцій. Тому програміст, який освоїв одну із мов функціонального програмування, без особливих зусиль в стані освоїти будь-яку іншу, володіючи основними методами та алгоритмами побудови функціональних програм.

Незважаючи на очевидні переваги декларативних мов перед процедурними, довший час ці мови розвивався і застосовувався в вузькому колі спеціалістів з проблем штучного інтелекту. Лише протягом останнього десятиліття декларативні мови програмування викликали зацікавлення широкого кола спеціалістів з інформатики. Зростання популярності цих мов зумовлено виходом теорії штучного інтелекту в сферу прикладного програмування.

На сьогоднішній день з’явилися престижні національні проекти створення ЕОМ нових поколінь в яких інтелектуальний інтерфейс, орієнтований на користувача непрофесіонала в інформатиці займає центральне місце.

Пролог був прийнятий в якості базової мови в японському проекті ЕОМ п’ятого покоління, який орієнтований на дослідження методів логічного програмування і штучного інтелекту.

Теоретичною основою Прологу є символьна логіка, яка дає механізм обчислення предикатів. Цій мові притаманний ряд властивостей, які відсутні

в традиційних мовах програмування. До них відносяться: механізм логічного доведення з пошуком і поверненням, вбудований механізм співставлення зі зразком, проста і виразна структура даних з можливістю її зміни. Пролог відрізняє однакова структура даних і програм. Оскільки дані і програми не відрізняються, то їх можна змінювати під час роботи Пролог-системи. В Пролозі, як і у функціональних мовах відсутні вказівники, оператори присвоювання та переходу. Природнім методом програмування тут є рекурсія. Ряд важливих рис теорії програмування випливає з декларативності мов, що дозволяє розуміти програму не досліджуючи динаміки її виконання. Однак істинна потужність цих мов ґрунтується не на одній якійсь властивості, а в сукупному і грамотному поєднанні всіх. Мови декларативного характеру орієнтовані не на розробку розв'язків, а на систематичний і формалізований опис задачі з тим, щоб розв'язок випливав із складеного опису. В сучасному програмуванні позначився перехід від пошуку розв'язку до визначення того, який розв'язок є потрібний. Якщо мови логічного програмування ідеально пристосовані для реалізації процедури логічного висновку, то функціональні мови дозволяють будь-яку синтаксично правильно побудовану програму на довільній мові програмування записати у вигляді функції, тим самим оптимальним чином побудувати інтерпретатор досліджуваної мови програмування.

Підручник складається з двох частин, перша з яких має три розділи, друга – п'ять розділів. У першому розділі викладаються основи логічного програмування, основні підходи та принципи побудови логічних предикатів та програм, вбудовані логічні механізми погодження та пере погодження цільових тверджень, логічного виводу, організації рекурсій та повтору, відсікання та вибору логічно вивіреного розв'язку, способи формування та роботи з базами даних і файлами.

У другому розділі першої частини підручника висвітлені основні підходи та найбільш вживані методи до побудови ефективних логічних програм та наведені приклади їх реалізації в середовищі Турбо Прологу. Після викладення основної методики побудови логічних програм, наводиться їх реалізація на сучасних комп'ютерах та розглядаються найбільш ефективні методи побудови елегантних логічних предикатів та програм в середовищі Турбо Прологу. В заключному розділі першої частини підручника відзначені найбільш перспективні напрями розвитку мов логічного програмування та доцільності їх використання. В третьому розділі підручника приведені вимоги до виконання шести лабораторних робіт в середовищі Турбо Прологу та описана методика їх проведення, оформлення звіту та захисту виконаної

роботи. Вимоги сформульовані таким чином, що роботи можуть виконуватися як під керівництвом викладача, так і самостійно всіма бажаючими освоїти основи логічного програмування та набути практичних навиків розробки прикладних програм в середовищі Турбо Прологу.

У наступних трьох розділах другої частини розглянуті теоретичні та практичні аспекти побудови ефективних функціональних програм. Функційна скерованість програми, коли важливим є результат - значення функції, дозволяє писати і відлагоджувати складні програмні комплекси. Ясність програм, чітке розділення їх функцій, відсутність каверзних сторонніх ефектів є необхідною умовою при розв'язанні логічно складних задач, якими є задачі штучного інтелекту.

Сьомий розділ вміщує вимоги до виконання лабораторних робіт в середовищі Турбо Прологу

Останній, восьмий розділ підручника присвячений викладенню суті системного підходу до освоєння та розвитку як логічних так і функціональних програм.

Весь матеріал викладається послідовно, логічно зв’язано, починаючи від простіших понять та їх програмних реалізацій до більш складних та цікавих у теоретичному та прикладному сенсі.

Авторський колектив висловлює щиру вдячність рецензентам підручника: завідувачу кафедри прикладної математики НУВГП професору, д.т.н. Мартинюку П.М.; завідувачу кафедри інформаційних систем і мереж Національного університету "Львівська політехніка" професору, д.т.н. Пасічнику В.В.; завідувачу кафедри прикладної математики Львівського національного університету ім. І. Франка. професору, д.фіз-мат.н. Савулі Я.Г.; завідувачу відділу Фізико-механічного інституту імені Г.В. Карпенка НАН України, професору Технологічно-природничого університету ім. Снідецьких (м. Бидгощ, Польща) професору, д.фіз.-мат.н. Яворському І.М. за рецензування роботи, конструктивні зауваження та фахове і змістовне обговорення в процесі підготовки та видання підручника.

Підручник призначений для студентів, спеціалістів та магістрів комп’ютерних спеціальностей, прикладної математики та споріднених технічних спеціальностей вищих навчальних закладів, а також може бути корисним фахівцям з розроблення прикладних систем штучного інтелекту, розпізнавання та ідентифікації об’єктів, інтелектуального опрацювання даних, інженерії знань.

ЧАСТИНА 1. ОСНОВИ ЛОГІЧНОГО ПРОГРАМУВАННЯ

Розділ 1. Основні поняття логічного програмування

1.1. Суть логічного програмування

1.1.1. Вступ до логічного програмування

Логічне програмування – це програмування мовою логічних висловлювань або тверджень. Серед мов логічного програмування найпоширеніша мова Пролог, яка з'явилася в 1970 р. одночасно з такими мовами як СІ, Паскаль. Її орієнтація – нетрадиційне застосування обчислювальної техніки, створення динамічних баз даних, розуміння природної мови, експертні системи, інтелектуальні системи прийняття рішень, інформаційно-аналітичні системи тобто ті напрямки, які прийнято називати задачами штучного інтелекту. Принципова особливість цієї мови, – те, що програма на Пролозі описує не процедуру розв'язання задачі, а логічну модель предметної галузі – деякі факти щодо властивостей предметної області і співвідношення між цими властивостями, а також правила виведення нових відношень і властивостей із вже заданих. Тому мову Пролог необхідно розглядати не як процедурну, а як описову.

Як більшість мов високого рівня Пролог має велику кількість реалізацій [3; 5; 11], що характеризуються певними семантичними і синтаксичними особливостями. Ми будемо розглядати базовий, або стандартний Пролог, знання якого дає змогу легко адаптуватися до конкретної його реалізації.

Для початку на інтуїтивному рівні познайомимося з основними ідеями логічного програмування.

Логічне програмування зводиться до описання об'єктів і співвідношення між ними. Воно використовується, якщо задача може бути зведена до описання властивостей об'єктів. Коли ми говоримо фразу: "Джон має книгу", то тим самим встановлюємо існування властивості володіння між об'єктами Джон і книга. Порядок слідування об'єктів має принципове значення, оскільки його зміна призводить до зміни змісту фрази. Ще одне зауваження принципового характеру слід зробити перш ніж розпочати програмування на мові логіки. Крім логічних тверджень, співвідношення між об'єктами можна задавати за допомогою правил. Наприклад, правило "Дві людини є сестрами, якщо вони обидві є жіночого роду і мають одних і тих самих батьків". Важливо відзначити, що правила є спрощеними, але вони можуть використовуватися як визначення. Зрештою, і не слід чекати, щоб

означення давало все про означуваний об'єкт. Треба вміти в означенні виділити ті істотні риси, які допоможуть розв'язати конкретну прикладну чи наукову задачу.

Розв'язуючи ту чи іншу логічну задачу, можна виділити такі етапи її програмування мовою логіки:

1) оголошення деяких фактів про об'єкти і відношення між ними;
2) визначення деяких правил про об'єкти і відношення між ними;
3) формулювання запитів про об'єкти і відношення між ними.

Програму на Пролозі можна розглядати як сховище фактів і правил, або базу даних, яка використовуються для пошуку відповідей на запитання.

Програмування на Пролозі полягає в тому, щоб задати всі ці факти і правила. Пролог-система після введення фактів і правил дає механізм, щоб робити логічні висновки, переходячи від одного факту до іншого, від одного правила до наступного, занесеного в базу даних.

При роботі в середовищі Прологу між користувачем і ЕОМ відбувається щось подібне до діалогу. Клавіатура комп'ютера використовується для введення інформації, а дисплей для виведення отриманих результатів. Після введення всіх фактів і правил, що стосуються задачі та формування всіх запитів згідно з синтаксисом мови, здійснюється пошук логічно правильної відповіді (можливо не одної) з виведенням результатів на екран дисплея. Якщо отримані результати не задовольняють користувача, формулюються нові запити або поповнюється база даних новою інформацією і задача розв'язується повторно.

1.1.2. Визначальні поняття та означення

Введемо послідовно основні поняття мови Пролог. Оголошуючи факт: "Джону подобається Мері", використовують стандартну форму запису факту

`подобається (джон, мері).`

Отже, оголошуючи факти, необхідно дотримуватися таких правил:

1) Імена всіх відношень і об'єктів починаються з малої букви.

Наприклад, **`подобається, має, джон, мері`**.

2) На першому місці записується ім'я відношення. Потім в дужках через кому записуються всі імена об'єктів.

3) Кожен факт обов'язково закінчується крапкою.

Визначаючи за допомогою фактів відношення між об'єктами, слід враховувати послідовність слідування імен об'єктів всередині круглих дужок. Ця послідовність може бути довільною, але якщо ми один раз встановили

певну послідовність, то її необхідно постійно дотримуватися. Отже, факт **подобається (джон, мері)** не є тим самим, що **подобається (мері, джон).**

Слід мати на увазі, що імена об'єктів можуть мати різну інтерпретацію природною мовою. Наприклад:

```
цінний (золото).          /*  Золото має цінність.*/
жінка(марія).             /*  Марія-жінка.*/
батько(іван,петро).       /*  Іван є батьком Петра.*/
давати(джон,мері,книга)./* Джон дає Мері книгу.*/
```

Одне і те саме ім'я може по-різному інтерпретуватися природною мовою і автор програми визначає конкретну інтерпретацію. Поки послідовно використовується одна і та сама інтерпретація імен, то ніяких проблем не виникає. Слід сказати, що імена об'єктів, які поміщені в круглі дужки, називаються аргументами, а ім'я відношення, яке записується перед круглими дужками, має назву предиката. Імена відношень і об'єктів є цілком довільними. Однак, як правило, імена об'єктів ми вибираємо так, щоб вони нагадували нам, що означають. Отже, заздалегідь слід визначити, що означають ті чи інші імена об'єктів і відношень і яка послідовність їх слідування, дотримуючись надалі визначеної послідовності.

Відношення може мати довільну кількість аргументів.

Наприклад:

```
грати(джон, мері, футбол).
грати(іван (захисник), петро
(нападник),гра(бадмінтон)).
```

Такий спосіб дозволяє записувати складні взаємозв'язки між об'єктами.

У Пролозі можна оголошувати факти, які не є істинними. Отже, факти в Пролозі дають змогу виразити довільні відношення між об'єктами.

Сукупність фактів в Пролозі (а також і правил) утворюють базу даних. Термін "база даних" будемо вживати кожен раз, коли йдеться про об'єднання ряду фактів (правил) для їх сумісного використання при розв'язанні конкретної задачі.

Після задання набору фактів та правил, формуються звертання до Прологу із запитами. Для цього використовується спеціальний символ "**?**", після якого йде тире. Розглянемо запит:

? – має (мері, книга).

Правила формування запитів цілком аналогічні правилам оголошення фактів, окрім того, що на першому місці стоїть спеціальний символ **'?'** і тире.

Після формулювання запиту ініціюється процедура перегляду бази даних, яка раніше введена в систему. Здійснюється пошук фактів, які можуть бути порівняні із запитом. Два факти є порівнянні, якщо імена відношень в цих фактах однакові, і їх аргументи попарно збігаються. Якщо Пролог знаходить факт порівняння із запитом, то відповідь буде "так", в протилежному випадку – "ні". Відповідь видається безпосередньо під запитом.

Нехай задана база даних:

```
подобається (джо, риба).
подобається (джо, мері).
подобається (мері, книга).
подобається (джо, книга).
```

Після введення цих фактів у Пролозі він би видав такі відповіді на запитання.

```
?-подобається (мері, джо).
Ні
?-подобається (джо, мері).
Так
?-король (джо, франція).
Ні
```


Відповідь "ні" в Пролозі означає, що ніщо в базі даних не може бути порівняне з цим запитом, але це зовсім не означає, що це твердження є хибним або не може бути доведеним.

Обговорювані досі факти і запити особливого інтересу не становлять. По суті, ми отримуємо ту інформацію, яку ввели в систему. Для вищенаведеної бази даних змістовнішим був би запит: ”**Які об’єкти подобаються Джо?**” Для формулювання такого запиту використовується поняття змінної. Змінну використовують для позначення деякого об’єкта, про який невідомо, чи він є в базі даних, чи ні. Змінна конкретизується, якщо існує об’єкт, який позначений цією змінною, в противному разі змінна є неконкретизована. У Пролозі існує домовленість, яка дає змогу відрізнити змінну від імені конкретного об’єкта – кожне ім’я, яке розпочинається з великої букви, називається змінною. Так, запит **“Чи що-небудь подобається Джо”**, – формулюється так:

```
?-подобається (джо, X).
```

Шукаючи відповідь, програма на Пролозі організовує перегляд бази даних так, щоб знайти об’єкт, з яким би ще неконкретизована змінна X могла

би бути порівняна. Якщо необхідно, змінним можна давати довші імена. Так, цілком прийнятним є запис

?-подобається (джо, Що-небудь_ що подобається джо).

У результаті будуть видані всі об'єкти, що подобаються Джо. Після отримання першого результату за допомогою спеціального маркера відмічається те місце в базі даних, звідки взята ця відповідь і очікуються подальші вказівки користувача. Якщо тепер натиснути клавішу RETURN, то пошук інших можливих відповідей припиниться. При натисканні клавіші **;** продовжується пошук відповідей, починаючи з місця, відміченого маркером. В цьому випадку ми говоримо, що програма намагається перепогодити ціль.

Як це буде здійснюватись? Забувається значення "**риба**", яким конкретизована наша змінна і знаходиться для її погодження факт "**подобається (джо, мері)**". Результатом виконання програми буде:

Що-небуть_що подобається джо = мері

Коли в базі даних не залишиться жодного факту для погодження отримаємо відповідь "ні". Ця відповідь означає, що в базі немає більше фактів, які б мали бути погоджені з нашим запитом.

Якщо б ми хотіли сформувати складніший запит: **"Чи подобається Джо і Мері один одному"**, то це можна зробити, сформувавши два запити, тобто два цільові твердження. Подібна комбінація цілей досить часто вживана в Пролозі, а тому для неї існує спеціальне позначення.

Нехай задана база:

```
подобається (марія, іван).
подобається (марія, вино).
подобається (іван, вино).
подобається (іван, марія).
```

Для того щоб сформувати кон'юнкцію (об'єднання) цілей ми використовуємо кому:

?-подобається (іван, марія), подобається (марія, іван).

Оскільки обидві цілі погоджується з базою даних, відповідь буде "так".

Поєднуючи можливості кон'юнкції цілей та поняття змінної, можна будувати доволі складні запити. Кон'юнкцію цілей можна подати в запиті як список цілей. При цьому відбувається погодження з базою кожної цілі, починаючи з крайньої лівої цілі в порядку їх запису. Якщо виявляється факт, порівняний з ціллю, то програма на Пролозі залишає в цьому місці маркер.

Крім того, деякі змінні, які раніше були конкретизовані, тепер розконкретизуються. Якщо конкретизується певна змінна, то конкретизуються всі входження цієї змінної в правило. Далі програма намагається задовольнити свого сусіда праворуч, починаючи пошук фактів з вершини бази. Якщо цільове рішення не виконується, відбувається повернення до сусіда ліворуч і намагання знайти нове порівняння, починаючи з місця, відзначеного маркером. Одночасно розконкретизовуються всі змінні, які конкретизувалися при досягненні даної цілі. Якщо не вдається знайти нове погодження для крайньої лівої цілі, яка вже не має сусіда ліворуч, то вся кон'юнкція цілей вважається непогодженою. Така поведінка Прологу, коли він намагається неодноразово погодити цілі, що входять в кон'юнкцію цілей, називається пошуком з поверненням (back tracing).

Якщо деякий факт залежить від сукупності інших фактів, то для цього доцільно користуватися визначенням правила. Під правилом будемо розуміти деяке загальне твердження про об'єкти або відношення між ними.

Допустимо, ми хочемо сказати, що Джону подобаються всі люди. Один із способів реалізації цього – це запис окремого факту для кожного чоловіка, що подобається Джону. Такий спосіб є доволі складним, тим більше, якщо в базі зустрінеться більше сотні імен. Другий спосіб виразити факт, що Джону подобаються всі люди – це сказати, що Джону подобається будь-який об'єкт за умови, що цей об'єкт є людиною.

Таке представлення є компактнішим, а значить, і доцільнішим. В природній мові для визначення правила ми можемо використовувати зв'язку "якщо". Наприклад:

- **Я користуюся парасолькою, якщо іде дощ.**
- **Джон купує вино, якщо воно дешевше від пива.**

Правила використовуються і для того, щоб дати означення:

```
X є птахом, якщо:
                    X – жива істота і
                    X – має пір'я.
```

Або

```
X і У є  сестрами, якщо :
                    X і У  – жінки та
                    X і У мають спільних родичів.
```

Треба мати на увазі, що кожне входження змінної в правило означає один і той самий об'єкт.

Розглянемо кілька прикладів використання правил, розпочинаючи з правила, що містить одну змінну і кон'юнкцію:

Джону подобається будь-хто, кому подобається вино

або з використанням змінної

Джону подобається Х, якщо Х подобається вино.

У Пролозі правило складається із заголовка правила і тіла правила, які з'єднуються символом **:- (якщо).** Тому наше правило на Пролозі набуває вигляд:

подобається (джон, Х) : – подобається (Х, вино).

Правило закінчується крапкою. Якщо змінна, яка входить в правило, конкретизована, то всі входження змінної **Х** в цьому правилі конкретизується цим значенням.

Продемонструємо дію правила, яке використовує більше ніж одну змінну, взявши дані з родини англійської королеви Вікторії.

Використаємо предикат **родичі** з трьома аргументами: **родичі (Х, Н, F),** яке означає, що **Х** має таких родичів: **М** – матір, **F** – батько.

Крім цього, використаємо предикати **чоловік (Z), жінка (М)** в звичайному їх розумінні.

Частина бази даних могла би мати вигляд:

```
чоловік (альберт).
чоловік (едуард).
жінка (аліса).
жінки (вікторія).
родичі (едуард, вікторія, альберт).
родичі (аліса, вікторія, альберт).
```

Введемо визначення правила **є_сестрою (х, у)** так:

```
Х є сестрою У, якщо:
1) Х є жінкою
2) Х має родичів М і F
3)У має тих же родичів, що Х.
```

Це може бути задане правилом:

```
є_сестрою (Х,У): – жінка (Х), родичі (Х, М, F),
родичі (Y, M, F), X < > Y.
```

В тілі правила використані змінні, яких немає в заголовку правила, хоч вони будуть опрацьовуватися які всі інші змінні.

Задамо запит:

```
?-є_сестрою (аліса, едуард).
```

Для його виконання будуть виконані дії:

1. Запит буде порівняно із заголовком правила, яке єдине в базі. Змінна X конкретизується значенням **аліса, Y – едуард**. Це місце буде помічене маркером і відбувається погодження трьох підцілей.

2. Першою підціллю буде **жінка (аліса),** що підтверджується фактом з бази даних. Місце погодження відмічається маркером.

3. Відбудеться погодження для предиката **родичі (аліса, M, F).** Після знаходження відповідного факту місце відзначається маркером і йде погодження третьої цілі.

4. Далі пошук підтвердження факту **родичі (едуард, вікторія, альберт).** Оскільки ця остання ціль погоджується, то буде видано відповідь **так**.

Як приклад розглянемо правило: **людина може щось вкрасти, якщо людина є злодієм, ця річ їй подобається і річ має певну цінність**. Складемо базу даних, використовуючи предикати **злодій, подобається, цінний**. Ми можемо пронумерувати факти, поміщаючи номери між спеціальними символами / *…*/, які в Пролозі використовуються для запису коментарів. Маємо базу:

```
/*1*/  злодій (джон).
/*2*/  подобається (мері, їжа).
/*3*/  подобається (мері, вино).
/*4*/   подобається (джон, X): – подобається (X, вино).
/*5*/  може_вкрасти (X, Z): – злодій (X), подобається
          (X, Z), цінний(Z).
/*6*/  цінний (вино).
/*7*/  цінний (вино).
```

У цьому прикладі визначення предикату **подобається** містить два факти і правило. При погодженні запиту з фактом пошук закінчується або у разі непогодження відбувається перехід до наступного твердження.

Правило викликає подальший пошук в базі даних для погодження власних підцілей. Для прикладу, при звертанні до Прологу з запитом: **"Що може вкрасти Джон?"**, який транслюється так:

```
?-може_вкрасти (джон, X).
```

здійснюється пошук в базі твердження, описаного предикатом **може вкрасти** за номером 5. Оскільки це правило, то необхідно погодити з базою даних тіло правила. При погодженні запиту з заголовком правила 5 змінна **X** в правилі конкретизується іменем **джон**, а змінна **Z** в правилі стає зчепленою зі змінною **X** в запиті. Як тільки в будь якому місці правила відбудеться конкретизація змінної, то вона набуває це значення в усіх місцях правила, де тільки зустрінеться. Далі йде погодження першої цілі в тілі правила фактом 1 з бази даних **злодій (джон).** Змінна **X** конкретизується всюди в запиті іменем **джон**. Далі Пролог намагається погодити другу підціль правила **подобається (джон, Z),** порівнюючи її з заголовком правила 4 **подобається (джон, X).** Для погодження правила 4 буде знайдений факт 3, і в результаті порівняння змінна **X** в правилі 4 набуває значення **мері**. Оскільки ціль в правилі 4 досягнута, то погоджене все правило 4. Отже, твердження 5 погоджується з базою даних при **X** зі значенням **мері**, а оскільки **X** є зчепленою зі змінною **X** вхідного запиту, то й ця змінна приймає значення **мері**. Отже, **джон** може вкрасти **мері,** оскільки третя. підціль в запиті буде погоджена фактом 7.Відзначимо, що факт 2 ніякого відношення до відповіді на цей конкретний запит не має і не був використаний при погодженні запиту.

З цим невеликим набором елементарних конструкцій мови Пролог можна писати корисні програми, які мають прикладне значення.

1.1.3. Контрольні питання та вправи

1. Суть логічного програмування та його особливості.
2. Правила формулювання фактів та запитів.
3. Основні етапи побудови Пролог-програми.
4. Поняття про цільове твердження та кон’юнкцію цілей.
5. Що таке змінна в логічному програмуванні? Навести приклади використання.
6. Дати визначення правила в логічних мовах.
7. Записати предикат, який визначає братні стосунки між двома особами.
8. Які спільні та відмінні риси між функціональним та логічним програмуванням?
9. Написати на стандартному Пролозі програму для визначення родинних стосунків власної сім'ї, попередньо створивши відповідну базу

даних.

10. На конкретному прикладі описати схему пошуку підтвердження цільового запиту з базою даних.

11. Маючи в базі даних такі твердження, які описують відношення:

```
Батько(X,У) /*X є батьком У*/
Мати  (X,У) /*X є матір'ю У*/
чоловік (X) /*X є чоловіком*/
Жінка (X) /*X є жінкою */
Родич (X,У) /*X є родичем (батьком або матір'ю)Z*/
Різні (X,Z) /*конкретні значення X і Z не
збігаються */,
```

написати правила до цих відношень:

```
Бути_матір'ю (X)      /*X є матір'ю */
Бути_батьком  (Z)    /*Z є батьком */
Бути_сином (X)  /*X є сином */
Бути_сестрою (X,Z  /*X є сестрою Z*/
Дід (X,Z)  /*X є дідом для Z*/
Тітка (X,Z)  /*X є тіткою для Z*/
Спільні родичі (X,Z)  /*X і Z мають спільних
родичів */.
```

1.2. Синтаксичні особливості мови Пролог

1.2.1. Синтаксис мови Пролог

Наведемо детальне описання тих елементів Прологу, які ми розглянули. Пролог забезпечує засоби для структурування даних в тій послідовності, в якій здійснюються спроби погодити цільові твердження з базою даних. Очевидно, для структурування даних необхідне знання синтаксичних правил, які використовуються для позначення даних та знання принципів роботи механізму повернення.

Синтаксичні правила будь-якої формальної чи навіть природної мови описують допустимі способи поєднання слів. Так, відповідно до норм англійської мови речення “I see a zebra” (Я бачу зебру) є синтаксично правильним, на відміну від речення “Zebra see I a” (Зебра бачу я). Аналогічно можна сказати, що всі речення мови Пролог будуються за строгими синтаксичними правилами. Ці речення складаються з термів. Терм – це константа, змінна або структура. Кожне з цих понять ми розглядали раніше

на конкретних прикладах, не даючи строгих визначень. Терм у мовах логічного програмування записується у вигляді послідовності літер. Літери діляться на чотири категорії :

1. Великі букви англійського алфавіту;
2. Малі букви англійського алфавіту;
3. Цифри від 0 до 9;
4. Спеціальні знаки : +, -, *, /, \, ^, >,<,~, :, . ,? , @ ,# ,$, &.

Насправді спеціальних знаків є більше, але вони використовуються у виключних випадках і не поширені у всіх версіях мови Пролог. Терми кожного типу мають свої правила творення імен термів на основі літер.

Константи – це пойменовані конкретні об'єкти або відношення. Розрізняють два типи констант: атоми або цілі числа. Наприклад: **`подобається, марія, книга, має, може_вкрасти: подобається, a1, 125, "Львів".`**

Атоми бувають двох типів: складені з букв і цифр, і складені із спеціальних знаків. Перші починаються з малої букви, а другі, як правило, складаються із спеціальних знаків. Якщо треба атом записати з великої букви, він береться в одиничні лапки (апострофи). Атом, взятий в одиничні лапки, може містити довільну послідовність літер. Змішаний атом завжди починається з літери. Для зручності читання між символьними атомами можна поміщати спеціальну літеру підкреслення "_". Спеціальні символи "**?-** ", що використовуються для формування запитів та "**:-**" – для розділення заголовка і тіла правила теж належать до атомів.

Такі об'єкти не є атомами: **`1m, жовто-синій, exo, Новий_рік.`**

Другий тип констант – цілі числа, використовуються для подання числової інформації з метою виконання арифметичних дій. Цілі числа складаються тільки з цифр і не можуть містити десяткової крапки. Як правило, в логічних задачах немає потреби оперувати великими дробовими або від'ємними числами. Тим не менше, Пролог дає можливість визначити предикати для операцій над раціональними числами з довільною точністю. Відзначимо, що у будь-якій версії Прологу можна оперувати числами від 0 до 16383.

Інший різновид термів Прологу – змінні. Щоб відрізнити її від константи, змінну починають з великої букви або символу підкреслення "_". Змінні можуть бути як завгодно довгими. Іноді виникає потреба послатися на змінну, яка ніколи не буде використана, так звану анонімну (непоіменовану) змінну. Для її позначення використовується символ "_", який ставиться на

першому місці. Якщо в одному правилі або в одній цілі зустрілося декілька змінних з одним іменем, то вони конкретизуються завжди одним об’єктом.

Інтерпретація анонімної змінної в логічному програмуванні не є однозначною, тим не менше, є корисною в прикладних застосуваннях, оскільки зникає потреба давати імена змінним, які зустрічаються в програмі лише один раз і більше ніде не будуть використані.

Найважливішим термом Прологу є структура, яку ми будемо розглядати як єдиний об'єкт, що складається з сукупності інших об'єктів – компонентів структури. Прикладом може служити абонементна картка читача бібліотеки, що містить кілька компонентів: дані про читача, дані про книгу, час видачі книги, дату повернення книги, місце зберігання книги. Дані з компонент, в свою чергу, можуть містити свої власні компоненти. Наприклад, дані про книгу містять назву книги, прізвища авторів, інвентарний номер книги.

Структури служать засобом організації даних в програмі. Ім'я структури називається функтором, після якого в круглих дужках записуються його компоненти, які, своєю чергою, можуть утворювати нові структури. Відзначимо, що таке вкладання даних може відбуватися на довільну глибину і дає змогу створювати ієрархічні бази даних. Прикладом структури може служити такий запис:

має (джон, книга (грозовий_перевал, автор (емілі, бронте))).

Ця структура інтерпретується фактом про наявність у Джона книги "Грозовий перевал", автором якої є англійська письменниця Емелі Бронте.

Як бачимо, синтаксис структури збігає з синтаксисом фактів. Предикат, який використовується в імені факту чи правила, є функтором деякої структури, компоненти якої збігаються з аргументами (об'єктами) факту чи правила.

Відзначимо, що всі частини програм на Пролозі складаються з констант, змінних і структур. Імена констант і змінних утворюються за допомогою літер, які розпізнаються Прологом.

Розрізняють два типи літер: друковані, які з'являються на терміналі при введенні чи виведенні і недруковані, які розпізнаються після виконання певних дій: поява пробілу, переведення курсору на новий рядок тощо. В Пролозі є такі літери, які можна використовувати для написання програми:

A B C D E F G H I J K L M N O P R S T U V W X Y Z
a b c d e f g h i j k l m n o p r s t u v w x y z
0 1 2 3 4 5 6 7 8 9

! " # $ % ' ' = - ~ ^ | \ / { } [ ] ( ) _ @ + ; * : < > , . ? &

Літери в Пролозі інтерпретуються як певні цілі числа в діапазоні від 0 до 127 згідно з таблицею кодування ASCII (American Standart Code for Information Interchange), яка широко використовується в мовах програмування у всьому світі.

1.2.2. Операторна форма представлення даних і програм

Формуючи структури, деякі функтори зручно записувати в операторній формі. Зокрема, ця форма є зручною для запису арифметичних операцій. Наприклад, вираз a+b*z у вигляді структури запишеться так + (a, * (b,z)). Оператори не виконують ніяких дій до появи знаку рівності. Так, 7 і 3+4 – це різні об'єкти Прологу.

Працюючи з операторами, необхідно знати їх основні властивості: позицію, пріоритет та асоціативність. Розглянемо добре відомі оператори арифметичних дій: +, -, *, /. За позицією це є інфіксні оператори, оскільки вони розміщені між своїми аргументами. Якщо оператор помістити перед аргументом (-x), то він є префіксним. Якщо він розташований після аргументу, його називають постінфіксним: h! (!- оператор розрахунку факторіалу).

Пріоритет оператора вказує порядок його виконання. В Пролозі кожен оператор має свій клас пріоритету, що являє собою ціле число, значення якого залежить від конкретної версії Прологу. Чим менше число, тим вищий пріоритет оператора.

Знання властивості асоціативності необхідне за наявності операторів з одним класом пріоритету. Розрізняють лівоасоціативні та правоасоціативні оператори. Лівоасоціативні оператори – це такі, які ліворуч від себе мають оператори меншого або рівного класу, а праворуч – тільки нижчого. Арифметичні оператори є лівоасоціативними. Тому вираз 8/2/2 еквівалентний (8/2)/2, а не 8/(2/2). Якщо є вираз, розуміння якого ускладнене правилами пріоритету або асоціативності, то доцільно застосовувати круглі дужки.

1.2.3. Арифметика мови Пролог

Необхідно відзначити, що структури, які утворюються арифметичними операторами аналогічні будь-яким іншим структурам і не викликають ніяких дій доти, доки не зустрінеться предикат "=", який є інфіксним оператором і встановлює рівність своїх аргументів. Предикат рівності є вбудованим і

намагається довести тотожність своїх аргументів. Запит

```
? - x = y
```

працює так, ніби то в базі даних міститься факт

```
x=x.
```

Оскільки як x і y можуть виступати будь-які терми, дозволені синтаксисом Прологу, то при конкретизації змінних у твердженнях діють правила:

1) Якщо **x** – неконкретизована змінна, а **y** – конкретизована, то незалежно від її значення **x** і **y** є тотожні.

2) Цілі числа і атоми завжди рівні між собою.

3) Дві структури рівні, якщо вони мають однаковий функтор, однакову кількість аргументів, причому всі аргументи рівні між собою. Наприклад в запиті

```
?- їхати (іван, велосипед) = їхати (іван, X)
```

змінній **X** буде присвоєно значення **велосипед**. Оскільки структури вкладаються на довільну глибину, то при перевірці їх рівності необхідно більше часу для аналізу всіх структур та їх компонент.

При порівнянні двох неконкретизованих змінних **X** і **Y** вони стають зчепленими і як тільки конкретизується одна з них, то друга набуває те саме значення. Отже, в правилі

```
нічого_не_робити (x, y) :-x = y
```

другий аргумент буде конкретизований тим самим значенням, що і перший. Останнє правило можна записати простіше у вигляді факту про те, що змінна дорівнює сама собі :

```
нічого _не_робити (x, x).
```

В Пролозі передбачений ще один вбудований предикат **"\ =",** тобто твердження не тотожні.

Цільове твердження **x \ = y** істинне, якщо не може бути доведена справедливість твердження **x=y.**

Для порівняння чисел у Пролозі передбачено такі предикати:

```
x=y   /* x і y є одне і те саме число */
x\=y  /*x і y є різні числа */
x<y  /* x менше від y */
x>y   /* x більше від y */
x=<y /* x дорівнює або менше від y */
x>= y /* x дорівнює або більше від y */
```

Як приклад використання чисел та арифметичних предикатів допустимо, що ми маємо базу даних про принців Уельсу, які правили країною в IX–X-му ст. Введемо предикат **правив (A, X, У)**, якщо принц з іменем **A** правив країною з року **X** до року **У**. Допустимо, що ми хочемо визначити, хто був принцом Уельсу в конкретний рік **Z**. Можна визначити правило **принц (X, Z)**, яке істинне, якщо принц на ім'я **X** був на троні в рік **Z**, якщо

1) **X** правив з року **A** по рік **B**;

2) рік **Z** знаходиться між роком **A** і **B** або збігається з одним із них.

Відповідна база даних на Пролозі має такий вигляд:

```
правив (родрі,  844, 878),
правив (анаравд, 878, 916),
правив (хівел, 916,  950),
правив (лаго, 950, 985),
правив (кадвалон, 985,  886),
правив (маредуду, 986, 999).
принц (X,У):- правив(X, A, B), У>=A, У=<B.
```

1.2.4. Структура Пролог-програми

Програма на Пролозі складається з п'яти розділів: опис доменів, розділ бази даних, опис предикатів, розділ опису цілі та опису тверджень або фактів. Ключові слова, які використовуються для цього – **domains, database, predicates, goal, clauses.**

1.**domains** – описує різні класи об'єктів, які використовуються в програмі;

2.**database** – описує твердження бази даних, які є предикатами динамічної бази даних;

3.**predicates** – опис використовуваних програмою предикатів;

4.**goal** – формується призначення програми мовою Пролог;

5.**clauses** – заносяться факти і правила, відомі апріорі.

```
/* коментар */
domains
/* опис доменів */
database
/* опис предикатів динамічної бази даних */
predicates
```

```
/* опис предикатів */
goal
/* цільові твердження */
clauses
/* запис фактів, правил */
```

Пролог дозволяє використовувати 6 типів доменів (див. табл. 1.1):

Таблиця 1.1

Типи даних Турбо Прологу

ТИП ДАНИХ	КЛЮЧОВЕ СЛОВО	ДІАПАЗОН ЗНАЧЕНЬ	ПРИКЛАДИ
Символи	Char	Всі дозволені символи	“a”,”в”,”B”,”/13”,”%”
Цілі числа	Integer	Від –32768 до 32767	-63, 84, 2349
Дійсні числа	Real	Від +1E-307 до +1E308	-42769, 360, 1.25E23,5.15E-9
Рядки	String	Послідовність символів (не більше ніж 250)	“today”,”123”,
Символічні імена	Symbol	1. Послідовність букв, цифр та підкреслень; перший символ – велика буква 2. Послідовність довільних символів, що є в лапках	“Stars and Stripes”? “singing in the rain”
Файли	Files	Допустиме в DOC ім’я файлу	mail.txt, BIRDS.DBA

Не кожна з програм Турбо Прологу містить всередині себе опис своєї цілі, часто ціль є зовнішньою. Такі програми називаються ітеративними. Це дає повну свободу користувачу, оскільки запити можна формулювати залежно від тих відповідей, які дав комп’ютер.

1.2.5. Контрольні питання та вправи

1. Дати визначення поняття терма.

2. До якого типу термів належать спеціальні символи , ? _ та : -“, які застосовуються для формування запитів та правил?

3. Обґрунтувати доцільність використання анонімної змінної.

4. Основні властивості операторів.

5. Означити поняття лівоасоціативного та правоасоціативного

оператора.

6. Чи правильні такі цільові твердження і якими значеннями будуть конкретизовані змінні:

```
космонавт (А, львів)=космонавт (львів, париж).
крапка (X, Y, Z)=крапка (X1,Y1,Z1)= крапка (1,2, 3).
іменник (діяч)=діяч.
"подвиг"=подвиг.
f(X, Y)=f(a, в).
f(X, g(a, d))=f(Z, g(Z, C)).
```

7. В базі даних задані предикати **нас (X, Y)**, – що визначає число жителів **Y** в країні **X** і **площа (X, S)**, що визначає площу **S** країни **X**. Записати правило **густина(X, P)**, яке визначає густину **P** країни **X**.

8. Навести вбудовані предикати базового Прологу для виконання арифметичних дій.

9. Дати визначення логічної бази даних.

1.3. Особливості роботи та вбудовані предикати Турбо Прологу

1.3.1. Особливості роботи в Турбо Пролозі

Запуск Прологу здійснюється командою PROLOG: для цього зчитуються всі системні файли Турбо-Прологу і завантажуються в оперативну пам'ять.

Заставка Турбо-Прологу, яка появляється після активації системи, складається з двох вікон: в першому вікні є інформація про рік випуску і номер версії системи, нижнє вікно дає дані про задану конфігурацію Прологу. У першому рядку подано ім’я програмного файла, яке задається за замовчуванням. Інші рядки висвічують імена робочих директорій системи: директорії файлів робочих програм (**PRO**), директорії об’єктних модулів (**OBJ directory**), директорії виконавчих файлів (**EXE directory**) і системної директорії Турбо-Прологу (**TURBO directory**).

Після натискання на будь-яку клавішу висвічується головне меню системи, яке містить 7 опцій, доступних користувачу і розміщених у верхній частині екрана. Ці опції, по суті, є функціями Турбо-Прологу:

1. Запуск програми на рахунок (виконання) (**RUN**).
2. Трансляція програми (**COMPILE**).

3. Редагування тексту програми (**EDIT**).

4. Задання опцій комп'ютера (**OPTIONS**).

5. Робота з файлами (**FILES**).

6. Настроювання системи відповідно до індивідуальних потреб користувача (**SETUP**).

7. Вихід із системи (**QUIT**).

Перехід від однієї команди до іншої здійснюється за допомогою стрілок або комбінацією клавіша **ALT** і заданням першої букви команди (великої або малої).

Головне меню містить чотири вікна: **EDITOR** – в верхньому правому куті екрана; вікно діалогу (**DIALOG**) – в верхньому лівому куті екрана; вікно повідомлень (**MESSAGE**) – в нижньому лівому куті; вікно трасування в нижньому правому куті (**TRASS**).

Турбо-Пролог має потужний вбудований екранний редактор, що значно полегшує роботу програміста, його команди, в основному, збігаються з командами редактора Word star, і близький до Турбо-Паскалю.

Для завантажування у вікно редактора вже існуючого файла вибирається опція Files і в ній підкоманда Load, і якщо далі на запит імені файла просто натиснути Enter, то висвітиться перелік файлів директорії **PRO**.

Для отримання інформації про будь-яку з команд Турбо-Прологу необхідно скористатися клавішею F1. З'являється меню підказок HELP в нижній частині екрана. Якщо з нього вибрати першу опцію, то на екрані з'являється вікно HELP, яке містить короткий перелік команд редактора та ще деяку корисну інформацію. Комбінацією клавіш Sift-F10 можна розширити це вікно до розмірів повного екрану, повторне натискання Sift-F10 поверне екран в попереднє положення. Для створення нового файла слід очистити вікно від старого файла, ввійшовши в підкоманду Zap file in editor команди file. Система запитає підтвердження і якщо натиснути YES, то екран буде очищений. Після цього з'являється маленьке вікно для введення нового файла. Якщо ім'я не задавати, то новий файл буде називатися старим іменем.

1.3.2. Основні команди редактора

Для видалення слова використовується комбінація клавіш **Ctrl-T**, символу – **Ctrl-G**. Знищення стрічки – **Ctrl-Y** або **Ctrl-Backspase**. Редактор текстів працює або в режимі вставки insert, або в режимі заміни owerwrite. Для переключення режимів слід натиснути клавішу **insert** або **Ctrl-V**.

Редактор працює в режимі автоматичного вирівнювання рядків, що задається за замовчуванням і позначене на екрані написом **indent**. Для відміни цього режиму застосовується клавіша **Ctrl-Q1**. Для зміни розмірів вікна редактора до повного екрана застосовується клавіша **Ctrl- F10**.

Команди редактора дають змогу виділити фрагменти тексту з тим, щоб перемістити їх в інше місце програми, копіювати, знищити.

Для виділення фрагмента необхідно підвести курсор до початку і натиснути клавішу **Ctrl-KB**, далі підвести курсор до кінця фрагмента програми і натиснути клавішу **Ctrl-KK**. Для копіювання фрагмента слід натиснути клавішу **Ctrl-KC**, для зняття виділення **Ctrl-KH**. Для знищення фрагменту слід натиснути клавішу **Ctrl- KY**. Для цих самих потреб можна використати і функційні клавіші: **F5** – копіювання, **F6** –переміщення, **F7** – знищення. Пошук фрагмента тексту здійснюється за допомогою функціональної клавіші **F3**. Всередині екрана з'явиться вікно, куди вводиться текст пошуку. При повторному натисканні на **F3** курсор вкаже на перше входження тексту. Для повторного пошуку фрагмента необхідно натиснути клавішу **Shift-F3**.

Команди пошуку-заміни дають змогу здійснити пошук і заміну тексту на інший. Для цього натискається клавіша **F4** і набирається текст, який слід замінити. Після чергового натискання клавіші **F4** підказка, яка появляється у вікні, пропонує ввести текст для заміни. Далі слід натиснути **Enter** і вказати режим заміни: глобальний – клавіша **G** або локальний – **L**. Тепер редактор попросить дозволу на здійснення заміни. Якщо натиснути клавішу **Y**, то після кожного находження тексту для заміни він буде запитувати дозволу на заміну. А тому для реалізації глобальної заміни слід натиснути **N**. Основні комбінації клавіш наведені в табл. 1.2.

Таблиця 1.2

Основні комбінації клавіш Турбо-Прологу

Пошук	**Ctrl-QF**	**F3**
Пошук в заміна	**Ctrl-QA**	**F4**
Повторний пошук	**Ctrl-L**	**Shift-F3**
Повтор останнього пошуку і заміни	**Ctrl-L**	**Shift-F4**

1.3.3. Функційні можливості додаткового редактора

Додатковий редактор підключається за допомогою функційної клавіші **F8**, внаслідок чого з'являється запит імені файла, який додатково слід відредагувати. Причому тут також є можливість використання всього арсеналу команд редагування. Виходять з нього за допомогою клавіші **F10**. Відредаговану версію файла можна записати на зовнішній носій і повернутися до головної програми. Додаткові можливості цього редактора такі:

Копіювання тексту з іншого файла. На відміну від інших редакторів, в TP є можливість зчитувати лише частину файла, що необхідно скопіювати. Для цього слід підвести курсор до того місця куди буде здійснюватися копіювання, і натиснути клавішу **F9**. Після введення імені файлу в додатковому вікні з'явиться зміст файла. Необхідно поставити курсор на початок фрагмента файла, натиснути **F9**, далі перемістити курсор на кінець фрагмента файла і знову натиснути **F9**. Додаткове вікно зникне, а відмічений фрагмент скопіюється в робочий файл.

Команда OPTIONS використовується для встановлення опцій компілятора **Memory**, **OBJfile**, **EXEfile**.

Команда FILES має ряд підкоманд **Load** (завантаження), **Save** (запис на диск), **Directory** (задання робочої директорії), **Print** (друк), Rename (перейменування), File **Name** (ім'я файлу), **Module list** (список модулів), **Zap file in editor** (очистити вікно редактора), **Erase** (видалення файла з директорії), **Operating system** (вихід в ОС).

Підкоманда Module list – це перелік модулів програми, які мають окремо компілюватися, а потім збиратися в одну програму за допомогою редактора зв'язків.

Operating system – дає змогу тимчасово вийти в ОС. Для повернення назад необхідно ввести команду **EXIT**.

Команда SETUP – дає змогу здійснювати власний варіант налаштування системи Турбо-Пролог і містить такі опції:

window size – дає змогу змінити розміри вікна, довжину і ширину, використовуючи рядок всередині екрана;

save configurationи – дає змогу зберегти поточне розташування вікон для подальшого їх використання; за замовчуванням після задання імені конфігураційного файла він записується в директорію вихідних текстів;

load configuration використовується для задання нової конфігурації. Необхідно ввійти в опцію, ввести ім'я файл f і натиснути Enter. За замовчуванням файл буде зчитуватися з директорії з розширенням .PRO. Для повернення в головне меню необхідно натиснути "_";

directories – для задання імен директорій, з якими Пролог буде спілкуватися за замовчуванням. Такими є :

.**PRO** – директорія вхідних текстів;

.**OBJ** – директорія об'єктних файлів;

.**EXE** – директорія файлів, які виконуються

.**TURBO** – системні директорії Прологу;

.**DOS** – директорії класу DOS;

Трасування програми. Для реалізації покрокової відладки програми необхідно помістити директиви **trace** на початок програми, перед розділом **predicates**.

Директива дозволяє призупинити виконання програми. Для продовження слід натиснути клавішу **F10**. Перервати трасування можна, використавши клавішу **ESC**.

Комбінація **Ctrl-T** дозволяє виключити трасування.

Директива **shorttrace** виконує ту саму функцію, лише коротша інформація видається у вікні **trace**.

Не кожна з програм Турбо Прологу містить всередині себе опис своєї цілі, часто ціль є зовнішньою. Такі програми називаються інтерактивними. Це дає повну свободу користувачу, оскільки запити можна формулювати в залежності від тих відповідей, які дав комп'ютер.

1.3.4. Вбудовані предикати мови Пролог

asserta (<факт>) (dbasedom) : (вх)	– заносить факт у початок бази даних;
assertz (<факт>)(dbasedom) :(вх)	– заносить факт у кінець бази даних;
back (Step) (integer) : (вх)	– зсуває екранне перо на кілька позицій назад;
beep (вх)	– викликає звуковий сигнал;
bound (Var)(<довільна змінна>) : (вх)	– предикат успішний, якщо Var – означена змінна;

char_int
(CharParam,IntParam):
(вх,вих),(вих,вх),(вх,вх)
:

(вх,вих) – присвоює змінній IntParam код ASCII значення IntParam;

(вих,вх) – присвоює CharParam символ, код якого дорівнює значенню IntParam;

(вх,вх) – предикат успішний, якщо код символу CharParam збігається зі значенням IntParam;

Clearwindow – очищує активне вікно;

closefile (NF) (file): – закриває файл з іменем NF;
(вх)

concat(St1,St2,St3) – об'єднання двох рядочків в St1 і
(string,string,string): St2 в один рядок символів St3
(вх,вх,вих):
(вх,вх,вих): st3=st1+st2

cursor(Row,Column)
(integer,integer):
(вх,вх),(вих,вих):

(вх,вх): – курсор поміщається в позицію з координатами (Row,Column)

(вих,вих): – змінним Row і Column присвоюється значення поточних координат курсора

date(Year,Month,Day)
(integer,integer,integ
er):
(вх,вх,вх),(вих,вих,вих):

(вх,вх,вх): – встановлює нове значення поточної дати за допомогою змінних Year(рік), Month(місяць), Day(день);

(вих,вих,вих): – зчитує поточну дату, яка визначається за показом

	вбудованого годинника комп'ютера;
deletefile(FN) (string): (вх)	– знищує із поточної директорії файл з заданим ім'ям;
dot(Row,Column,Color) (integer,integer,integer) : (вх,вх,вх),(вх,вх,вих): (вх,вх,вх)	– зображає на екрані крапку кольору Color в позиції з координатами (Row,Column), якщо монітор знаходиться в графічному режимі;
(вх,вх,вих):	– змінна Color отримує номер кольору пікселя, координати якого задаються змінними Row і Column;
eof (FN) (file) : (вх)	– предикат успішний, якщо вказівник файла стоїть в кінці файла з іменем FN;
existfile(F_N) (string) : (вх)	– успішний, якщо в поточній директорії є файл з іменем FN;
exit	– виконання програми зупиняється;
Fail	– викликає механізм повернення через неуспіх даного предиката;
forward (Step) (integer) : (вх)	– в графічному режимі forward зсуває екранне перо на Step позицій уперед;
free (Var) (<змінна >): (вих)	– успішний, якщо змінна Var неозначена
frontchar(St, FrontChar, RestString)(string,char,string): (вх,вих,вих), (вх,вх,вих), (вх,вих,вх), (вх,вх,вх), (вих,вх,вх)	– (вх,вих,вих): присвоює перший символ рядка St змінній FrontChar, а залишок рядка – змінній RestString
frontstr(NumerOfChars,String1,StartStr,String2) (integer, string, string,	– присвоює перші NumerOfChars рядка символів String1 змінній StartStr, а залишок – змінній

string): (вх,вх,вих,вих)	String2
fronttoken(String,Token,RestString) (string, string, string): (вх,вих,вих), (вх,вх,вих), (вх,вих,вх), (вх,вх,вх), (вих,вх,вх)	– (вих, вх, вх): змінній String присвоюється результат конкатенації Token і RestString
gotowindow (WindowNo)(integer): (вх)	– здійснює швидкий перехід з біжучого вікна в вікно з No;
graphics(ModelParam,Palette,Background)(integer,integer,integer): (вх,вх,вх)	активізує графічні засоби Турбо-Прологу;
keypressed(stringParam)(string): (вх)	– успішний, якщо буде натиснута довільна клавіша;
line (Row1,Col1,Row2,Col2,Color)(integer, integer, integer, integer): (вх,вх,вх,вх)	– в графічному режимі рисує лінію між точками з координатами (Row1,Col1) та (Row2,Col2), номер кольору задається аргументом Color;
Pendown	– після виконання цього предикату стає можливим рисування ліній за допомогою предикатів forward, back;
penpos(Row,Column,Direction)(integer, integer, integer): (вх,вх,вх), (вих,вих,вих) (вх,вх,вх): (вих,вих,вих):	– розміщує графічне перо в позицію з координатами (Row,Column) та орієнтує його в напрямку, що задається аргументом Direction; – присвоює змінним координати поточної позиції пера та номер його напрямку;
Penup	– припиняє рисування;
openmodify(SFN,DFN) (file,string): (вх,вх)	– відкриває для читання та запису файл DFN та зв'язує його з логічним файлом SFN;
openread (SFN,DFN) (file,string): (вх,вх)	– відкриває для читання файл DFN та зв'язує його з логічним файлом

SFN;

openwrite (file,string): (вх,вх) – відкриває для запису файл DFN та зв'язує його з логічним файлом SFN;

readchar (CharVar)(char): (вих) – зчитує символ з поточного пристрою читання readdevice

readint(IntVar)(integer): (вих) – зчитує символ з поточного пристрою читання readdevice ціле число ;

readln(StringVar)(string): (вих) – зчитує символ з поточного пристрою читання readdevice символьний рядок;

Removewindow – видаляє поточне вікно;

text – повертає екран в алфавітно-цифровий режим;

time (H,M,C,Hund) (integer, integer, integer, integer): (вх,вх,вх,вх),
(вих,вих,вих,вих): (вх,вх,вх,вх): – встановлює системні покази години : H – години, M –хвилини, C – секунди, Hund – соті частки секунди;

(вих,вих,вих,вих): – присвоює змінним покази системного годинника;

write (e1,e2,e3,…eN) (вх) – виводить на поточний пристрій writedevice константи або значення. Кількість аргументів може бути довільною.

1.3.4. Контрольні питання та вправи

1. Дати характеристику основних складових програми на Турбо Пролозі.
2. Основні типи даних Турбо-Прологу.
3. Які є способи задання запитів в середовищі Турбо-Прологу?
4. Принципова відмінність між рядковим та символьним типом даних у Турбо-Пролозі.

5. В базі задані факти:

```
 сума (2).
 сума(3).
сума(a+b).
сума (x+y).
```

Якою буде відповідь на запит **сума (2+3)**?

6. Побудувати предикат, який перетворює список символів в список цілих чисел згідно з абеткою кодування ASCII.

1.4. Рекурсивне подання даних і програм

1.4.1. Деревоподібна форма подання даних і програм

Рекурсія в перекладі з англійської мови означає зворотний рух, повернення. Апарат рекурсії в мовах логічного програмування, як і в мовах функціонального програмування, є надзвичайно потужним і досить популярним засобом в галузі нечислового програмування.

У мові Пролог рекурсія – природний спосіб представлення як структур даних, так і самих програм.

Коли маємо в логічному програмуванні складну структуру, то, як правило, її подають у вигляді дерева, де кожному функтору у відповідність ставиться вершина дерева, а компонентам функтора відповідають гілки, які виходять з вершини. Для прикладу, структури Прологу:

```
Родичі (ельдар, вікторія, альберт)
a+b*c або +(a, *(b, c))
книга (Земля, автор (Ольга, Кобилянська))
```

можуть бути подані такими деревами:

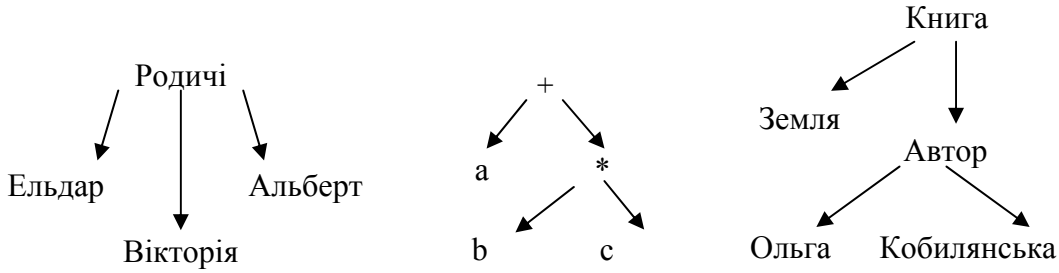


Рис. 1.1. Деревоподібна форма подання даних у Пролозі

Структури будь-якої мови (природної чи штучної) можна подавати у вигляді дерева. Так, в англійській мові існує просте правило (синтаксичне) побудови речень у такий спосіб, що на перше місце ставиться підмет, далі йде дієслівна форма, що включає в себе дієслово і другий іменник. Так речення "Марії подобається книга" може бути подане у вигляді структури Прологу:

речення (підмет (книга), дієслівна форма (дієслово (подобається), іменник (марія)))

Деревоподібне подання набуває вигляду:

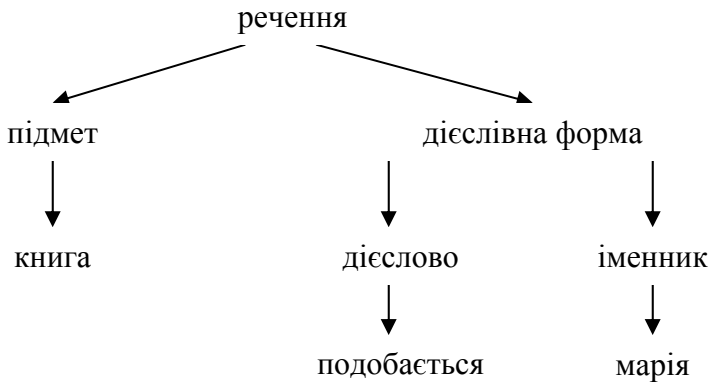


Рис. 1.2. Деревоподібна форма подання речень у Пролозі

Цей приклад ілюструє можливість Прологу розуміти прості речення природної мови. Знаючи, якою частиною мови є кожне слово в реченні, можна за допомогою структур Прологу записати відношення між різними словами у реченні і вести з комп'ютером "розумний" діалог. Ця проблема розуміння мови є важливою і до неї ми ще повернемося. Деревоподібне представлення може застосовуватися і при рекурсивних викликах, що дає змогу встановити зв'язки між зчепленими змінними чи функторами, які зустрічаються декілька разів в одній породжуючій (батьківській) цілі. Наприклад, структура `y(x, z(x, a), w(x, b))` набуває вигляду:

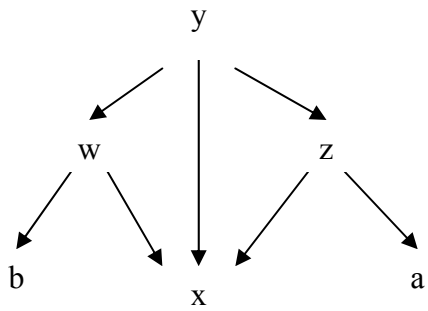


Рис. 1.3. Деревоподібна форма подання структур Прологу

У галузі штучного інтелекту та нечислового подання даних широко застосовується така структура даних, як `список` – впорядкована послідовність елементів, яка може мати довільну довжину. В мові Ліспі список є єдиним способом подання даних. Як елементи логічного списку можуть виступати константи, змінні, структури, які можуть включати в себе інші структури як елементи нового списку нижчого рівня ієрархії. Такий

спосіб подання даних корисний в прикладних застосуваннях, коли апріорі довжина списку не є визначеною і значення самих елементів не є визначеними. В Пролозі списком можна представляти будь-які структури, які дозволено використовувати версією мови програмування. Така форма представлення даних застосовується для складання карт місцевості, дерев синтаксичного розбору речень, математичних об‘єктів таких як графи, формули, функції.

Списки можуть бути подані як спеціальної форми дерева. Список – це будь-який порожній список, що не містить жодного елементу, або структура, складена з двох компонент: голови і хвоста списку. Кінець списку, зазвичай, подають як хвіст, який є порожнім списком. Порожній список записують у вигляді квадратної дужки, що відкриває, за якою йде квадратна дужка, що закриває: []. Голова і хвіст списку є компонентами функтора, який позначається крапкою “.”. Так, список з одного елемента “a” можна подати у вигляді **`.(a, [])`** або у вигляді дерева:

Рис. 1.4. Деревоподібна форма подання списку з одного елементу у Пролозі

Аналогічно список з трьох елементів **`.(a, .(b, .(c, [])))`** як дерево набирає вигляд:

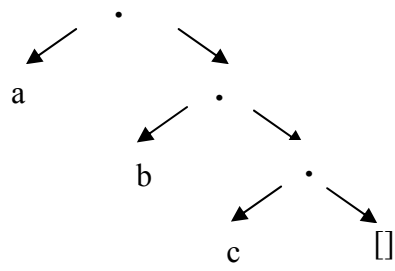


Рис. 1.5. Деревоподібна форма подання списку з трьох елементів у Пролозі

Списки можуть бути подані і в операторній формі, яка для двох описаних випадків набирає вигляд **`a.[]  i  a.(b.(c.[])).`** Оскільки список є правоасоціативним оператором, то другий приклад подання списку записується як **`a.b.c.[].`** При записі вкладених один в одного списків їх зручно подавати у вигляді дерева, що “росте” зліва направо, а гілки звисають вниз. Останній список тут набирає вигляду:

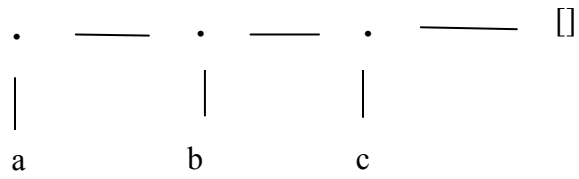


Рис. 1.6. Деревоподібна форма подання вкладених списків з трьох елементів у Пролозі

Така форма дає змогу бачити структуру списку, але часто буває незручною при записі довгих списків. У такому разі для запису списків передбачена ще додаткова форма запису списків. Наприклад: **`[]; [a, b, c]; [a]; [перший, другий, десятий]; [a, v10, [b, c], [x, y]]`**. Змінні, які входять у списки, нічим не відрізняються від змінних, які входять у будь-які інші структури Прологу. При погодженні цільового твердження у будь-який момент може відбутися їх конкретизація і заповнення порожніх місць бажаними даними.

Робота зі списками полягає у поділі їх на голову та хвіст списку. Голова – це перший аргумент функтора “.”, який використовується для конструювання списку. Хвіст – це другий аргумент функтора “.”. При використанні дужкової форми запису списку голова – це перший елемент списку, а хвіст – весь залишок списку, окрім першого елемента. Розщеплення списку на голову і хвіст демонструють приклади, наведені в табл. 1.3:

Таблиця 1.3

Приклади розщеплення списку на голову і хвіст

Список	Голова	Хвіст
`[a, b, c]`	`A`	`[b, c]`
`[a]`	`A`	`[]`
`[цей, дім, новий]`	`Цей`	`[дім, новий]`
`[цей, [дім, новий]]`	`Цей`	`[[дім, новий]]`
`[[цей], дім, новий]`	`[цей]`	`[дім, новий]`
`[x+y, a+b]`	`x+y`	`[a+b]`
`[]`	`-`	`-`

Зауважимо, що пустий список не має ні голови ні хвоста. Для розділення списку на голову і хвіст використовується вертикальна риска. Так, запис **`[X|Y]`** означає список з головою **`X`** і хвостом **`Y`**. При конкретизації вищенаведеної структури з **`X`** порівнюється голова описка, а з **`Y`** – хвіст списка. Наведемо приклади конкретизації двох списків, які зіставляються при

погодженні (табл. 3):

Дужкова форма, як видно з останнього прикладу, дозволяє формувати структури, подібні на списки, але які не закінчуються порожнім списком. Опрацювання таких структур потребує великої акуратності, якщо хвіст списку не є ні списком, ні порожнім списком.

Списки часто застосовують для представлення рядків літер. Якщо рядок взято в подвійні лапки, то він подається у Пролозі списком цілих чисел згідно з абеткою кодування ASCII. Так, стрічка **"system"** в Пролозі набирає вигляд **[115, 121, 115, 116, 101, 109]**.

Таблиця 1.4

Приклади конкретизації списків

Список 1	Список 2	Конкретизація
[x, y, z]	[Іван, читає, книгу]	x=Іван y=читає z=книгу
[пілот]	[X|Y]	X=пілот Y=[]
[X, Y|Z]	[Марії, подобається, книга]	X=Марії У=подобається Z=[книга]
[білий|птах]	[птах|X]	Порівняння неможливе
[цей, білий|птах, X]	[цей, білий|птах, летить]	Некоректний список
[білий|T]	[білий|птах]	T=птах
[білий|T]	[X|птах]	X=білий T=птах

1.4.2. Побудова предиката належності елемента до списку

Нехай набір кольорових олівців поданий у вигляді списку: **[чорний, білий, зелений, червоний, синій].** Для встановлення факту, чи міститься бажаний колір **X** у заданому списку кольорів, необхідно порівняти **X** з головою списку і якщо порівняння не відбудеться, то порівняти з хвостом списку, почергово розглядаючи кожен елемент. Якщо дійдемо до порожнього списку, то бажаного кольору в списку не міститься. Щоб записати цей

алгоритм на Пролозі, слід ввести предикат належності елемента **X** до списку **У**, який назвемо **належить (X, У)**. Факт належності елемента до голови можна записати так:

```
належить (X, [X,_]).
```

або у вигляді правила

```
належить (X, [Y,_]):- X=Y.
```

Щоб закінчити побудову, необхідно перевірити, чи належить X до хвоста списку. Це реалізується правилом:

```
належить(X, [_|Y]):- належить(X, У)
```

Ці два правила визначають предикат належності елемента до списку і вказують спосіб перегляду списку від початку до кінця. У цьому предикаті ми маємо справу з рекурсивною побудовою, оскільки в другому правилі вимагається повторний виклик предиката **належить**.

Будуючи такі предикати, чітко слід простежити за виконанням рекурсивних викликів, щоб гарантувати, що рекурсія закінчиться.

Для предиката є дві граничні умови. Або об‘єкт пошуку є в списку, або його там немає. Перша гранична умова розпізнається першим фактом і у разі успіху пошук елемента закінчується. Друга гранична умова зустрічається, коли в другому аргументі предиката трапиться порожній список.

Для гарантії того, що рекурсія рано чи пізно завершиться успіхом (бажаний колір є в списку) або неуспіхом, необхідно простежити за використанням рекурсії в другому правилі. Відзначимо, що кожен раз при порівнянні цілі з заголовком другого правила йде звертання до коротшого списку, оскільки хвіст завжди коротший за весь список. Очевидно, рано чи пізно станеться одна із подій і або відбудеться порівняння з першим фактом (правилом) або другим елементом предиката буде порожній список. Будь-яка з цих подій приводить до припинення породження цілей. В першому випадку маємо підтвердження наявності елемента у списку. У другому випадку порожній список не може бути розщеплений на голову і хвіст і не відбувається порівняння ні з першим фактом, ні з другим правилом. Отже, пошук елемента **X** у списку **У** закінчується неуспіхом. Наведені міркування демонструють такі приклади на Пролозі:

```
належить (X, [X|]).

належить (X, [-|У]):належить(X, У).
```

? – належить (d, [a, b, c, d, e]).

так

? – належить (2, [3, a, в, 4]).

ні

? – належить (червоний, [білий, червовий, зелений]).

так.

При побудові рекурсивних предикатів слід уникати зациклених визначень. Наприклад:

родич(X, У):дитина(У, X)

дитина (A, B):родич(B, A)

Для погодження цілі **родич** необхідно погодити ціль **дитина**, яка в свою чергу потребує погодження тієї самої цілі **родич**. Оскільки Пролог не зможе знайти інших фактів і правил для її погодження, то такий рекурсивний цикл може закінчитися лише коли буде вичерпаний весь машинний ресурс.

Ще одна проблема, яка виникає при побудові рекурсивних визначень – це лівостороння рекурсія, яка виникає при породженні заголовком правила цілі, яка еквівалентна йому. Наприклад:

людина (X): - людина (У), мати (X, У).

людина (Адам).

На запит

? – людина(Z).

Пролог породжує рекурсивну ціль **людина**, яка, своєю чергою, породжує еквівалентну підціль **людина**. Оскільки погодження нової підцілі здійснюється тим самим правилом, то не має ніякої можливості закінчити рекурсію. Породження еквівалентних підцілей триватиме до повного вичерпання комп‘ютерного ресурсу.

Для наведеного прикладу усунення помилки досягається поміщенням факту перед правилом. Цей евристичний принцип розташування фактів перед правилами, де це є можливим, не завжди спрацьовує для різних значень аргументів.

Розглянемо визначення предикату **список(X)**, який істинний, коли **X** – список:

```
список[A|B]:список(B).

список([]).
```

Це визначення працює вірно для запитів:

```
? – список ([1, 2, 3])

? – список ([])

? – список (a)
```

Але коли сформувати запит:

```
? – список(X)
```

то відбудеться зациклення. Уникнути зациклення можна, задавши таке визначення:

```
список 1([])

список 1([_|_])
```


Останнє визначення уникає зациклення, коли аргумент предиката є змінною. Але воно не є строгим тестом списку, оскільки дає змогу сприймати як список структуру, яка не закінчується порожнім списком. Тому не варто передбачати, що якщо в Пролозі наявні всі факти і правила для розв‘язання конкретної задачі, то він їх всі виявить. Кожен раз, розробляючи програму на Пролозі, необхідно простежувати, як здійснюється пошук в базі даних, як відбувається конкретизація змінних при заданій структурі аргументів.

1.4.3. Програма підтримки діалогу мовою логіки

Використаємо спискову форму представлення та рекурсивний спосіб задання правил для побудови програми підтримки усного діалогу мовою логічних висловлювань. Метою програми буде перетворення речень, які задаються у вигляді списку слів англійської мови, в інші речення, які є відповіддю на сформульовані запити і мають логічний зміст. Запис таких речень у звичній для користувача формі є предметом окремих досліджень і тут розглядатися не буде.

Фрагмент цієї програми міг би підтримувати такий діалог:

Користувач: you are a computer /*Ви – комп‘ютер*/

Пролог: i am not a computer /*Я – не комп‘ютер*/

Користувач: do you speak ukraine /*Ви розмовляєте українською*/

Пролог: no i speak english /*Ні, я розмовляю англійською*/

Для реалізації цієї програми слід виконати таку послідовність дій:

1. Ввести речення користувача з термінала.
2. Замінити кожне входження слова

- **you** на **is;**
- **are** на **am not**;
- **ukraine** на **english**;
- **do** на **no**.

Якщо необхідне розширення діалогу, збільшується набір для заміни слів, а решта програми залишається без змін і реалізує процедуру перетворення слів. Очевидно, слід означити предикат **перетворити(X, У)**, де **X, У** – це списки, елементи яких є відповідними словами вхідного речення користувача та словами відповіді комп‘ютера. Після побудови цього предиката на запит користувача

перетворити ([do, you, speak, english], Y)

Пролог мав би відповідати

Y=[no, i, speak, english]

Перш ніж будувати предикат, розглянемо випадок порожнього списку, оскільки речення подані у вигляді списків. Оскільки порожній список не потребує ніяких перетворень, то це запишемо у вигляді факту:

перетворити ([], [])

Побудову предиката перетворення можна звести до таких дій:

1. Замінити голову вхідного списку відповідним словом і помістити його в голову вихідного списку.
2. Здійснити перетворення хвоста вхідного списку і помістити його в хвіст вихідного списку.

3. Якщо досягнутий кінець вхідного списку (пустий список), то ніяких перетворень з вихідним робити не потрібно, тобто закінчити його порожнім списком.

Замінити вхідні слова вихідними можна, створивши в базі даних набір фактів **`замінити (X, У)`**. Очевидно, таку базу слід закінчити фактом-капканом, якщо зустрінуться слова, які залишаються без змін. Роль такого капкана може виконати факт **`замінити (X, X)`** або правило: **`замінити(X, У):-X=У`**.

Отже, фрагмент програми перетворення речень для підтримки діалогу у вигляді змістовних списків слів можна подати так:

```
замінити (you, i).
замінити (are, [am, not]).
замінити (ukraine, english).
замінити (do, no).
замінити (X, Y):-X=Y.
перетворити ([], []).
перетворити ([X|Y], [X1|Y1]):-замінити (X, X1), перетворити (У, У1).
```

Відзначимо, що вихід на граничні умови відбувається, коли вхідний список залишається порожнім (тобто переглянуті всі слова) або переглянуті всі факти у **`замінити`**. У першому випадку до вихідного списку додається порожній список, а в другому – замінюється слово або слово записується без зміни за допомогою використання факту-капкану.

1.4.4. Контрольні питання та вправи

1. Дати визначення поняття рекурсії в логічному програмуванні та навести конкретні приклади деревоподібного представлення рекурсивних структур.
2. Способи представлення списків у Пролозі.
3. Дати означення голови та хвоста порожнього списку.
4. Побудувати предикат обертання списку на довільному рівні ієрархії.
5. Основні проблеми побудови рекурсивних предикатів.
6. Побудувати предикат перевірки того, що його аргумент є списком при довільній структурі аргументу.
7. Записати факт заміни слів за допомогою відповідного правила.
8. Що таке факт-капкан та яка доцільність його використання.

9. Написати фрагмент-програми для впорядкування слів в алфавітному порядку.

10. Як можна реалізувати взаємну заміну слів у програмі підтримки усного діалогу, не збільшуючи розміру бази даних вдвічі?

1.5. Механізм повернення та відсічки в логічному програмуванні

1.5.1. Механізм перепогодження цільових тверджень

За наявності деякого цільового твердження (цілі) в базі даних Прологу можуть виникати такі ситуації:

1. Спроба довести погодженість цілі з базою даних, починаючи з її вершини. Необхідно виділити два випадки:

а) Знайдений факт (або заголовок правила), який може бути порівняний з ціллю. Це місце в базі даних відмічається спеціальним маркером і одночасно змінні конкретизуються. Якщо ж порівнюється з заголовком правила, то насамперед робиться спроба погодити підцілі, які породжені цим правилом. Пролог здійснює погодження підцілей поступово, рухаючись зліва направо від крайньої лівої підцілі до крайньої правої в тілі правила.

б) В базі даних Прологу немає ні факту, ні правила для порівняння з цільовим запитом. Тоді можна стверджувати, що спроба довести погодженість цілі з базою даних зазнала невдачі і відбувається спроба перепогодити цільове твердження іншим способом.

2. Спроба перепогодити цільовий запит приводить до руху маркера знизу вверх по базі даних.

Якщо при перепогодженні деякого цільового твердження не знаходиться альтернативного розв’язку, то Пролог робить спробу перепогодити всі попередні цілі, повертаючись до самої вихідної. Одночасно всі змінні розконкретизовуються, стають невизначеними і всі результати знищуються. Маркер починає рух знову вниз по базі даних, намагаючись по новому погодити цільові твердження, можуть знову виникати ситуації а) або б).

Розглянемо більш детально механізм перепогодження цілей на конкретних прикладах.

Найпростіший випадок, коли породжується множина розв’язків, ілюструє така база даних:

```
батько (марія, іван).
батько (петро, іван).
батько (ігор, михайло).
батько (іра, михайло).
```

Тоді Пролог на запит

```
?-батько (X, Y).
```

Видасть відповіді:

```
X=марія     Y=іван
X=петро     Y=іван
X=ігорY=михайло
X=іра       Y=михайло
```

Розглянемо цікавіший випадок, коли в запиті є дві підцілі, для яких можливі альтернативні розв'язки:

```
можлива_пара (X,Y) – хлопець (X), дівчина (Y).
хлопець(іван).          дівчина(іра).
хлопець(петро).         дівчина(ольга).
хлопець(дмитро).        дівчина(оксана).
```

Відповіді Прологу на запит:

```
?-можлива_пара (X,Y).
```

будуть такі:

```
X=іванУ=іра     X=петро    У=ольга  X=дмитро
У=ольга
X=іванУ=ольга   X=петро    У=оксана
X=іванУ=оксана  X=дмитро   У=іра
X=петро     У=іра      X=дмитро   У=ольга
```

Ці приклади є доволі простими, оскільки породжують скінченну множину розв'язків. В прикладних задачах, як правило, апріорі невідома кількість розв'язків, оскільки їх може існувати цілий континуум. В таких випадках використовується рекурсивне представлення програми.

Розглянемо предикат, який породжує множину цілих чисел:

```
ціле_число (0).
ціле_число (X):-ціле_число (Y), X=Y+1.
```

На запит

```
?-ціле_число (Z).
```

буде породжуватися множина цілих чисел від 0 доти, доки не вичерпається весь машинний ресурс.

Більшість правил, які мають прикладне застосування, породжують цілий континуум розв'язків, що пов'язано з наявністю в них великої кількості неконкретизованих змінних.

Так, правило визначення належності деякого об'єкта до списку:

```
належить (X, [X|_]).
належить (X, [_|Y]):- належить (X, Y).
```

на запит:

?- належить (a, X).

породжує цілий континуум розв'язків

[a|_];

[_a|_];

[__a|_];

.....

Очевидно, щоб успішно опрацювати такі розв'язки, зупиняючись на бажаному, необхідно мати відповідний механізм. Такий механізм в Пролозі передбачений і він дає змогу вказати, які з раніше зроблених розв'язків не слід перепогоджувати при поверненні по ланцюжку погодження цільових тверджень. Отже, цей механізм, який має назву механізму відсічки, дає змогу відсікати можливі альтернативні розв'язки і зупинятися на бажаному.

1.5.2. Суть механізму відсічки у логічному програмуванні

Слово "відсічка" походить від англійського слова cut, що означає скорочення. Але термін "відсікання" використовувався в перших роботах з логічного програмування і використовується досі.

Можна виділити, принаймні, дві причини, які зумовлюють необхідність використання цього механізму для побудови логічних програм:

- програма виконуватиметься швидше, якщо не будуть перебиратися розв'язки, про які відомо, що вони не несуть більше ніякої корисної інформації;
- забезпечується значна економія пам'яті, оскільки немає потреби запам'ятовувати місця погодження цілей, для яких зникає потреба їх перепогоджувати;
- іноді увімкнення цього механізму означає перехід від непрацюючої програми до працюючої, що має прикладне значення.

Згідно із синтаксисом мови Пролог для введення в програму механізму відсічки використовується предикат **!**, який не має аргументів. Цей предикат

завжди погоджений з базою даних і не потребує порівняння чи повторного погодження. Окрім цього цей предикат має ще й сторонній ефект, який змінює перепогодження цілей. Суть ефекту в тому, що маркери деяких цілей стають недоступними і для них неможливо знайти нове порівняння.

Як приклад розглянемо базу даних на Пролозі, яка містить інформацію про наявність книг, хто і які книги взяв і коли закінчується термін їх повернення. Один із запитів, який можна сформулювати – це які послуги можна надавати читачам. Допустиме користування каталогом та отримання довідок є доступним для кожного читача. Назвемо ці послуги основними. Додаткові послуги, такі, як абонемент та міжбібліотечний абонемент, можна надавати читачам, які не мають боргів. Очевидно, слід сформулювати правило, згідно якого читач, що не повернув книгу в зазначений термін, не може отримувати додаткові послуги доти, доки книга не буде повернена. Частина програми, яка використовує це правило, записана нижче на псевдо-Пролозі:

```
    послуги(Читач, Вид_послуг):-
книга_не_повернута(Читач, Книга),
    !, основні_послуги(Вид_послуг).
    послуги(Читач, Вид_послуг):-
загальні_послуги(Вид_послуг).
    основні_послуги(користування_каталогом).
    основні_послуги(користування_довідкою).
    додаткові_послуги(абонемент).
    додаткові_послуги(міжбібліотечний_абонемент).
    загальні_послуги(X):- основні_послуги(X).
    загальні_послуги(X):- додаткові_послуги(X).
    книга_не_повернута('Іванов',книга54).
    ............
    читач('Іванов').
    читач('Петров').
    ............
```

В першому правилі цієї програми використовується відсічка після погодження цілі **книга_не_повернута.** З цього моменту всі прийняті рішення з моменту вибору твердження **послуги** заморожуються і ніяк змінені бути не можуть. Це логічно правильно, бо після цього буде погоджено ціль **основні_послуги** і більш ніякі інші альтернативні

рішення ні до цілі **книга_не_повернута**, ні до цілі **послуги** розглядатися не будуть.

Якщо читач не є боржником, ціль **книга_не_повернута** не може бути погоджена з базою даних і маркер програми на Пролозі переміститься до другого правила визначення предикату **послуги**, згідно з яким можуть бути надані всі види послуг.

У розглянутому прикладі відсічка привела до скорочення всіх можливих альтернатив розв'язків (рішень), які прийняті після вибору цільового твердження **послуги**. Цільове твердження (заголовок правила), яке приводить до використання механізму відсічки називається батьківським. З формального погляду ефект механізму відсічки можна сформулювати так:

за наявності відсічки в ролі цільового твердження система не має можливості змінити рішення, які прийняті з моменту виклику батьківського твердження і всі можливі альтернативи прийнятим рішенням не розглядаються. Отже, спроба перепогодження цільового твердження, що є між батьківською ціллю і власне відсічкою, завжди завершується неуспіхом.

Механізм відсічки можна інтерпретувати як деякий кордон, до пересікання якого всі підцілі можуть погоджуватися і перепогоджуватися довільну кількість разів, аналогічно до того, як це відбувається без предиката !. Але якщо виникає ситуація, при якій відбудеться повернення системи через відсічку справа наліво, то не буде здійснено ніяких спроб до перепогодження цілей ліворуч від відсічки. В цьому випадку доведення погодженості кон'юнкції цілей зазнає невдачі.

Відзначимо, що в Пролозі є можливість вказувати альтернативні рішення в рамках одного правила, використовуючи вбудований предикат **;** (інтерпретується як або). На вибір альтернатив механізм відсічки діє у такий самий спосіб, як і на будь-які інші підцілі. Тобто за наявності відсічки всі зроблені або-вибори фіксуються з моменту виклику правила, яке містить відсічку після повторного переходу через неї справа наліво.

1.5.3. Основні випадки застосування механізму відсічки

З суто методологічного погляду зору для розуміння причин використання відсічки можна виділити три основні випадки його використання:

1) в такому разі Пролог-системі вказується, що зникає правило

для підтвердження цілі і подальший пошук альтернатив необхідно припинити;

2) цей випадок використовується для негайного припинення пошуку альтернативних рішень, оскільки підтвердження цілі є неможливим. Тут використовується поєднання відсічки і предикату **fail,** який є завжди погоджений з базою і означає неуспіх цільового твердження, в якому зустрівся цей предикат.

3) у такому разі відсічка вимагає негайного припинення пошуку альтернатив, оскільки знайдений єдиний розв'язок і неможливо знайти інші.

Відзначимо, що у всіх цих випадках суть механізму залишається однією і тією самою.

У Пролозі часто для описання одного предиката застосовується кілька тверджень з різними типами аргументів. В реальних ситуаціях не завжди можливо передбачити види всіх можливих аргументів, а тому доцільно вводити правило-капкан для неврахованих чи непередбачених видів аргументів. Як приклад розглянемо предикат визначення суми N чисел **сума(N,S),** де N – ціле число, X – значення суми цих чисел:

сума(1,1):-!.

сума(N,R):- N1=N-1, сума(N1,R1),R=R1+N.

Наведений приклад відповідає вимозі 1, коли вказується Прологу, що при досягненні N=1 необхідно припинити будь-який пошук альтернатив, оскільки досягнуто граничне значення параметра. Якщо N≠1, спрацьовує друге правило, яке рекурсивно викликає само себе, присвоюючи результат підсумовування чисел змінній R. Оскільки змінна N1 кожен раз зменшується на 1, то рано чи пізно буде здійснене погодження цілі **сума(N1,R1)** першим правилом, далі зустрінеться відсічка і пошук альтернатив припиниться.

Загальний принцип полягає в тому, що використання відсічки у Пролозі вказує на вибір єдиного правильного правила. Це може бути замінене застосуванням предиката **not(X)**, який є істинним, якщо твердження X не може бути погоджено з базою даних. Тому варіант предиката **сума(N, R)** може бути записаний без відсічки:

```
сума(1,1).
сума(N,R):-not(N=1), N1=N-1, сума(N1,R1), R=R1+R.
```

Використання цього предиката **not** замість відсічки властиве доброму стилю програмування, оскільки програми з відсічкою є значно складнішими для розуміння. Однак часто використання **not** призводить до втрати

ефективності програми.

Так, в програмі

```
A: - B, E.
A:- not(B), D.
```

для успішного закінчення буде зроблено дві спроби доведення твердження `B`. Щоб цього не було, слід використовувати відсічку

```
A:- B, !, E.
A:-D.
```

в результаті чого програма стає ефективнішою.

Отже, необхідно порівнювати переваги ясності програми з перевагами її швидкого виконання.

1.5.4. Контрольні питання та вправи

1. Суть механізму повернення та перепогодження цільових тверджень.
2. Яким чином можна інтерпретуватиї механізм відсічки?
3. Які є варіанти заміни механізму відсічки?
4. Спільні та відмінні риси механізму відсічки у та перепогодження
5. Записати предикат **`сума(N,R)`** так, щоб він здійснював сумування яяк відємних, так і додатних цілих чисел.
6. Що спільного і відмінного у предикатах **`!`** і **`fail`**?
7. Навести приклад предикату, який рекурсивно звертається до себе.
8. Записати предикат паліндром, який встановлює чи слово, що задано у вигляді списку читається аналогічно і в зворотному напрямі.

1.6. Типові способи використання механізму відсічки та повернення

1.6.1. Комбінація відсічки та предиката fail

Розглянемо другий випадок використання відсічки в комбінації з вбудованим предикатом **`fail`**, який не має аргументів. Дія цього предиката полягає в тому, що він завжди непогоджений з базою даних, а тому його наявність в кон'юнкції цілей приводить до активації механізму повернення.

Якщо перед **`fail`** поставити предикат **`!`**, то природний хід погодження цілей між батьківською ціллю і **`fail`** буде змінений і всі цілі вважаються непогодженими і немає ніякого шансу отримати позитивний результат. Ця комбінація **`!`** і **`fail`** має широке практичне використання.

Розглянемо, як доцільно використовувати цю комбінацію для написання програми нарахування податку, який необхідно заплатити тій чи іншій людині. Визначимо предикат **`сер_податоплатівник(X),`** який означатиме, що об'єкт X є середнім податкоплатником, якщо він має прибуток, який не перевищує певної величини, а також більший за певний мінімум. Очевидно, окремо слід виділити випадок, коли людина є іноземцем і не може бути середнім податкоплатником через податкові обов'язки щодо своєї країни. Фрагмент такої програми міг би виглядати так:

```
сер_податоплатівник(X):-іноземець, !, fail.
сер_податоплатівник(X):-дохід(X,D), D>200, D<2000.
дохід(X,Y):-заробітна_плата(X,Z),
дохід_від_капіталовкладень(X,W), Y=Z+W.
дохід_від_капіталовкладень(X,W):-.....  .
```

Перше правило у разі, якщо людина є іноземцем, не дає змогу по-іншому погодити ціль, а наявність предиката **`fail`** призводить до того, що вся ціль є неуспішною. Друге правило визначає середнього податкоплатника при знаходженні всіх видів доходів в певному діапазоні, який диктується Податковим кодексом.

Цікавий приклад використання **`!`** і **`fail`** містить визначення предикату **`not(X).`** Незважаючи на те, що це вбудований предикат, наведемо його визначення:

```
not(X):-call(X),!,fail.

not(X).
```

Предикат **`call`** інтерпретує свій аргумент і намагається погодити його з базою даних. Якщо погодження відбудеться, то відсічка (**`!`**) не дасть змоги перепогодити твердження, а наявність **`fail`** забезпечує неуспіх всієї цілі. У разі непогодження твердження X спрацьовує другий факт і предикат `not` є істинним.

1.6.2. Типові випадки використання механізму генерування розв'язків та перевірки

В якості ще одного прикладу використання відсічки розглянемо третій випадок породження і перевірки варіантів. Розглянемо програму ділення цілих чисел, використовуючи операції підсумовування та множення.

```
розділити(N1,N2,R):-ціле_число(R),          D1=R*N2,
D2=(R+1)*N2,
                                N1>=D1, N1<D2,!.
```

В цьому правилі використано рекурсивний предикат **ціле_число** для породження цілих чисел. Його визначення дано раніше. Інші цільові твердження виконують функцію перевірки. Очевидно, результатом може бути одне-єдине число. Незважаючи на те, що генератор **ціле_число(N)** породжує цілий континуум передбачуваних розв'язків, як тільки виконуються нерівності, зустрічається відсічка й інші альтернативи розв'язання стають неможливими.

При використанні відсічки необхідно добре усвідомлювати, як будуть використані правила з відсічкою, оскільки її незаплановане використання може привести до непередбачуваного результату. Розглянемо приклад незапланованого використання відсічки. Розглянемо предикат визначення кількості родичів людини:

```
число_родичів (адам, 0):-!.
число_родичів (єва, 0):-!.
число_родичів (X, 2).
```


Адам і Єва не мають родичів, у всіх інших людей їх є двоє. Але якщо тепер звернутися до Прологу із запитанням:

?-число_родичів(адам, 2).

то відповідь буде:

так

У такому разі покращення можна досягти, якщо змінити визначення так:

```
число_родичів (адам, 0).
число_родичів (єва, 0).
число_родичів (X, 2):-X\=адам,Y\=єва.
```

Але знову сформулюємо запит:

```
?-число_родичів (X, Y)
```

наше визначення не дасть можливості видати всі результати. Звідси можна зробити висновок, що надійне використання механізму відсічки можливе лише у разі повного уявлення про його використання у сформульованих правилах.

1.6.3. Контрольні питання та вправи

1. Суть механізму повернення та перепогодження цільових тверджень.
2. Дати варіанти інтерпретації механізму відсічки.
3. Характеристика способів використання відсічки у принципах зчитування.
4. Модифікувати предикат **сума(N,R)** у такий спосіб, щоб він був коректним у випадку від'ємних чисел.
5. Обґрунтувати доцільність використання комбінації **!** і **fail**.
6. Записати фрагмент програми **сер_податоплатівник(X),** використовуючи предикат **not** замість **!.** Обґрунтувати доцільність або недоцільність такого запису.
7. Дати характеристику проблем, пов'язаних з використанням механізму відсічки.
8. Що відбудеться при перепогодженні цілей, якщо з першого твердження предиката **сума(N,R)** видалити **!**?
9. Записати предикат **викл(E,X,Z)**, який виключає перше входження елементу **E** в список **X** і формує список **Z**.
10. Модифікувати предикат **викл** з попередньої вправи так, щоб він забезпечив всі видалення **E** зі списку **X**.

1.7. Методи організації повторень та рекурсії при побудові логічних програм

1.7.1. Формування повторень у вигляді правил

Часто в Пролозі необхідно виконати одну і ту саму задачу декілька разів. У програмах на Турбо-Пролозі для цих потреб використовують правила, які реалізують механізм повернення та рекурсії. Причому ці механізми використовують вбудовані предикати **fail, cut** для управління процесом повернення. Правила, які виконують повторення, застосовують механізм повернення, а правила, які виконують рекурсію, застосовують самовиклик імені правила. Вигляд правила для реалізації повторень такий:

```
repete-rule:- / * правило повторення*/
< предикати правила>
fail      /*невдача*/
```

Вигляд правила, яке реалізує рекурсію:

```
recursive_rule : - /*рекурсивне правило*/
<предикат і правила>, recursive_rule.
```

Рекурсивне правило в тілі правила містить саме себе, у такому разі ім'я **recursive_rule,** забезпечує тим самим рекурсивне виконання всієї програми під іменем **recursive_rule.** Водночас правила повтору завершуються вбудованим предикатом **fail.** Він завжди непогоджений з базою даних і вимагає погодження всіх під цілей, в тілі яких він зустрівся. По суті, правила повтору і рекурсій мають один і той самий результат, хоча і алгоритм їх виконання є різним. Кожен з них має свої недоліки і переваги.

При використанні зовнішньої цілі змінні набувають всі можливі значення, які можуть взяти з бази даних одне за другим. Якщо ж ми використовуємо внутрішню ціль для побудови програми, то пошук розв'язків припиняється після першого вирахування цілі. У такому разі необхідно змусити внутрішні уніфікаційні програми повторно здійснювати пошук всіх можливих розв'язків.

Приклад:

```
 /* програма: міста США*/
domains
 name=symbol
predicates
citios (name)
show_citios
goal
write ("here are the cities : "), nl,
sow_sities.
Clauses
cities ("ATLANTA").
cities ("BOSTON").
cities ("SAN DIEEGO").
cities ("SAN ANTONIO").
show_sities:- cities (s),
write ("      ", s), nl,
 fail.
```

Відсутність предиката **fail** дало би змогу вивести лише значення першого міста **"ATLANTA"** і на тому пошук цілей припинився. Оскільки **fail** завжди неуспішний, то пошук триватиме, поки не буде переглянуте останнє твердження бази даних.

Правило повторення може бути визначене і користувачем. Вигляд його такий:

```
repeat  /* повторити*/
repeat :- repeat .
```

Перший **repeat** є твердженням, який оголошує предикат **repeat** істинним. Оскільки він не створює підцілей, то завжди є успішним. Другий **repeat** – це правило, яке використовує себе як компоненту в тілі правила (третій **repeat**). Третій **repeat** завжди погоджується з першим фактом, а тому і правило є завжди успішним. Це правило широко використовується в програмах для побудови рекурсивних правил. Як приклад запишемо програму **"exo"**, яка після введення рядка з клавіатури дублює її на екран і чекає наступного введення. Якщо ввести **stop**, то програма закінчиться.

```
/* програма: exo */
domains
name=symbol
predicates
wrt_inf.
repeat
do_echo
out (name)
goal  wrt_inf,  do_echo.
clauses
repeat.
repeat :-repeat.
wrt_inf:-  nl,write("Введіть імена"),nl,write ("Я повторю їх"),nl,write ("Щоб зупинити мене введіть stop"), nl.
            do_echo:-
repeat, readln (N), write (N), nl, out(N),!
out (stop) :- write ("Все зроблено").
out (_) : - fail.
```


Правило **repeat** є першим в розділі тверджень програми **"exo"**. Друге правило видає інформацію користувачу. Третє правило **do_echo** є правилом повтору, яке визначене користувачем. Його перша компонента **repeat**

викликає повторне виконання всіх компонент, які слідують за ним. Предикат **readln (N)** зчитує з клавіатури рядок і видає його на екран – моделює **"exo"**, останнє правило **out (N)** має два можливих значення, якщо введений рядок має значення **stop**, то правило успішне. Появляється фраза **"Все зроблено"** на екрані монітора і повторення закінчується. При введенні іншого рядка результатом виконання програми є предикат **fail** і відбудеться повернення до правила **repeat**.

Правило повтору є досить гнучким засобом програмування і, будучи однією з компонент деяких правил, забезпечує циклічне виконання функцій цього правила. Можна використовувати більше від одного правила повтору:

```
do_two. thing s:-
repeat 1
<тіло повтору>,
<умови виходу 1>,!,
repeat 2
< тіло повтору>,
< умова виходу 2>,!,
repeat 1.
repeat 1:- repeat 1
repeat 2.
repeat 2:- repeat 2.
< правило для умови виходу 1>.
< правило для умови виходу 2>.
```


Під час визначення правила повтору можна використовувати будь-яке інше ім’я як альтернативу **repeat**:

```
interate.
interate :-interate.
     recurse.
recurse:- recurse.
```

Цей метод повторення найефективніший для реалізації доступу до бази даних, для організації виведення на екран, для формування меню.

1.7.2. Універсальний спосіб організації рекурсії

Правила, які не використовують правил повтору як компоненти, є найзагальнішим способом організації рекурсії, який використовується для обробки файлів з різними структурами даних, так і для проведення

математичних обчислень.

Правило рекурсії – це таке правило, яке містить саме себе в якості компоненти.

Таке правило є рекурсивним:

```
wrt_stn : /* видати стрічку */
write   ("Приклад простої рекурсії"), nl,
wrt_stn.
```

Правило складене з трьох компонент. Перші дві видають рядок **"Приклад простої рекурсії"** і переводять курсор на новий рядок. Третя компонента – це саме ім'я правила. Щоб бути успішним, воно мусить задовольняти саме себе. Це призводить до видачі на екран рядка і зміщення курсора на початок нового рядка екрана. Процес буде продовжуватися до нескінченності. Це приводить до переповнення стека, що є небажаним, бо може призвести до втрати інформації. Уникнути такої ситуації можна, розширяючи розмір стека, використовуючи опцію miscellaneous setting в меню setup (встановлення).

Щоб уникнути нескінченної рекурсії, треба передбачити предикат закінчення рекурсії, який містить умову виходу з рекурсії. Це може бути правило, в якому йде зчитування символів і при зчитуванні символу **"#"** рекурсія закінчується:

```
read_str:-
readchar (c) , c <>'#', write ( c), read_str.
```

Перша компонента правила – це вбудований предикат Турбо-Прологу, який забезпечує зчитування символу. Наступне правило перевіряє, чи символ є **"#"**. Якщо ні, то правило успішне, символ видруковується на екран і повторно викликається **read_str.** Цей процес продовжуватиметься доти, доки внутрішня перевірка не виявить недопустимий символ **"#"**.

Крім простої рекурсії, при побудові програм використовується узагальнене правило рекурсії, яке складається з кількох тверджень, які можуть бути виконані, і умов завершення їх виконання. Вигляд узагальненого правила рекурсії такий:

```
< ім'я правила рекурсії > :- < список предикатів
 < умова виходу >
< список предикатів >
< ім'я правила рекурсії >
 < список предикатів >.
```

В цьому правилі є п'ять компонент. Перша – це група предикатів. Успіх чи невдача будь-якого з них на рекурсію не впливає. Наступна компонента – умова виходу, в разі її неуспіху припиняє рекурсію або продовжує її в разі успіху. Третя компонента – список предикатів, як і в першому випадку, на рекурсію не впливає. Наступна компонента – ім'я рекурсивного правила. Успіх цього правила викликає рекурсію. П'ята компонента також отримає певне значення, яке буде поміщене в стек під час виконання рекурсії.

Як приклад розглянемо правило рекурсії для друкування ряду цілих чисел від 1 до 8:

```
write_n (8).
write_n (N) : - N<8, write (N), nl, N1 =N+1, write_n (N1).
```

При формуванні запиту **write_n (N1)** спочатку відбувається порівняння з першим правилом **write_n (8).** Оскільки 1 не дорівнює 8, порівняння неуспішне. Після цього йде порівняння підцілі з заголовком правила **write_n (N).** На цей раз порівняння успішне і змінній **N** присвоюється значення 1. Оскільки 1 менше від 8, то правило успішне, змінній **N1** присвоюється значення 2 і рекурсивно викликається заголовок правила програми. Таким чином, цикл буде продовжуватись доти, доки змінна **N** не набуде значення, що не дорівнює 8. В цей момент ціль погоджена, правило успішне, програма закінчує свою роботу.

Ще один приклад рекурсивної програми – це визначення факторіалу від деякого числа **N**. Для його визначення використовується послідовний добуто чисел від 1 до **N**. Правило рекурсії для визначення факторіалу має вигляд:

```
fact ( 1, 1): - !
fact ( N, R ) :- N1=N-1, fact ( N1, R1 ), R= N*R1.
```

Якщо сформувати запит **fact (7, R)**, то отримаємо **R=5040**. Як ще один приклад розглянемо рекурсивну програму впорядкування слів в алфавітній послідовності, яка має широке прикладне використання.

1.7.3. Алгоритм програми впорядкування слів на мові логіки

Розглянемо алгоритм побудови предикату, який назвемо **менше** і він є істинним, якщо **X** і **Y** є атомами і **X** у алфавіті передує **Y**. Так предикат **менше (вікна, двері)** є істинний, а **менше (ліс, дім)** хибний. Хибний і предикат **менше (картина, картина).** Порівнюючи два слова, ми порівнюємо їх послідовно, буква за буквою і визначаємо, яка з наступних

умов виконується:

1. Досягнутий кінець першого слова, але не досягнутий кінець другого слова. При такій ситуації предикат є істинним.

2. Чергова буква в першому слові передує відповідній букві в другому слові**: менше (слово, слон).** Буква **"в"** в слові **"слово"** передує букві **"н"** в слові **"слон".**

3. Літера в першому слові збігається з літерою в другому слові. Очевидно, слід порівняти залишки слів. Наприклад, якщо дано **менше (олівець, одиниця)**, то оскільки перші букви обох компонент однакові, необхідно порівняти залишки слів **(лівець, диниця)**.

4. Одночасно досягнутий кінець першого і другого слів. Очевидно, предикат хибний.

5. Опрацьовані всі букви першого слова, але залишились ще букви другого слова. Наприклад **менше (алфавіт, алфавітний)**. В такій ситуації предикат **менше** повинен бути фальшивим.

Після того як сформульований алгоритм програми, нескладно записати його на Пролозі. Будемо представляти слова у вигляді списку літер (цілих чисел з деякого діапазону). Для цього потрібен спосіб перетворення атома в список цілих чисел. Цю функцію покладемо на предикат, який назвемо **атчис(Х,У)**. Цільове твердження **атчис(Х,У)** погоджується з базою даних, якщо набір символів, які є значенням **Х** відповідає списку цілих чисел згідно з абеткою кодування ASCII. Наприклад:

```
?- атчис(apl,S).
S = [97,108,112]
?- атчис(S, [97,108,112].
S  = apl
```

Цей предикат реалізується так:

```
атчис(X,[H|S])      :-          frontchar(X,H1,S1),
char_int(H1,H), атчис(S1,[H1|S1),
приєднати(H1,S1,S).
 приєднати([],S1,S1).
 приєднати([H|X],S1,[H|S]) :- приєднати(X,S1,S).
```

Тоді першим твердженням в визначенні предикату **менше** є правило:

```
менше (X, Y) :- атчис (X, L), атчис (Y, M), менше_1
(L, M).
```

Предикат **менше_1** повинен містити 5 тверджень вищеописаного

алгоритму впорядкування слів в алфавітній послідовності. Перше твердження є істинним, якщо перший аргумент порожній список, а другий – довільний список, але не порожній:

```
менше_1 ([   ], [ _  | _-]).
```

Друга умова має вигляд:

```
менше_1 ([ H | _ ], [Y  | _  ] :- H<Y.
```

Третє твердження запишемо наступним чином:

```
менше_1 ([ A | X ], [B | Y]) : - A=B, менше_1(X, Y).
```

Останні два твердження є фальшиві і для них можна не передбачати ніяких предикатів.

Отже, наша програма має вигляд:

```
менше (X, Y) :- атчис(X, L),  атчис(Y, M),  менше_1 (L, M).
менше_1 ( [  ], [ _   | _ ].
менше_1 ( [ H | _ ], [ Y | _ ] : - X < Y.
менше_1 ( [ D | Q ], [ R | S ]) : - D=R, менше_1 (Q, R).
```


В цій програмі предикат **атчис (X, L)** визначається за допомогою вбудованих предикатів **frontchar (S, S1, S2)**, який записує перший символ з **S** в **S1**, а залишок символьної змінної **S** в **S2** та предиката **char_int (X, Y)**, який визначає код ASCII символу **X** і присвоює його змінній **Y**.

1.7.4. Контрольні питання та вправи

1. Навести приклад структури Прологу для задання універсального правила рекурсії.
2. Навести запис універсального правила повторення.
3. Записати фрагмент програми перетворення списку символьних імен в послідовність рядка, символьні імена якого розділені пробілами.

1.8. Робота з файлами в середовищі Турбо Прологу

1.8.1. Предикати роботи з файлами

Робота з даними, що містяться в файлах називається файловою обробкою. До типових операцій над файлами належать створення файла, запис у файл і читання з файла, модифікація існуючого файла і дописування в файл. Турбо-Пролог має потужні можливості для зручної і ефективної обробки даних, використовуючи вбудовані предикати для роботи з файлами.

В Турбо-Пролозі є такі предикати для роботи з файлами:

deletefile – знищення файла;

save – збереження файла;

renamefile – перейменування файла;

existfile – тест на перевірку існування файла з цим іменем;

flush – викидання даних з внутрішнього буфера;

disk – вибір дисковода і шляху доступу до нього;

dir – видача каталогу директорії.

Для використання файлів в програмі необхідно описати їх в розділі доменів. Цей опис для одного файлу виглядає так

```
file = datafile.
```

Цей опис відрізняється від загальноприйнятого, оскільки тут тип домена задається з лівої сторони. Ім'я файла **datafile** є логічним іменем і в програмі з ним може бути зв'язане ім'я файла DOS. Якщо в програмі вводиться кілька файлів з відповідними символьними іменами, то вони розділяються символом **';'**:

```
file = datafile1; datafile2; datafile3;
```

Для того, щоб здійснювати запис в файл, його необхідно створити. В Турбо-Пролозі файли створюються за допомогою предиката **openfile:**

```
openfile(datafile,"FILE.DAT").
```

де **datafile** – це введений користувачем файловий домен, а **FILE.DAT** – ім'я файла в DOS. Цей предикат встановлює зв'язок між вказаним іменем та логічним іменем і зберігає його до закриття файла.

Відзначимо, що якщо файл **"FILE.DAT"** вже був присутній в каталозі, то виклик **openfile** призведе до втрати вмісту файла, тому, щоб

застрахуватися від можливої втрати даних, необхідно перевірити, чи файл вже присутній предикатом

```
existfile ("FILE.DAT")
```

і вжити відповідні заходи, якщо предикат є успішним.

Для призначення файла як пристрій запису використовується предикат

```
writedevice (datafile).
```

Для запису в файл даних можна використовувати будь-які предикати типу **write** або **writef**, чи інші предикати, які реалізують цілі програми.

Для закриття файлу використовується предикат

```
closefile (datafile).
```

Для закритого файла операції зчитування чи запису недопустимі. Недопустимі також будь-які маніпуляції зі змістом файла. Можливо лише реалізувати операції з файлом загалом. Ось фрагмент програми, який пояснює вищесказане

```
openwrite(datafile, "WRITE.DAT"),
writedevice(datafile),
<любі правила або предикати запису в файл>,
<будь-які інші правила та предикати>,
closefile(datafile).
```


Якщо файл вже існує (відкритий), то з ним можна здійснювати цілий ряд операцій: можна зчитати весь зміст файла, вибірково зчитати дані, модифікувати їх, дописати дані в кінці файлу.

Для читання даних з файла необхідно відкрити файл для зчитування, призначити файл пристроєм читання, здійснити читання за допомогою відповідного предиката чи правила або будь-якого правила чи предиката, що відповідають цілям програми і закрити файл.

Ілюстрацією сказаному служить такий запис

```
openread(datafile, "FILE.DAT"),
readdevice(datafile),
<любі правила чи предикати читання з файлу>
<любі другі правила чи предикати програми>
closefile(datafile).
```

Для модифікації файла, який вже існує, необхідна процедура, яка відрізняється від запису чи читання з файла. Для цього існує предикат `openmodify`, який успішний тільки якщо такий файл існує. Іншими словами перед тим необхідно створити файл за допомогою предикату **openwrite**.

Необхідно пам'ятати, що **openwrite** створює новий файл навіть тоді, коли файл з таким іменем вже існує, затираючи в ньому все, що там знаходилося.

Вмістиме файла можна розглядати як потік символів. Кожен з них знаходиться у файлі на певній позиції. Позиція файла визначається віддаленістю від першого символу. Так, перший символ займає нульову позицію, другий – першу і т.д. Уявимо собі невидимий вказівник, який вказує на довільну позицію файла. При відкритті фала на модифікацію цей вказівник зміщується на початок файла. Модифікацію файла можна реалізувати послідовністю таких кроків: відкриття файла на модифікацію, переадресація пристрою виводу в файл, запис нових даних або використання правил, що відповідають цілям програми, закриття файла. Ця послідовність дій записується так

```
openmodify(datafile,"FILE.DAT"),
writedevice(datafile),
<правила для вибіркового запису в файл>
<будь-які правила чи предикати програми>
closefile(datafile).
```


Для дописування даних в кінець файла, який існує, використовується предикат openappend. При його виклику курсор файла зміщується в кінець файла. Далі реалізуються процедури переадресації пристрою виводу в файл, дописування даних, закриття файла за допомогою таких тверджень:

```
openappend(datafile,"FILE.DAT"),
writedevice(datafile),
< будь-які правила для дописування в файл>
< будь-які інші правила чи предикати програми>
closefile(datafile).
```

1.8.2. Програма запису даних у файл та виводу на екран

Як приклад розглянемо програму, яка бере вхідні дані з бази даних, а результат записує на екран і в файл на диску:

```
/* Вивід інформації з бази даних на екран дисплею
          і в файл на диску */
domains
 str = string
 file = f1
```

```
    predicates
     data(str)
     wrt_line
    goal
     openwrite(f, "F.DAT"),
     wrt_line, closefile(f1).
    clauses
     data("Програма читає дані з бази даних").
     data("Дані видаються на екран").
     data("Дані записуються в файл").
     wrt_line     :-     data(L),     write(L),     nl,
writedevice(f1),
         write(L), nl, writedevice(screen), fail.
    wrt_line.
    /* Кінець програми */
```

В програмі основним є правило **wrt_line,** яке використовує метод повернення (відкату) після неуспіху. Перший з предикатів цього правила присвоює змінній L об'єкт першого твердження **data.** Це значення друкується на заданий пристрій виводу за замовчуванням, тобто на екран дисплея. Наступні три предикати переадресовують вивід в файл, записують туди дані і знову переадресовують пристрій виводу на екран. Предикат **fail** викликає повернення до наступного предиката **data,** так, що будуть перебрані всі дані бази даних з предикатом **data.** Останній предикат **wrt_line** потрібен для того, щоб задовольнити ціль, коли перший варіант правила зазнав неуспіх внаслідок вичерпання бази даних.

1.8.3. Програма зчитування даних з файла та виводу на екран

Для процесу зчитування даних із вже існуючого файла необхідно відкрити файл для зчитування, вивести дані на екран чи принтер і закрити файл. Цей процес ілюструє така програма

```
    /* Читання даних з файла і вивід їх на екран */
     domains
     str = string
     file = f1
    predicates
     r_w_line
    goal
```

```
     openread(f, "f1.dat"),
          r_w_line, closefile(f1).
    clauses
    r_w_line    :-    readdevice(f1),    not(eof(f1)),
readln(L), writedevice(screen), write(L), nl, r_w_line.
     r_w_line.
    /* кінець програми */
```

Основне правило програми **r_w_line** використовує вбудований предикат **eof,** який має успіх, якщо виявлений кінець файла. Якщо в процесі зчитування з файла досягнутий кінець файла, то ніяких читань більше не відбудеться. Неуспішною буде будь-яка ціль, яка намагається виконати читання при вказівнику на кінець файла.

На природній мові використання **eof** з логічним запереченням на **f** можна інтерпретувати так: "Продовжувати читати і друкувати дот, доки не буде досягнутий кінець файла". Предикат **nl** при запису дає ознаку кінця рядка. Відповідно **readln** зчитує дані до цієї ознаки: комбінації символів **CR - LF** – повернення каретки і переведення рядка (їх код ASCII 13 і 10). Останні два предикати правила переадресовують вивід на екран і виводять туди дані. Якщо більше переадресацій виводу в програмі немає, то цей предикат **writedevice(screen)** не є обов'язковим. Хоча його включення в програму не є помилкою, більше того, явний опис виводу є ознакою надійного програмування.

1.8.4. Програма зчитування даних з клавіатури та запис у файл

Однією з найпоширеніших операцій при роботі з файлами є ввід даних з клавіатури з подальшим їх засиланням у файл на диск. Така програма реалізує процедуру запису даних у файл при їх зчитуванні з клавіатури:

```
    /* Зчитування даних з клавіатури і запис їх у файл
на диску */
    domains
     file = f1
     dstr, cstr = string
     predicates
     readln(dstr, cstr)
```

```
     create_a_file
    goal
     create_a_file.
     clauses
     create_a_file :-
    write("Введіть     ім'я     файла"),     readln(F),
openwrite(f1,F),
    writedevice(f1),     readln(Dstr),     concat(Dstr,
"\13\10", Cstr),
    readln(Dstr, Cstr), closefile(f1).
     readln("done", _) :- !.

    readln(_,Cstr) :-
    write(Cstr), readln(Dstr1), concat(Dstr1, "\13\10",
Cstr1),
    writedevice(f1), readln(Dstr1, Cstr1).
    /* кінець програми */
```

Ім'я файла, куди будуть записуватися дані, запитується з екрана дисплея. Для закінчення запису необхідно ввести слово **done** – зроблено, і натиснути **enter**. Створення файла на цьому закінчується. Далі предикат **readln(F)** правила **create_a_file** зчитує перший з введених рядків з клавіатури, предикат **concat** приєднує до неї комбінацію **CR-LF**, що потрібно при подальшому зчитуванні з файла. Результат **Cstr** записується в файл. Далі викликається правило **readln**, яке продовжує читання і запис. Перший його варіант успішний, якщо **Dstr** має значення **done**. Використання відсічки перериває можливість повернення для подальшого вводу даних.

Другий варіант правила використовує поширений спосіб випереджувального читання. Спочатку правило **readLn** вводить рядок, отриманий за допомогою **concat** в файл. Змінна **Dstr1** означується за допомогою нового рядка, введеного з клавіатури, який зчіплюється з комбінацією **CR-LF** і формується змінна **Cstr1**. Вивід переадресовується в файл на диск і цей рядок попаде в файл на початку нового кола рекурсії **readln**.

Кожен символ у файлі має певну позицію і вказівник файла може бути встановлений на будь-яку з них. Файли з такою формою доступу до даних

називаються файлами прямого доступу. Для роботи з ними використовують предикати **openmodify** і **filepos**. Форма запису **filepos** така:

filepos (Name, Pos, Mode)

де **Name** – логічне ім'я файла;

Pos – дійсне число, яке вказує позицію в файлі, з якої буде зчитаний чи записаний символ;

Mode – набуває значення 0,1,2, відповідно до якого відраховується зміщення від початку файла, поточної позиції або кінця файла.

Використовуючи цей предикат, можна записати дані в будь-яке місце файла або зчитати його з будь-якого місця.

1.8.5. Контрольні питання та вправи

1. Характеристики вбудованих предикатів для роботи з файлами.
2. Навести фрагмент програми для зчитування даних з клавіатури та запис їх у файл на диску.
3. Написати фрагмент програми для зчитування даних з бази даних Прологу, виведення їх на екран дисплея та запис у файл на диску.
4. На Псевдо-Пролозі побудувати предикат дозапису даних в кінець файла із заданим ім'ям.
5. Написати програму зчитування даних з екрана, записування їх у файл та виведення на друк.

Розділ 2. Побудова прикладних програм мовою логіки

2.1. Створення динамічної бази даних

2.1.1. Основні поняття баз даних

База даних (БД), записана на Турбо-Пролозі, подається набором символів і цифр (фактів і правил), записаних в синтаксично правильній формі. Ця упорядкована сукупність є вмістом бази даних.

Для оперативного доступу до даних в середовищі Турбо-Прологу передбачена можливість дописування, вилучення і коригування даних. Ці функції реалізують спеціальні предикати. Сукупність бази даних та відповідних програм управління ними утворюють систему керування базою даних (СКБД).

Існують три добре відомі моделі організації БД. Це ієрархічна модель, мережева модель і реляційна модель. В ієрархічній моделі дані зберігаються в окремих модулях, які мають встановлений пріоритет. У мережевій моделі дані містяться у вигляді зв'язаних модулів, що утворюють мережу. У реляційній БД дані зберігаються у вигляді окремих таблиць. В сучасних СКБД найчастіше використовують реляційні моделі. В Турбо-Пролозі такі моделі легко реалізуються.

2.1.2. Файл бази даних

Файл бази даних являє собою набір зв’язаних між собою записів. Файл БД має вигляд простого файла, але дані, які він містить, мають певну внутрішню організацію. Дані всередині кожного запису мають однакову структуру. Часто є один спеціальний запис, в якому подано спосіб організації записів. Ця таблиця (схема) відіграє роль “карти”, керуючись якою, СКБД працює з даними.

Файл – це сукупність записів. Своєю чергою, запис – це сукупність полів. Кожне поле містить частину даних. Як приклад розглянемо базу даних, яку створено на основі інформації про зарплатню працівників деякої компанії. Дані можуть бути організовані у вигляді файла записів, що показано в табл. 2.1.

Таблиця 2.1

Структура файла бази даних

	Поле імені	Поле відділу	Поле ставки
Запис 1	John Walker	ACCT	3.5
Запис 2	Tom Sellack	OPER	4.5
Запис 3	Jack Hunter	ADVE	4.5
Запис 4	Sam Ray	DATA	6.5

Кожен запис містить три поля, які мають імена **NAME (ім'я)**, **DEPARTMENT (відділ)** і **RATE (ставка**). Вміст **полів** першого запису – John Walker, ACCT і 3.5. Така структура доволі характерна для реляційних баз даних.

Базові операції роботи з файлами такі: додавання нових записів, модифікація раніше записаних і запити в БД.

Зазвичай всі записи файла БД мають однакову довжину, тому розміщення будь-якого запису в файлі визначити легко. В БД на Турбо-Пролозі записи в файлі не обов'язково однакової довжини, оскільки записи розташовані за шаблоном.

2.1.3. Реляційні бази даних

Дані в реляційних моделях подаються у вигляді таблиць, що складаються з рядків і стовпців (табл. 2.2).

Таблиця 2.2

Відношення бази даних

	Атрибут 1 (ім'я)	Атрибут 2 (команда)	Атрибут 3 (позиція)
Елемент відношення 1	Dan Marino	Miami Dophins	QB
Елемент відношення 2	Richard Dent	Chicago Bears	DE
Елемент відношення 3	Bernie Kosar	Cleveland Browns	QB
Елемент відношення 4	Doug Cosbie	Dallas Cowboys	TE
Елемент відношення 5	Mark Malone	Pittsburgh Steelers	QB

Ця таблиця складається з трьох стовпчиків і п’яти рядків. В ній кількість елементів відношення дорівнює 5, а кількість атрибутів – 3. Ім'я **"гравець"** є назвою усієї структури даних. Гравець є відношенням. Ім'я стовпчика задає ім'я атрибута, що зв'язаний у таблиці з іншими атрибутами. Для відношення "player" введені три атрибути. Набір атрибутів, у нас атрибути **Name (ім’я), Team (команда) і Position (позиція)**, називають реляційною схемою. Кількість стовпчиків (атрибутів) називають арністю відношення. Арність відношення "player" дорівнює 3.

Рядок таблиці в реляційній базі даних звичайно називають елементом відношення. У нашому прикладі першим елементом відношення є

Dan Marino, Miami Dolphins, QB

Кількість елементів відношення називають потужністю відношення. Потужність відношення "**player**" дорівнює 5.

Таблиця 2.3

Еквівалентне відношення бази даних в Турбо-Пролозі

Різні варіанти твердження	Об’єкт 1 (ім’я)	Об’єкт 2 (команда)	Об’єкт 3 (позиція)
Твердження 1	Dan Marino	Miami Dophins	QB
Твердження 2	Richard Dent	Chicago Bears	DE
Твердження 3	Bernie Kosar	Cleveland Browns	QB
Твердження 4	Doug Cosbie	Dallas Cowboys	TE
Твердження 5	Mark Malone	Pittsburgh Steelers	QB

У наведеній табл. 2.3 кількість різних тверджень дорівнює п’яти, а кількість об’єктів – трьом.

У Турбо-Пролозі існують спеціальні засоби для організації баз даних. Вони розраховані на роботу з реляційними базами даних, тому що Турбо-Пролог особливо зручний для написання діалогової системи саме для реляційної БД: внутрішні процедури мови здійснюють автоматичну вибірку фактів з потрібними значеннями відомих параметрів і присвоюють значення ще не визначеним. Такий механізм дає змогу знаходити всі наявні відповіді на зроблений запит.

Порівнюючи табл. 5 і 6, можна помітити схожість між стандартною реляційною базою даних і базою даних мовою Турбо-Пролог. Щоб зрозуміти, як у Турбо-Пролозі реалізується звертання до БД, розглянемо запит

player ("Bernie Kosar", Team,Pos).

У цьому твердженні **Team і Pos** є змінними, значення яких потрібно знайти. Процедури Турбо-Прологу переглядають твердження БД на предмет

порівняння з твердженням, що містить **Bernie Kosar**. Якщо в базі даних таке твердження наявне, тоді змінній Team присвоюється значення **Cleveland Browns**, а змінній **Pos - QB**.

Якщо трактувати **dplayer** як предикат БД Турбо-Прологу, то звідси випливає таке його описання

```
database
dplayer (name, team, position)
```

Розділ **database** у Турбо-Пролозі призначений для описання предикатів бази даних, таких, як **dplayer**. Усі різні твердження цього предиката утворюють динамічну базу даних Турбо-Прологу. БД називається динамічною, якщо з неї можна вилучати будь-які твердження, що містяться в ній, а також додавати нові. У цьому її відмінність від "статичних" баз даних, де твердження є частиною коду програми і не можуть бути змінені під час роботи. Інша важлива особливість динамічної бази даних полягає в тому, що така база може бути записана на диск, а також зчитана з диска в оперативну пам’ять.

Іноді необхідно мати частину інформації бази даних у вигляді тверджень статичної БД; ці дані заносять в динамічну БД відразу після активізації програми. (Для цієї мети використовують предикати **asserta** і **assertz**, що будуть розглянуті нижче). Загалом, предикати статичної БД мають інше ім'я, але ту саму форму подання даних, що і предикати динамічної. Предикат статичної БД, що відповідає предикатові **dplayer** динамічної бази даних, такий

```
predicates
player(name, team, position)
clauses
player("Dan Marino", "Miami Dolphins", "QB").
player("Richart Dent", "Chicago Bears", "DE").
player("Bernie Kosar", "Cleveland Browns", "QB").
player("Doug Cosbie", "Dallas Cowboy", "TE").
player("Mark Malone", "Pittsburgh Steelers", "QB").
```

Зауважимо, що уся відмінність предиката **dplayer** порівняно з **player** полягає лише в одній зайвій букві терма. Додавання латинської букви **d** – звичайний спосіб розрізняти предикати динамічної і статичної баз даних.

Правилом для занесення в динамічну БД інформації з тверджень предиката **player** є

```
assert_database :-
player(Name, Team, Number),
assertz(dplayer(Name, Team, Number)),fail.
assert_database :- !.
```

У цьому правилі застосовується метод, який після невдачі дає змогу перебрати усі твердження предиката **player**.

Порівнюючи цей приклад з відношенням бази даних в табл. 4, можна вказати таку відповідність:

База даних Турбо-Прологу	**Реляційна база даних**
предикат БД	**відношення**
об'єкт	**атрибут**
окреме твердження	**елемент відношення**
кількість тверджень	**потужність**

Порівняння дає змогу зрозуміти, що введена раніше термінологія Турбо-Прологу [10, 11] набуває нового значення, оскільки вона застосовується до реляційних БД.

2.1.4. Предикати динамічної бази даних у Турбо-Пролозі

У Турбо-Пролозі існують і спеціальні вбудовані предикати для роботи з динамічною базою даних. Такими є **asserta, assertz, retract, save, consult, readterm і findall**.

Предикати для роботи з твердженнями динамічної бази даних

Предикати **asserta, assertz і retract** дають змогу занести факт у задане місце динамічної БД і вилучити з неї вже наявний факт. Новий факт міститься перед усіма вже внесеними твердженнями цього предиката. Цей предикат має такий синтаксис:

```
asserta(Clause).
```

Отже, щоб помістити в БД твердження

```
dplayer("Bernie Kosar", "Cleveland Browns", "QB").
```

перед уже наявним там твердженням

```
dplayer("Doug Cosbie", "Dallas Cowboy", "TE").
```

(що міститься у цей момент у базі даних на першому місці), необхідний такий предикатний вираз

```
asserta(dplayer("Bernie Kosar", "Cleveland Browns",
"QB")).
```

Тепер БД містить два твердження, причому твердження з даними про Kosar передує твердженню з даними про Cosbie:

```
dplayer("Bernie Kosar", "Cleveland Browns", "QB").
dplayer("Doug Cosbie", "Dallas Cowboy", "TE").
```

Вкрай важливо усвідомлювати, що в динамічній базі даних можуть міститись тільки факти (не правила). У цьому полягає відмінність Турбо-Прологу від інших реалізацій мови Пролог.

Предикат **assertz** так само, як і **asserta**, може заносити нові твердження в базу даних. Однак він поміщає нове твердження після усіх вже наявних у базі тверджень того самого предиката. Синтаксис предиката простий:

```
assertz(Clause).
```

Для додавання до двох уже наявних у БД тверджень третього

```
dplayer("Mark Malone", "Pittsburgh Steelers",
"QB").
```

потрібно ввести такий предикатний вираз:

```
assertz(dplayer("Mark Malone", "Pittsburgh
Steelers", "QB")).
```

після цього БД буде містити три твердження

```
dplayer("Bernie Kosar", "Cleveland Browns", "QB").
dplayer("Doug Cosbie","Dallas Cowboy", "TE").
dplayer("Mark Malone", "Pittsburgh Steelers",
"QB").
```

Третє, нове, тільки що занесене твердження, міститься за двома старими.

Предикат **retract** знищує твердження з динамічної БД. Його синтаксис такий:

```
retract(Existing_clause).
```

Наприклад, ви хочете вилучити з бази даних друге твердження. Для цього необхідно написати вираз

```
retract(dplayer("Doug Cosbie", "Dallas Cowboy",
"TE")).
```

і БД буде складатися вже тільки з двох тверджень:

```
dplayer("Bernie Kosar", "Cleveland Browns", "QB").
dplayer("Mark Malone", "Pittsburgh Steelers",
"QB").
```

Так само, як **asserta** і **assertz**, **retract** застосовується тільки стосовно до фактів.

Для модифікації бази даних можна використовувати комбінацію виразів із предикатами **asserta, assertz і retract**. Наприклад, для того,

щоб відредагувати наявні в БД твердження, наша програма повинна одержати дані від користувача, скласти нове твердження, знищити попереднє і занести нове.

Предикати для роботи з базою даних загалом

Предикати **save** і **consult** застосовують для запису динамічної БД у файл на диск і для завантаження вмісту файла в динамічну БД.

Предикат **save** зберігає базу даних, що міститься в оперативній пам'яті, у текстовому файлі. Синтаксис цього предиката

```
save(DOS_file_name).
```

де **DOS_file_name** – це довільне допустиме в MS DOS або PC DOS ім'я файлу.

Для того, щоб зберегти вміст БД "Гра у футбол" у файлі з ім'ям **football.dba**, потрібен предикат

```
save("football.dba").
```

У результаті усі твердження, що містяться в оперативній пам'яті динамічної БД, будуть записані у файл football.dba. Якщо файл із таким ім'ям вже існує на диску, то він буде знищений.

Файл БД може бути записаний у пам'ять (завантажений) за допомогою предиката **consult**, синтаксис якого такий:

```
consult(DOS_file_name).
```

Для завантаження файла БД "Гра у футбол" потрібний вираз

```
consult("football.dba").
```

Предикат **consult** не можна використовувати, якщо файл із вказаним ім'ям відсутній на диску, або якщо цей файл містить помилки, як, наприклад, у разі невідповідності синтаксису предиката файлові, описаному у розділі програми **database** або вміст файла неможливо розмістити в пам'яті через відсутність місця.

Предикат **readterm** використовується для читання з файла об'єктів, що належать до визначеного в програмі домена. Синтаксис цього предиката

```
readterm(Domain, Term).,
```

де **Domain** задає ім'я домена, а **Term** – різні набори значень об'єктів цього домена. Розглянемо, наприклад, предикатний вираз

```
readterm(auto_record, auto(Name, Year, Price)).
```

У цьому виразі **Domain** замінений на **auto_record**, **Term** – на **auto(Name, Year, Price).** Терм **auto** визначає всі набори значень цього домена.

В цьому випадку опис доменів повинен виглядати так:

```
    domains
    name                    =       string
    year                    =       integer
    price                   =       real
    auto_record             =       auto(name, year,
price)
    file                    =       auto_file
```

Предикат **readterm** зазвичай використовують для зчитування з диска даних, записаних у формі тверджень. Так, у нас, наприклад, у файлі можуть міститися такі рядки:

```
auto("Pontiac", 1984, 8550)
auto("Chevrolet", 1982, 2300)
auto("Chevette", 1982, 1500)
auto("Toyota", 1986, 11000)
```

Для отримання доступу до файла спочатку необхідно скористатися предикатами **openread** і **readdevice**, після чого можна застосувати **readterm**.

У нашому прикладі предикат **readterm** намагається зіставити **auto** з відповідними твердженнями з файла. Якщо предикат успішний, змінні **Name**, **Year** і **Price** одержують значення з відповідного твердження. Ці значення можна вивести на екран відео або на будь-який інший пристрій.

2.2. Створення баз даних, що розташовані в оперативній пам'яті комп'ютера

Створення бази даних у Турбо-Пролозі починається з етапу проектування бази. Потрібно врахувати такі чинники:

1) розмір бази даних;

2) організацію елементів бази даних;

3) способи роботи і вмісту бази даних.

Використання баз даних, розміщених в оперативній пам'яті (резидентних), цілком виправдано, якщо ця БД має не занадто великий обсяг.

Для початку необхідно задати початкові дані і створити базу. Потім настає черга системи керування базою даних (СКБД), орієнтованої на діалог з користувачем. Будь-яка система такого роду містить як мінімум такі можливості:

1) занесення в базу нових даних ;

2) вилучення даних з бази ;

3) вибірка і виведення даних, що містяться в базі.

Ці вимоги передбачають наявність у системі меню, що дає змогу

користувачеві легко орієнтуватися, звертаючись до стандартних функцій СКБД, а також віконної системи, що дає чітке уявлення про доступні користувачеві засоби.

2.2.1. Проектування бази даних

Кращий спосіб розпочати проектування програми – це нарисувати її діаграму потоків даних, як це зроблено на рис. 2.1.

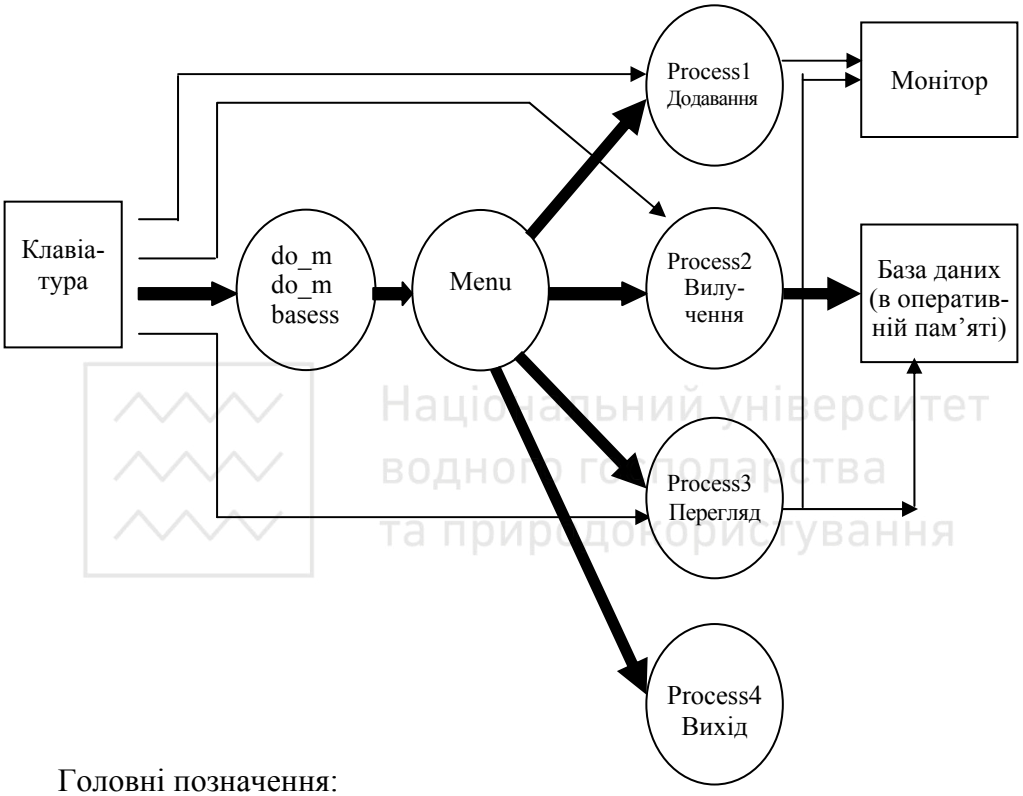


Рис. 2.1. Діаграма потоків даних резидентної БД

Стрілки на цій діаграмі показують потік даних і послідовність передавання керування у програмі. Стрілки одного типу позначають переміщення даних від одного модуля програми до іншого, а стрілки іншого типу – послідовність виклику цих модулів під час роботи у разі виконання деякого завдання.

На діаграмі з рис. 2.1 програмні модулі вибудувані в ієрархічну структуру. Керування потоком даних залежить від вибору команди

користувачем. Структурна схема, що відповідає цій ДПД, наведена на рис. 2.2. Вона показує, що модуль **menu** дає змогу користувачеві вибрати між чотирма модулями: **process(1)** для запису даних у базу, **process(2)** для вилучення даних, **process(3)** для виведення даних на монітор, і **process(4)** для виходу із системи.

Розглянемо операцію запису в базу нових даних. Користувач, що працює за клавіатурою, запускає програму. Для запису нових даних вибирають опцію **Add**. Керування в такий спосіб передається від основного модуля **do_mbase** до модуля **menu**, а потім до **process(1)**. Оскільки введення здійснюється з клавіатури, то саме від неї і бере початок потік даних; далі він йде до **process(1)** і закінчується на моніторі і базі даних.

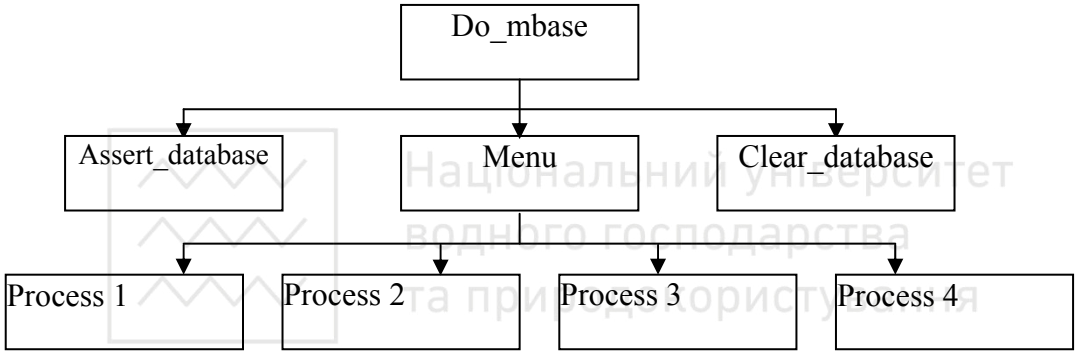


Рис. 2.2. Структурна схема резидентної БД

Щоб написати програми БД мовою Турбо-Пролог використовують дані, що є інформацією про п'ять професійних футболістів. База включає імена гравців, назви їхніх команд, номери, позиції, зріст, вагу, кількість зіграних у Національній футбольній лізі сезонів, а також назву університету, за який виступав цей гравець до переходу в професіонали. Вся інформація міститься в табл. 2.4.

Для роботи з нею необхідний предикат, що кодує цю інформацію.

```
player(p_name,    /* повне ім'я гравця (string) */
t_name,           /* назва команди (string) */
p_number,         /* номер гравця (integer) */
pos,              /* позиція гравця (string) */
height,           /* зріст (string) */
weight,           /* вага (string) */
nfl_exp,          /* стаж виступів (integer) */
college)          /* університет (string) */
```

Таблиця 2.4

Дані про гравців

Ім'я	Команда	N	Поз.	Зріст	Вага	Стаж	Університет
Dan Marino	Miami Dophins	13	QB	6'3"	215	4	Pittsburgh
Richard Dent	Chicago Bears	95	DE	6'5"	263	4	Tenn.State
Bernie Kosar	Cleveland Browns	19	QB	6'5"	210	2	Miami
Doug Cosbie	Dallas Cowboys	84	TE	6'6"	235	8	Santa Clara
Mark Malone	Pittsburgh Steelers	16	QB	6'4"	223	7	Arizona State

Об'єктам предиката присвоєні імена, що пояснюють суть справи, і тому запам'ятовуються легко. Об'єкт **p_name** кодує ім'я гравця, **t_name** – назву команди і т.д. Цей предикат є основою для подальшої побудови бази даних.

Турбо-Пролог вимагає, щоб усі твердження того самого предиката були згруповані в одному місці. Відповідно до цієї вимоги групу предиката **player** записують у вигляді:

```
player("Dan Marino","Miami Dolphins",13,"QB","6-3",
215,4,"Pittsburgh").
player("Richart Dent","Chicago Bears",95,"DE","6-
5",263,4,"Tennessee State").
player("Bernie Kosar","Cleveland Browns",19,"QB","6-
5",210,2,"Miami").
player("Doug Cosbie","Dallas Cowboys",84,"TE","6-
6",235,8,"Santa Clara").
player("Mark Malone","Pittsburgh Steelers", 16, "QB",
"6-4", 223, 7, "Arizona State").
```

Зауважимо, що якщо об'єкти тверджень є рядками і починаються з великих букв, то їх беруть в лапки. Також відзначимо, що зріст гравця задається у вигляді рядка. Хоча він і має числове значення, але у БД як число його не використовують.

Наступною фазою створення БД є задання відповідних описань типів. Розділ нашої програми **domains** буде виглядати так:

```
Domains
p_name, t_name, pos, height, college = string
p_number, weight, nfl_exp = integer
```

Звичайно, якщо створюється власна база даних, можуть знадобитися й об'єкти типу **symbol** або **real**. Їх необхідно також описати в розділі **domains**.

Предикати динамічної бази даних описують в розділі програми **database**. У нас у необхідний лише один такий предикат:

```
database
dplayer(p_name, t_name, p_number, pos,height,
weight, nfl_exp, college)
```

Коли програма запускається, твердження динамічної БД містяться в оперативній пам'яті окремо від "звичайних" тверджень. (Це одна з причин того, що предикати динамічної БД описують в спеціальному розділі програми). У цей момент БД повністю готова до роботи.

У розділі **predicates** варто описати всі інші предикати, використовуванні в програмі. Ці описання виглядають так:

```
predicates
repeat                  /* повтор */
do_mbase                /* ціль */
assert_database         /* створення БД */
menu                    /* інтерфейс у вигляді меню */
process(integer)        /* різні операції з переліку
меню */
clear_database          /* очищення БД */
player(p_name, t_name, p_number, pos,
height, weight, nfl_exp, college)
error                    /* видання повідомлення про
помилку */
```

На початку роботи програми необхідно занести в динамічну БД призначену для неї інформацію, що міститься в статичній БД. Це завдання виконує предикат **assert_database**. Предикат **clear_database** призначений для розв’язання суміжної задачі: очищення БД перед закінченням роботи програми. Насправді, звичайно, ця робота є зайвою, але ми включили цей предикат у нашу програму, тому що очищення БД потрібне в деяких додатках.

Завданням предиката **error** є реагування на введення неправильної вхідної інформації.

Предикат **player** призначений для задання початкового вмісту бази даних, тієї інформації, що є в табл. 4. Коли програма розпочинає роботу, ця інформація заноситься у твердження предиката **dplayer**.

Предикат **do_mbase** є головним правилом (модулем) програми. Він також наявний у цільовому твердженні. Предикат **menu** визначає правило, що здійснює інтерфейс із користувачем за допомогою меню. Предикат **process(integer)** визначає різні правила, що виконують усі можливі операції над БД.

Розділ програми **goal** містить правило **do_mbase**:

```
goal
do_mbase.
```

Тіла правил будуть визначені в розділі програми **clauses**.

2.2.2. Реалізація програмних модулів

Після закінчення етапу проектування можна розпочати стадію реалізації проекту. Тепер нашим завданням є написати основні і допоміжні правила, що будуть потрібні основним модулям.

Головний модуль

Головний модуль програми **do_mbase** є одночасно і метою програми:

```
do_mbase :-
assert_database,
makewindow(1,7,7," PRO FOOTBALL DATABASE
",0,0,25,80),
menu,
clear_database.
```

Модуль засилає в базу інформацію з тверджень **player**, створює вікно, висвічує меню й очищає БД після закінчення роботи програми.

Вхідні дані БД задають за допомогою тверджень з використанням статичних предикатів. Правило для занесення в базу цієї інформації таке

```
assert_database :-
player(P_name,T_name,P_number,Pos,Ht,Wt,Exp,College),
assertz(dplayer(P_name,T_name,P_number,Pos,Ht,Wt,Exp,
College)), fail.
assert_database :- !.
```

Предикат очищення бази даних – це

```
clear_database :-
retract( dplayer(_,_,_,_,_,_,_,_) ),
fail.
clear_database :- !.
```

Об'єкти тверджень **dplayer** у цьому правилі не становлять інтересу, тому використовуються анонімні змінні.

Меню призначено, щоб користувачеві було зручно вибирати програмні функції. Для забезпечення відповідності вимогам віконного простору створюється вікно на увесь екран (25 рядків, 80 стовпчиків). Модуль **menu** висвічує чотири доступні користувачеві опції. Як уже було сказано під час розгляду проекту програми, такими є

1. **Add a player to database** (занесення в БД нової інформації про гравців).

2. **Delete a player from database** (вилучення інформації про гравців із БД).

3. **View a player from database** (видавання інформації на екран).

4. **Quit from this program** (закінчення роботи з програмою).

Модуль **menu** переважно складається з предикатів **write**, котрі висвічують на екрані перераховані вище опції. Зірочки використовують тут для виділення простору, що містить меню. В модулі наявні предикати **write**, що створюють цей бордюр із зірочок, і предикат, що запитує в користувача ціле число в діапазоні від 1 до 4.

Модуль **menu** цілком відповідає поставленим до нього в проекті програми вимогам:

```
menu :-
repeat,
clearwindow,
write(" * * * * * * * * * * * * * * * * * * * *
"),nl,
write(" *                                     *
"),nl,
write(" * 1. Add a player to database         *
"),nl,
write(" * 2. Delete a player from database    *
```

```
"),nl,
    write(" * 3. View a player from database * "),nl,
    write(" * 4. Quit from this program           *
"),nl,
    write(" *                                     *
"),nl,
    write(" * * * * * * * * * * * * * * * * * * * *
"),nl,
    nl,
    write(" Please enter your choice, 1, 2, 3 or 4 :
"),
    readint(Choice),nl,
    process(Choice),
    Choice = 4,
    !.
```

Вигляд екрана з меню наведено на рис. 2.3.

PRO FOOTBALL DATABASE

```
* * * * * * * * * * * * * * * * * *
*1. Add a player to database          *
* 2. Delete a player from database    *
* 3.View a player from database
* 4. Quit from this program           *
*                                     *
* * * * * * * * * * * * * * * * * * * *
Please enter your choice, 1, 2 , 3 or 4:
```

Рис. 2.3. Меню програми “База даних гри у футбол”

Якщо користувач введе число, що дорівнює 4 (4 викликає закінчення програми), вибір Choice = 4 викликає повернення до предиката **repeat**.

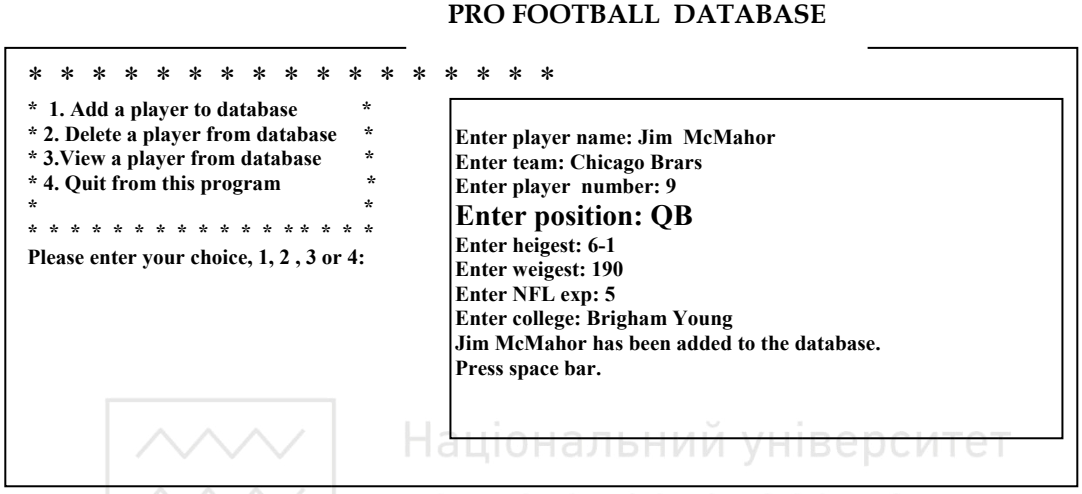
Розглянемо реалізацію правила **process**.

Модуль для введення даних

Правило **process(1)** призначено для занесення даних в БД. Цей модуль створює вікно для тексту, просить користувача ввести дані з клавіатури, зчитує їх і заносить в БД нове твердження **dplayer**. Після цього

модуль забирає створене вікно і повертає керування головному меню.

Окреме, менше за розміром, вікно створюється для забезпечення діалогу з програмою. За предикатом **makewindow**, що створює це вікно, йдуть предикати **write**, **readln** і **readint**, що інформують користувача про те, які дані він повинен увести, і зчитують ці дані з клавіатури:




```
    process(1) :-
    makewindow(2,7,7," Add Player to DATABASE
",2,20,18,58),
     shiftwindow(2),
     write("Enter player name: "),
     readln(P_name), write("Enter team: "),
readln(T_name),
```

Аналогічно за допомогою **write**, **readln** і **readint** вводять номер гравця, його позицію, зріст, вагу, стаж виступів і університет.

Об'єктами нового твердження є значення, присвоєне змінним **P_name**, **T_name**, **P_number** і т.п.; змінні визначаються в предикатах читання.

```
    assertz(dplayer(P_name,T_name,P_number,Pos,Ht,Wt,Ex
p,College)),
    write(P_name," has been added to the database."),
    nl, !,
    write("Press space bar. "),
    readchar(_),
    removewindow.
```

Останні рядки сигналізують про закінчення введення і забирають додаткове вікно.

Модуль для вилучення даних

Призначенням модуля **process(2)** є вилучення інформації з БД. Це правило, як і правило **process(1)**, створює власне вікно, запитує в користувача ім'я гравця і вилучає з БД твердження, що містить інформацію про нього. Після очищення вікна керування знову передається головному меню.

Після предикатів, що створюють вікно і рухають ним, йдуть предикати, що запитують ім'я гравця. Введене користувачем значення присвоюється змінній **P_name**:

```
    process(2) :-
    makewindow(3,7,7," Delete Player from DATABASE
",10,30,7,40),
    shiftwindow(3), write("Enter name to DELETE: "),
    readln(P_name),
```

Наступна частина правила здійснює операцію вилучення твердження з БД, надсилає коротке повідомлення про це користувачеві, чекає натискання ним довільної клавіші і забирає з екрана додаткове вікно.

```
    retract(dplayer(P_name,_,_,_,_,_,_,_)),
    write(P_name," has been deleted from the
database."),
    nl, !,
    write("Press space bar."),
    readchar(_),
    removewindow.
```

Модуль для вибірки даних

Призначенням модуля **process(3)** є пошук даних, що містяться в базі. Цей модуль, як і два попередні, створює власне вікно, а потім запитує ім'я гравця. Якщо в БД є твердження, що містить введене ім'я, модуль робить вибірку даних і виводить їх на екран у зручному форматі.

```
    process(3) :-
    makewindow(4,7,7," View Window ",7,30,16,47),
    shiftwindow(4),
    write("Enter name to view: "), readln(P_name),
    dplayer(P_name,T_name,P_number,Pos,Ht,Wt,Exp,Colleg
e).
```

Предикат **dplayer** шукає потрібне твердження в базі даних і вибирає рядкові і цілі значення з кожного запитуваного пункту. Ціла низка предикатів **write** потім виводить отримані значення:

```
nl, write("              NFL League Player"),nl,
nl, write(" Player Name :            ",P_name),
nl, write(" Team Name :              ",T_name),
nl, write(" Position :               ",Pos),
nl, write(" Player Number :          ",P_number),
nl, write(" Player's Height :  ",Ht," ft-in"),
nl, write(" Player's Weight :  ",Wt," lb "),
nl, write(" Player's NFL-exp : ",Exp," hyear(s)"),
nl, write(" Player's College : ",College),
nl, nl, !,nl, write("Press space bar"),
readchar(_),removewindow.
```

На рис. 2.5 наведено результати роботи цього модуля.

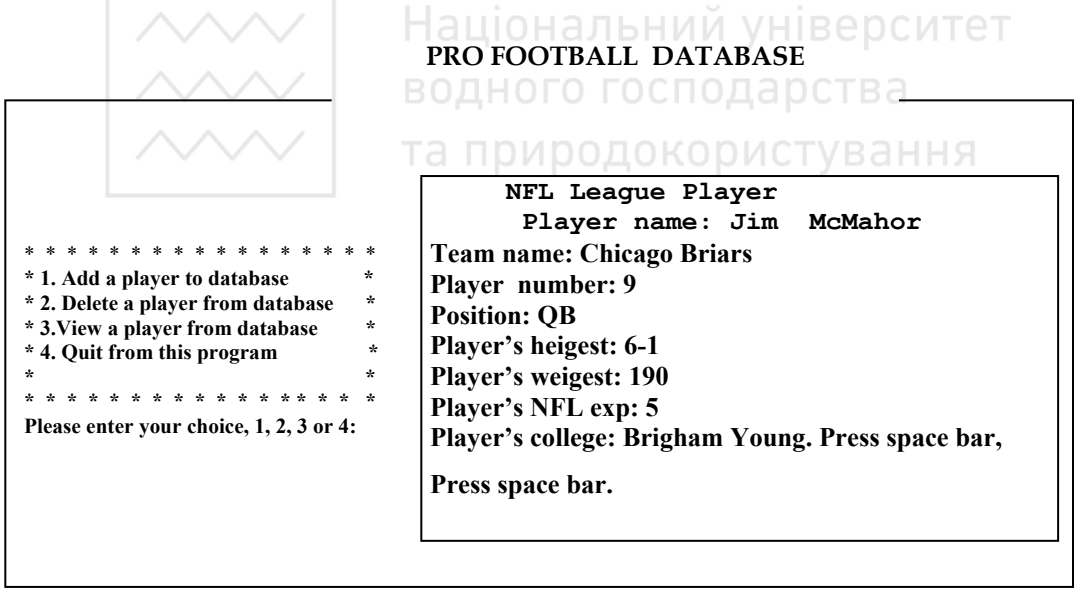


Рис. 2.5. Перегляд даних, що містяться в базі

Якщо в БД відсутнє твердження з введеним користувачем ім'ям гравця, програма видає повідомлення про помилку. Вікно для цього повідомлення повинно розташовуватися в центрі екрана. Варіант **process(3)**, відповідальний за видавання повідомлення про помилку, виглядає так:

```
    process(3) :-
    makewindow(5,7,7," No Luck ",14,7,5,60),
    shiftwindow(5),
    write("Can't find that player in the
database."),nl,
    write("Sorry, bye!"),
    nl, !,
    write("Press space bar."),
    readchar(_),
    removewindow,
    shiftwindow(1).
```

Модуль закінчення роботи з програмою

Модуль **process(4)** забезпечує нормальне закінчення сеансу роботи з базою даних. Цей модуль не створює власного вікна. Модуль вимагає від користувача чіткої відповіді на запитання, чи хоче він закінчити роботу з програмою:

```
    process(4) :-
    write("Are you sure want to quit (y/n)"),
    readln(Answer),
    frontchar(Answer,'y',_), !.
```

Предикат **frontchar** успішний тільки якщо відповідь користувача на запит програми починається з букви **Y**. Якщо вводиться інша буква, предикат неуспішний, тому відбувається повернення до предиката **repeat** модуля **menu**.

Модуль реакції на помилку

Програма повинна належно реагувати на допущені користувачем помилки під час введення. Якщо користувач введе число, менше за 1 або більше за 4, буде виконуватись одне з правил:

```
    process(Choice) :-
    Choice < 1, error.
    process(Choice) :- Choice > 4, error.
```

Ці правила викликають модуль **error**:

```
    error :-
    write("Please enter a number from 1 to 4."),
    write("(Press the space bar to continue)"),
    readchar(_).
```

2.2.3. Приклад бази даних гри у футбол

Реалізацією цього проекту є програма “База даних гри у футбол” (лістинг 2.1).

Лістинг 2.1

```
/* Програма:   База даних гри у футбол            */
/*              Файл: PROG0901.PRO                */
/*                                                */
/* Призначення:  Приклад бази даних, яка працює   */
/* База даних містить такі операції: додавання,   */
/*               вилучення і вибірку даних.       */
/*           Вибірка передбачає перегляд даних.   */
/* Зауваження:   Ця програма створює базу даних і */
/*              поміщає її в оперативну пам'ять.  */
domains
p_name, t_name, pos, height, college = string
p_number, weight, nfl_exp = integer
database
dplayer(p_name, t_name, p_number, pos,
height, weight, nfl_exp, college)
predicates
repeat
do_mbase
assert_database
menu
process(integer)
clear_database
player(p_name, t_name, p_number, pos,
height, weight, nfl_exp, college)
error
goal
do_mbase.
clauses
repeat.
repeat :- repeat.
/* База даних гри у футбол */
```

```
player("Dan Marino","Miami Dolphins",13,"QB","6-
3",215,4, "Pittsburgh").
player("Richart Dent","Chicago Bears",95,"DE","6-
5",263,4, "Tennessee State").
player("Bernie Kosar","Cleveland Browns",19,"QB","6-
5",210,2, "Miami").
player("Doug Cosbie","Dallas Cowboys",84,"TE","6-6",235,8,
"Santa Clara").
player("Mark Malone","Pittsburgh Steelers",16,"QB","6-
4",223,7,
"Arizona State").
/* кінець початкових даних */
assert_database :-
player(P_name,T_name,P_number,Pos,Ht,Wt,Exp,College),
assertz(
dplayer(P_name,T_name,P_number,Pos,Ht,Wt,Exp,College) ),
fail.
assert_database :- !.
clear_database :-
retract( dplayer(_,_,_,_,_,_,_,_) ),
fail.
clear_database :- !.
/* Діалог з цією БД здійснюється за принципом меню.
```

Ґрунтуючись на запиті користувача, СКБД активізує відповідні процеси для реалізаційного запиту. Меню можна розширити за рахунок введення нових функцій.*/

```
do_mbase :-
assert_database,
makewindow(1,7,7," PRO FOOTBALL DATABASE ",0,0,25,80),
menu,
clear_database.
menu :-
repeat,
clearwindow,
write(" * * * * * * * * * * * * * * * * * * * * "),nl,
write(" *                                       *
"),nl,
```

```
write(" * 1. Add a player to database              *
"),nl,
write(" * 2. Delete a player from database  * "),nl,
write(" * 3. View a player from database           *
"),nl,
write(" * 4. Quit from this program                *
"),nl,
write(" *                                          *
"),nl,
write(" * * * * * * * * * * * * * * * * * * * * "),nl,
nl,
write(" Please enter your choice, 1, 2, 3 or 4 : "),
readint(Choice),nl,
process(Choice),
Choice = 4,
!.
/* Додавання інформації про гравця в БД */
process(1) :-
makewindow(2,7,7," Add Player to DATABASE
",2,20,18,58),
shiftwindow(2),
write("Enter player name: "), readln(P_name),
write("Enter team: "), readln(T_name),
write("Enter player number: "), readint(P_number),
write("Enter position: "), readln(Pos),
write("Enter height: "), readln(Ht),
write("Enter weight: "), readint(Wt),
write("Enter NFL exp: "), readint(Exp),
write("Enter college: "), readln(College),
assertz(dplayer(P_name,T_name,P_number,Pos,Ht,Wt,
Exp,College)),
write(P_name," has been added to the database."),
nl, !,
write("Press space bar. "),
readchar(_),
removewindow.
/* Вилучення інформації про гравця з БД */
```

```
process(2) :-
makewindow(3,7,7," Delete Player from DATABASE
",10,30,7,40),
shiftwindow(3),
write("Enter name to DELETE: "), readln(P_name),
retract(dplayer(P_name,_,_,_,_,_,_,_)),
write(P_name," has been deleted from the database."),
nl, !,
write("Press space bar."),
readchar(_),
removewindow.
/* Перегляд інформації про гравця */
process(3) :-
makewindow(4,7,7," View Window ",7,30,16,47),
shiftwindow(4),
write("Enter name to view: "), readln(P_name),
dplayer(P_name,T_name,P_number,Pos,Ht,Wt,Exp,College),
nl, write("          NFL League Player"),nl,
nl, write(" Player Name :                ",P_name),
nl, write(" Team Name :                  ",T_name),
nl, write(" Position :                   ",Pos),
nl, write(" Player Number :              ",P_number),
nl, write(" Player's Height :  ",Ht," ft-in"),
nl, write(" Player's Weight :  ",Wt," lb "),
nl, write("Player's NFL-exp : ",Exp," year(s)"),nl,
write(" Player's College : ",College),
nl, nl, !,
nl, write("Press space bar"),
readchar(_),
removewindow.
process(3) :-
makewindow(5,7,7," No Luck ",14,7,5,60),
shiftwindow(5),
write("Can't find that player in the database."),nl,
write("Sorry, bye!"),
nl, !,
write("Press space bar."),
```

```
readchar(_),
removewindow,
shiftwindow(1).
/* Вихід з діалогу */
process(4) :-
write("Are you sure want to quit (y/n)"),
readln(Answer),
frontchar(Answer,'y',_), !.
/* Неправильне звертання до БД */
process(Choice) :-
Choice < 1, error.
process(Choice) :- Choice > 4, error.
error :-
write("Please enter a number from 1 to 4."),
write("(Press the space bar to continue)"),
readchar(_).
/*****                    кінець програми
                          *****/
```


Вхідна інформація для БД міститься на початку розділу **clauses**. Коли програма запускається, **assert_database** створює твердження **dplayer**, що містять такі самі дані, що і твердження статичного предиката **player**, і заносить ці твердження в динамічну БД. Після цього можна додавати, вилучати або переглядати дані, вибираючи відповідні опції меню.

2.3. Створення бази даних на зовнішніх носіях інформації

Слабким місцем усіх розглянутих БД є те, що в міру нагромадження в ній нових даних обмежувальним чинником стає обсяг оперативної пам'яті комп'ютера. Це обмеження, ймовірно, буде доволі істотним у будь-якому прикладному додатку БД.

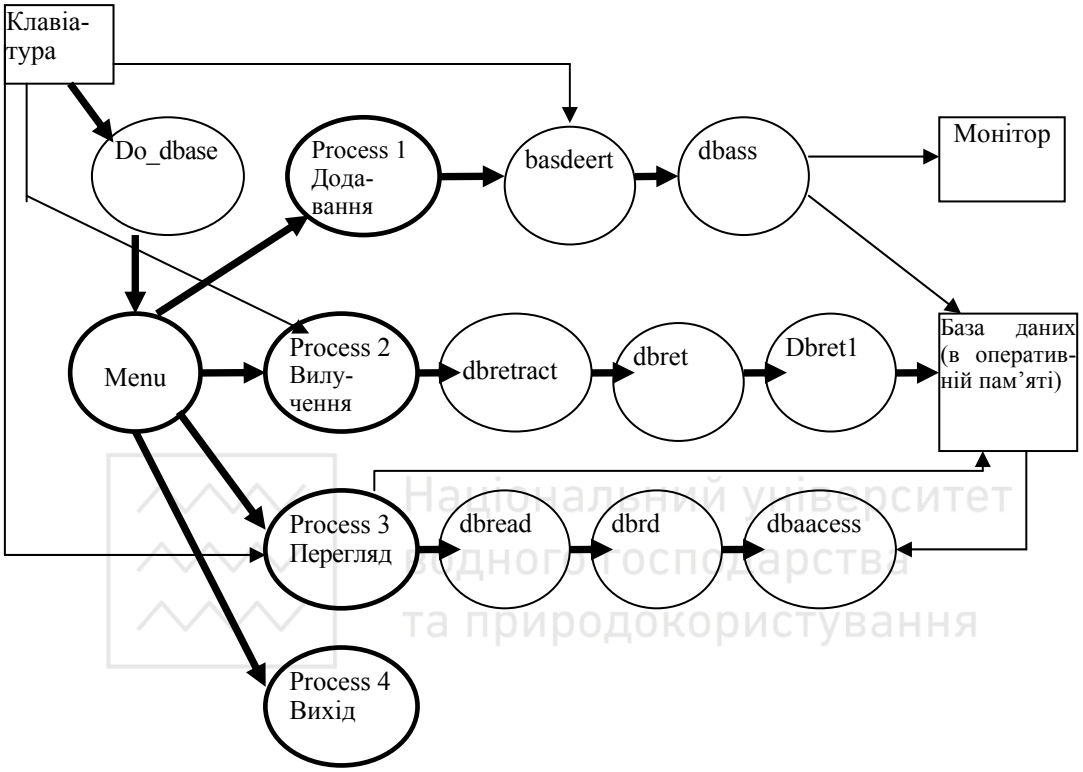
Життєздатнішим є СКБД, що зберігають дані на диску (нерезиденті), оскільки обсяг зовнішньої пам'яті істотно більший за обсяг оперативної пам'яті. Системи такого класу придатні для більшості практичних задач.

2.3.1. Розгляд проекту

Створення будь-якої програми на Турбо-Пролозі, що працює з БД починається з розгляду проекту бази даних.

Подібні міркування доречні і для БД на диску. Новими тут є правила введення і виведення інформації, названі введенням – виведенням на диск.

Важливою особливістю програми є забезпечення ефективного доступу до бази. Ця вимога змушує використовувати індексні файли.



Умовні позначення:

– потік даних

– передача керування

Рис. 2.6. Діаграма потоків нерезидентної БД

Розпочнемо проектування з діаграми потоків даних (рис. 2.6).

Структура цієї діаграми є розширенням ДПД для бази даних, що міститься в оперативній пам'яті (рис. 2.5). Якщо порівняти ці дві діаграми, неважко побачити, що у нову входить додатково кілька модулів між **process** і БД. Це допоміжні модулі, що викликаються з модулями **process**. Так, наприклад, **process(1)** викликає модуль **dbassert**, що у своєю чергою, викликає модуль **dbass**. Тонкі стрілки на діаграмі показують напрямок потоку даних, а товсті – послідовність передачі керування.

Структурна схема програми наведена на рис. 2.7. Вона дуже схожа на

структурну схему попередньої програми.

Допоміжні модулі – **dbassert** і **dbass** модуля **process(1)** заносять у базу нові дані. Аналогічні структури модулів використовують і для вилучення, вибірки і виведення даних на екран.

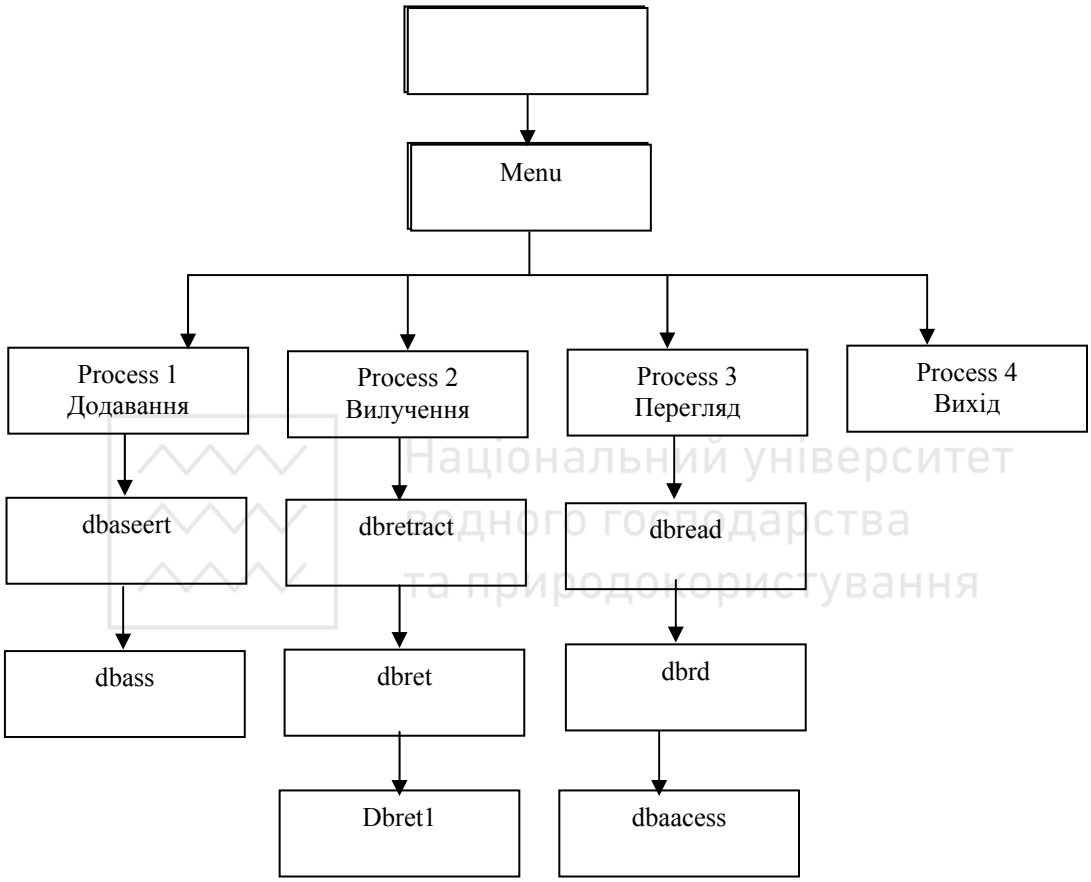


Рис. 2.7. Структурна схема нерезидентної БД

2.3.2. Створення бази даних

База даних "Університетський футбол" містить інформацію про футбольні команди університетів. Характер інформації і відповідні їй об'єкти Турбо-Прологу наведені в табл. 2.5.

Кожна з вибраних характеристик може вносити якусь нову рису в уявлення про команду. Проектуючи БД, варто уникати включення у відношення зайвої інформації. Іншими словами, якщо який-небудь параметр має те самі значення для всіх елементів відношення, то його не варто

зберігати в базі. Так, у нашу БД немає необхідності включати об'єкт "вид спорту". Йдеться про футбольні команди, і для кожного твердження бази цей об'єкт мав би значення **football**. Вилучення з БД інформації, загальної для усіх відношень, називають нормалізацією. Нормалізація дає змогу заощаджувати пам'ять і полегшує роботу з правилами, що здійснюють доступ до бази.

Таблиця 2.5

Інформація, що зберігається в БД "Університетський футбол"

Інформація	Об’єкт Турбо-Прологу	Тип даних
Назва команди	name	String
Назва університету	school	string
Місто	city	string
Штат	state	string
Кольори команди	color	string
Стадіон	stadium	string
Ім'я тренера	coach	string
Рік, за який наведені дані	year	integer
Рейтинг	team_power	integer
Рейтинг нападу	offensive_power	integer
Рейтинг захисту	defensive_power	integer
Рейтинг волі до перемоги	winning_power	integer
Рейтинг свого поля	home_power	integer

Дані в нашому прикладі вимагають введення предиката такої структури:

```
team(name, school, city, state, color, stadium,
coach, year, team_power, offensive_power,
defensive_power, winning_power, home_power)
```

Предикат з'являється в розділі програми **database**. Необхідно також описати (у розділі **domains**) об'єкти цього предиката. У результаті розділ **domains** набуває такого вигляду:

```
domains
file = datafile ;
indexfile
name, school, city, state, color,
stadium, coach  = string
```

```
    year,team_power, offensive_power,
    defensive_power, winning_power, home_power =
integer
    З попередньої програми  можна залишити предикати:
    do_dbase   /* ціль */
    menu            /* інтерфейс у виді меню */
    process         /* різні операції з переліку меню */
```

Додатково до них треба ввести допоміжні модулі, відповідальні за операції введення–виведення: додавання, вилучення і виведення даних. Описання цих предикатів закінчує розробку першої частини програми БД на зовнішньому носії. Повний текст розділу **predicates** наведений нижче:

```
    predicates
    repeat
    menu
    process(integer)
    do_dbase
    dbassert(dbasedom)   /* додавання даних */
    dbass(dbasedom, string, string)
    dbretract(dbasedom)   /* видалення  даних  */
    dbret(dbasedom, string, string)
    dbret1(dbasedom, real)
    dbread(dbasedom)          /* читання даних */
    dbrd(dbasedom, string, string)
    dbaaccess(dbasedom, real)
```

Модуль **do_dbase** є одночасно метою цієї програми. Тепер необхідно описати введені правила в розділі **clauses**.

Так само, як і попередня програма, **do_dbase** викликає модуль **menu**. Своєю чергою **menu** викликає один з модулів **process**; який саме з них викликають, залежить від значення введеного користувачем числа. Структура модулів **do_dbase** і **menu** залишається такою самою, якою вона була в програмі "База даних гри у футбол". Модулі **process** схожі на однойменні модулі тієї самої програми, з деякими відмінностями:

1) **process(1)** викликає допоміжний модуль **dbassert**, призначений для засилання даних у базу на диск;

2) **process(2)** викликає допоміжний модуль **dbretract**, що вилучає дані з бази на диску;

3) **process(3)** викликає допоміжний модуль **dbread**, що здійснює вибірку даних для видання їх на екран.

Ці допоміжні модулі мають такий опис:

```
dbassert(Term) :-
dbass(Term,"cfootball.ind","cfootball.dba").
dbretract(Term) :-
dbret(Term,"cfootball.ind","cfootball.dba").
dbread(Term) :-
dbrd(Term,"cfootball.ind","cfootball.dba").
```

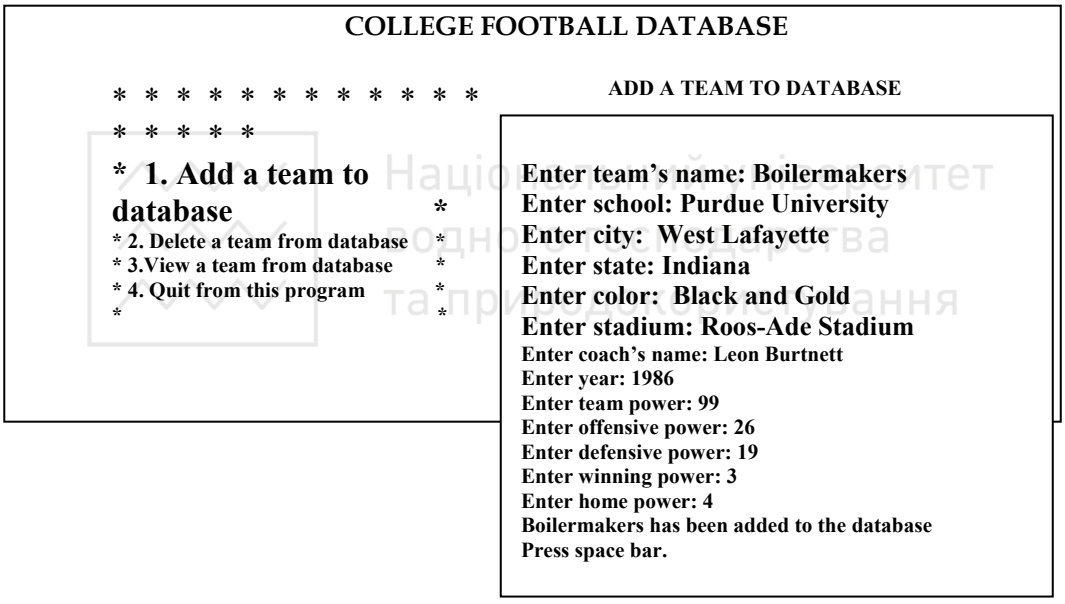


Рис. 2.8. Модуль для введення даних про футбольну команду

```
dbass(Term,Indexfile,Datafile) :-
existfile(Indexfile),
existfile(Datafile),
openappend(datafile,Datafile),
writedevice(datafile),
filepos(datafile,Pos,0),
write(Term), nl,
closefile(datafile),
openappend(indexfile,Indexfile),
```

```
    writedevice(indexfile),
    writef("%7.0\n",Pos),
    closefile(indexfile).

    dbass(Term,Indexfile,Datafile) :-
    openwrite(datafile,Datafile),
    writedevice(datafile),
    filepos(datafile,Pos,0),
    write(Term), nl,
    closefile(datafile),
    openwrite(indexfile,Indexfile),
    writedevice(indexfile),
    writef("%7.0\n",Pos),
    closefile(indexfile).
```

Зверніть увагу на використання в правилі предиката

```
writef("%7.0\n",Pos),
```

який призначений для запису значення індексу, що задається змінній **Pos**. Для запису індексу виділяється поле, що складається із семи позицій. Значення, присвоєне змінній **Pos**, визначає розташування запису у файлі БД.

Перший варіант правила **dbass** призначений для запису даних у вже наявний файл. Обидва варіанти правила, власне кажучи, ідентичні, відмінність полягає лише в тому, що перший варіант використовує предикат **existfile**, а також **openappend** замість **openwrite**.

На рис. 2.8 наведено стан екрана після того, як користувач увів дані про одну з університетських команд. Функційні призначення згаданих модулів наведено нижче.

Модуль для вилучення даних

Модуль **dbretract** вилучає дані з БД. Він викликає модуль **dbret**, що робить можливою операцію вилучення після відкриття файла БД і індексного файла.

```
    dbret(Term,Indexfile,Datafile) :-
    openread(datafile,Datafile),
    openmodify(indexfile,Indexfile),
    dbret1(Term,-1),
    closefile(datafile),
    closefile(indexfile).
```

У правилі використовується допоміжний модуль **dbret1**, що здійснює пошук потрібного запису і його вилучення:

```
    dbret1(Term,Datpos) :-
    Datpos >= 0, filepos(datafile,Datpos,0),
readdevice(datafile), readterm(Dbasedom,Term), !,
filepos(indexfile,-9,1), flush(indexfile),
writedevice(indexfile), writef("%7.0\n",-1),
readdevice(keyboard), writedevice(screen).
```

Предикат **flush** викликає запис на диск вмісту внутрішнього буфера індексного файла. У такий спосіб **dbret1** не допускає роботи з даними, що вже були вилучені до цього.

Пошук потрібного індексу в індексному файлі здійснює такий варіант **dbret1:**

```
    dbret1(Term,_) :-
    readdevice(indexfile),
    readreal(Datpos1),
    dbret1(Term,Datpos1).
```


Модуль для вибірки даних

Призначенням модуля **dbrd** є пошук і читання даних, що містяться в БД. Модуль оформлений у такий спосіб:

```
    dbrd(Term,Indexfile,Datafile) :-
    openread(datafile,Datafile),
    openread(indexfile,Indexfile),
    dbaaccess(Term,-1),
    closefile(datafile),
    closefile(indexfile).
```

Як видно з наведеного тексту, тут використовується допоміжний модуль **dbaaccess**, що здійснює пошук і вибірку даних з файла БД. Варіант правила, що працює з файлом БД, задається таким виразом:

```
    dbaaccess(Term,Datpos) :-
    Datpos >= 0, filepos(datafile,Datpos,0),
readdevice(datafile),
    readterm(dbasedom,Term).
```

Цей предикат зчитує дані, логічно зв'язані зі значенням індексу, що надається змінній **Datapos**. Відповідне значення індексу шукається в індексному файлі іншим варіантом **dbaaccess:**

```
dbaaccess(Term,_) :-
readdevice(indexfile),
readreal(Datpos1),
dbaaccess(Term,Datpos1).
```

Це правило намагається знайти в базі такий запис, індекс якого є наявним в індексному файлі. Якщо індекс знайдено, то правило успішне; якщо ні, то невдале. У разі успіху змінна Term набуває потрібних користувачеві значень.

Модуль виведення даних у разі запиту про університетську футбольну команду наведено на рис. 2.9. Дані виводяться в зручному для читання форматі.

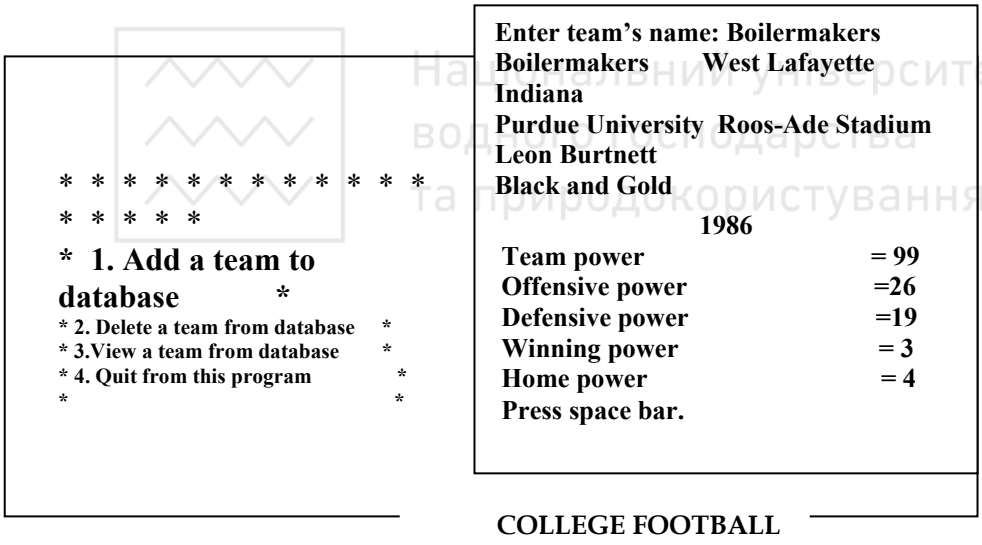


Рис. 2.9. Модуль для виведення даних про команду

2.3.3. Приклад бази даних “Університетський футбол”

Програма "Університетський футбол" (лістинг 2.2) реалізує запропонований у попередніх розділах проект бази даних.

Лістинг 2.2

```
/* Програма:        Університетський футбол.    */
/*                  Файл: PROG0902.PRO          */
/* Призначення: Приклад бази даних, яка працює */
/*і містить відомості про футбольну команду    */
/*    База даних допускає такі операції:        */
/*    додавання, вилучення, і вибірку даних.    */
/*     Вибірка передбачає перегляд даних        */
/* Зауваження:Ця програма створює базу даних і */
/*       розміщає на диску індексний файл.      */
/*       Ім'я цього файла - CFOOTBALL.DBA,      */
/*     ім'я зв'язаного індексного файла   -     */
/*    CFOOTBALL.IND. Файли розташовуються в     */
/*       поточній директорії.                   */
domains
file = datafile ;
indexfile
name, school, city, state, color,
stadium, coach  = string
year,team_power, offensive_power,
defensive_power, winning_power, home_power =
integer
database
team(name, school, city, state, color, stadium,
coach, year, team_power,
offensive_power,defensive_power, winning_power,
home_power)
predicates
repeat
menu
process(integer)
do_dbase
```

```
    dbassert(dbasedom)
    dbass(dbasedom,string,string)
    dbretract(dbasedom)
    dbret(dbasedom,string,string)
    dbret1(dbasedom,real)
    dbread(dbasedom)
    dbrd(dbasedom,string,string)
    dbaaccess(dbasedom,real)
    goal
    do_dbase.
    clauses
    /* Діалог з цієї БД здійснюється за  принципом
меню.
    Для цього використовують віконні засоби Турбо-
Прологу. Ґрунтуючись на запиті користувача, СКБД
активізує відповідні процеси для задоволення цього
запиту. */
    /* задання мети у вигляді правила */
    do_dbase :-
    makewindow(1,7,7," COLLEGE FOOTBALL DATABASE ",
0,0,24,80),
    menu.
    menu :-
    repeat,
    clearwindow,
    nl,
    write(" * * * * * * * * * * * * * * * * * * * *"),nl,
    write(" *                                     *
"),nl,
    write(" * 1. Add a player to database         *
"),nl,
    write(" * 2. Delete a player from database    *
"),nl,
    write(" * 3. View a player from database      *
"),nl,
    write(" * 4. Quit from this program           *
"),nl,
```

```
    write(" *                                                *
"),nl,
    write(" * * * * * * * * * * * * * * * * * * * * * * * *
"),nl, nl,
    write(" Please enter your choice, 1, 2, 3 or 4 :
"),readint(Choice),nl,
    Choice > 0, Choice < 5, process(Choice),
    Choice = 4,!.
    /* Додавання інформації про команду в БД */
    process(1) :-
    makewindow(2,7,7," Add a Team to DATABASE ",
    2,20,18,58), shiftwindow(2), write("Enter team's
nickname: "), readln(Name), write("Enter school: "),
readln(School), write("Enter city: "), readln(City),
write("Enter state: "), readln(State), write("Enter
color: "), readln(Color), write("Enter stadium: "),
readln(Stadium), write("Enter coach's name: "),
readln(Coach), write("Enter year: "), readint(Year),
write("Enter team power: "), readint(TMP), write("Enter
offensive power: "), readint(OFP), write("Enter
defensive power: "), readint(DEP), write("Enter winning
power: "), readint(WNP), write("Enter home power: "),
readint(HMP),nl,
dbassert(team(Name,School,City,State,Color,Stadium,
    Coach,Year,TMP,OFP,DEP,WNP,HMP)), write(Name," has
been added to the database."), nl, !,
    write("Press space bar. "),
    readchar(_),
    removewindow,
    shiftwindow(1).
    /* Вилучення інформації про команду з БД */
    process(2) :-
    makewindow(3,7,7," Delete a Team from DATABASE ",
    10,30,7,40), shiftwindow(3), write("Enter name to
DELETE: "), readln(Name),
dbretract(team(Name,_,_,_,_,_,_,_,_,_,_,_,_)),
```

```
write(Name," has been deleted from the database."), nl,
!,
    write("Press space bar."),
    readchar(_),
    removewindow,
    shiftwindow(1).
    /* Перегляд інформації про команду */
    process(3) :-
    makewindow(4,7,7," View a Team ",
    6,18,17,58), shiftwindow(4), write("Enter the
team's nickname: "), readln(Name),
dbread(team(Name,School,City,State,Color,Stadium,
    Coach,Year,TMP,OFP,DEP,WNP,HMP)),nl, write("
",Name),nl,
    write("   ",School,"   ",City,"   ",State),nl,
    write("   ",Color,"   ",Stadium,"   ",Coach),nl,nl,
    write("                    ",Year),nl,nl,
    write("              Team Power      = ",TMP),nl,
    write("              Offensive Power  = ",OFP),nl,
    write("        Defensive Power      =     ",DEP),nl,
    write("        Winning Power        =     ",WNP),nl,
    write("        Home Power           =     ",HMP),nl,
!,
    write("Press space bar"),
    readchar(_),
    removewindow,
    shiftwindow(1).
    process(3) :-
    makewindow(5,7,7," Message Window ",14,7,5,50),
    shiftwindow(5),
    write("Can't find that team in the database."),nl,
    write("Sorry, bye!"),
    closefile(datafile),
    closefile(indexfile), nl, !,
    write("Press space bar."),
    readchar(_),
    removewindow,
```

```
    shiftwindow(1).
    /* Вихід з діалогу */
    process(4) :-
    write("You are now exiting the 'College"),
    write(" Football Database' program."),nl,
           write(" Please press the space bar."),
    readchar(_),
    exit.
    repeat. repeat :- repeat.
    /* Правила для роботи з БД */
    dbassert(Term) :-
    dbass(Term,"cfootball.ind","cfootball.dba").
    dbretract(Term) :-
    dbret(Term,"cfootball.ind","cfootball.dba").
    dbread(Term) :-
    dbrd(Term,"cfootball.ind","cfootball.dba").
    /* Правило dbass записує інформацію у файл datafile
    і модифікує файл indexfile                          *
/
    dbass(Term,Indexfile,Datafile) :-
    existfile(Indexfile),
    existfile(Datafile),
              openappend(datafile,Datafile),
    writedevice(datafile),
    filepos(datafile,Pos,0),
    write(Term), nl,
    closefile(datafile),
    openappend(indexfile,Indexfile),
    writedevice(indexfile),
    writef("%7.0\n",Pos),
    closefile(indexfile).
    dbass(Term,Indexfile,Datafile) :-
               openwrite(datafile,Datafile),
    writedevice(datafile),
    filepos(datafile,Pos,0),
    write(Term), nl,
```

```
    closefile(datafile),
              openwrite(indexfile,Indexfile),
    writedevice(indexfile),
    writef("%7.0\n",Pos),
    closefile(indexfile).
           /* Правило dbret вилучає дані з БД */
    dbret(Term,Indexfile,Datafile) :-
    openread(datafile,Datafile),
    openmodify(indexfile,Indexfile),
    dbret1(Term,-1),
    closefile(datafile),
    closefile(indexfile).
    dbret1(Term,Datpos) :-
    Datpos >= 0, filepos(datafile,Datpos,0),
readdevice(datafile), readterm(Dbasedom,Term), !,
filepos(indexfile,-9,1), flush(indexfile),
writedevice(indexfile), writef("%7.0\n",-1),
readdevice(keyboard), writedevice(screen).
    dbret1(Term,_) :- readdevice(indexfile),
readreal(Datpos1), dbret1(Term,Datpos1).
    /*   Правило dbrd витягає інформацію з файлу
    datafile
                                                    *
/
    dbrd(Term,Indexfile,Datafile) :-
    openread(datafile,Datafile),
    openread(indexfile,Indexfile),
    dbaaccess(Term,-1),
    closefile(datafile),
    closefile(indexfile).
    dbaaccess(Term,Datpos) :-
    Datpos >= 0, filepos(datafile,Datpos,0),
readdevice(datafile), readterm(dbasedom,Term).
    dbaaccess(Term,_) :- readdevice(indexfile),
readreal(Datpos1), dbaaccess(Term,Datpos1).
    /*****                  кінець програми
                             *****/
```

В програмі містяться коментарі, які інформують про те, що вона робить. Одночасно пояснюється призначення різних програмних модулів. Наведена програма, на відміну від програми " База даних гра у футбол", спочатку створює порожню БД. Для того, щоб у БД з'явилася яка-небудь інформація, необхідно вибрати опцію **`Add a team to database`** головного меню для запису на диск нових даних. Після введення даних можна задати опцію **`View a team from the database`** і переглянути дані на екрані дисплея. Таким способом у БД можна записати стільки даних, скільки є вільного місця на диску. Так, можна ввести в базу назви і показники кращих футбольних команд. "Пожвавлення" БД реальною інформацією звичайно робить роботу з нею цікавішою.

2.3.4. Контрольні питання та вправи

1. Класифікація баз даних мовою логіки.
2. Означення реляційної бази даних.
3. Подібні та відмінні риси основних понять про бази даних в СУБД та Турбо-Пролозі.
4. Вбудовані предикати Турбо-Прологу для побудови динамічних баз даних.
5. Описати процедуру створення бази даних в Турбо-Пролозі на зовнішніх носіях.
6. Написати фрагмент програми для визначення столиць країн Європи за наявності в базі даних столиць світу.

2.4. Побудова експертних систем

Експертна система – це комп'ютерна програма, що у деякій предметній галузі виявляє рівень пізнання, який еквівалентний рівню пізнання людини-експерта. Звичайно ця галузь строго обмежена. Однак кількість застосувань експертних систем величезна. До них входять: розуміння мови, аналіз зображень, прогноз погоди, оцінка майбутнього врожаю, медична діагностика, розроблення інтегральних схем, фінансування, керування повітряним рухом, ведення бою і багато інших прикладних галузей.

Кілька експертних систем уже вважають класичними. Прикладом можуть служити розроблені в Стенфордському університеті такі системи, як DENDRAL, яка визначає молекулярну структуру невідомої хімічної сполуки за допомогою даних мас-спектрометрії і MYCIN, яка визначає наявність

інфекції в пацієнта, що ідентифікує мікроорганізми, вибирає придатні ліки і призначає ефективний режим приймання ліків. В університеті Карнегі Мелона (США) розроблено також експертну систему XCON, що визначає конфігурацію комп'ютерних систем VAX фірми DEC, перевіряє специфікацію частин і правильність з'єднання в необхідну комп'ютерну систему.

2.4.1. Структура експертних систем

Щоб виконувати експертизу, комп'ютерна програма забезпечується засобами отримання логічних висновків з заданого набору даних, які можуть змінюватися. Програма має доступ до системи фактів, названої базою даних. Програма також в режимі консультації робить висновки з інформації, що містяться в базі даних. Деякі експертні системи можуть також використовувати нову інформацію, що додається під час консультації. Експертну систему можна зобразити, як систему, що складається з трьох частин:

1. База знань (БЗ).
2. Механізм висновку (МВ).
3. Система користувацького інтерфейсу (СКІ).

Взаємне розташування цих трьох частин показане на рис. 10.1.

База знань (БЗ), яка включає в себе динамічну базу даних і правила висновку – центральна частина експертної системи. Вона містить правила, що описують відносини або явища, методи і знання для розв’язання задач з галузі застосування системи. Можна подати БЗ як базу фактичних даних і набору правил для формування нових даних. Твердження "Джон Ф. Кеннеді був тридцять п’ятим президентом Сполучених Штатів" – приклад фактичних даних. "Якщо у вас болить голова, то прийміть дві таблетки цитрамону" – приклад знання для висновку. Сама БД звичайно записана на диску або іншому носії.

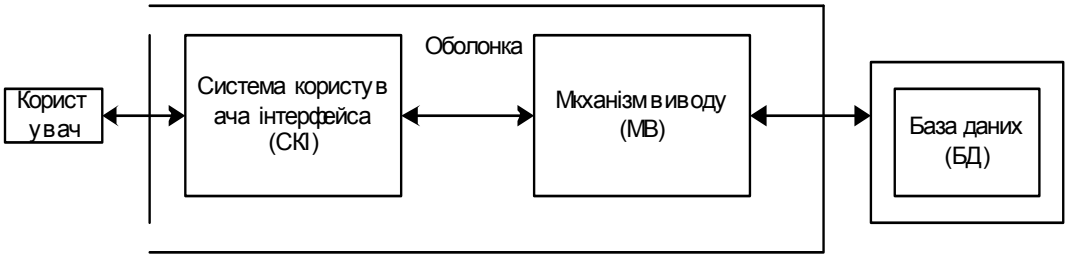


Рис. 2.10. Загальна структура експертної системи

Механізм висновку містить принципи і правила роботи. Механізм висновку "знає", як використати БД для отримання висновків з інформації, що міститься в ній.

Коли експертній системі задають питання, механізм висновку вибирає спосіб застосування правил БД для розв’язання задачі, поставленої в питанні. Фактично, механізм висновку запускає експертну систему в роботу, визначаючи, які правила потрібно викликати й організувати доступ до них у БД. Механізм висновку виконує правила, визначає, коли знайдені правильні рішення і передає результати програмі інтерфейсу з користувачем. Коли питання повинно бути попередньо обробленим, то доступ до БД здійснюється через інтерфейс із користувачем. Інтерфейс – це частина експертної системи, що взаємодіє з користувачем.

Як правило, користувачі мало знають про організацію БД, тому інтерфейс може допомогти їм працювати з експертною системою навіть тоді, коли вони не знають, як вона організована. Інтерфейс може також пояснити користувачеві, як експертна система виводить результат.

Система інтерфейсу приймає від користувача і передає йому інформацію. Іншими словами, система інтерфейсу повинна переконатися, що, після того, як користувач описав задачу, уся необхідна інформація отримана. Інтерфейс, ґрунтуючись на виді і природі інформації, заданої користувачем, передає необхідну інформацію механізмові висновку. Коли механізм висновку повертає знання, виведені з БД, інтерфейс передає їх назад користувачеві в зручній формі. Інтерфейс із користувачем і механізм висновку можуть розглядатися як "додаток" до БД. Вони разом утворюють оболонку експертної системи, як показано на рис. 10.1. Для БД, що містить велику і різноманітну інформацію, можуть бути розроблені і реалізовані кілька різних оболонок. Добре розроблені оболонки експертних систем зазвичай містять механізм для додавання і відновлення інформації в БД.

Як бачимо, експертна система складається з трьох основних частин. Взаємозв'язок між частинами може бути складним, залежно від природи й організації даних, а також від методів і цілей виведення. Наступні розділи описують ці аспекти експертних систем. Спочатку описується подання даних разом з деякими простими прикладами. Це описання застосовне як до систем, побудованих на правилах, так і до систем, що базуються на логіці. Потім розглядаються методи висновку. Далі наведене описання систем інтерфейсу з користувачем разом із прикладами обробки вхідних даних і висновку.

Розглянуто дві конкретні методики проектування експертних систем: систем, що базуються на правилах, і систем, що ґрунтуються на логіці.

2.4.2. Подання даних

Правило для проектування моделей даних – це організація даних у такій формі, що дає змогу легко здійснювати доступ до них за допомогою природних і простих механізмів. "Чим простіше, тим краще" – правило, яке потрібно пам'ятати, працюючи з поданням даних.

Експертні системи часто створюються "інженером із знань" (або проектувальниками експертних систем), що працюють з людиною-експертом, щоб закодувати знання експерта в БД. Проектувальник експертної системи повинен мати змогу маніпулювати отриманими знаннями і працювати з людиною-експертом. Ці роботи належать до галузі інженерії, що інтенсивно розвивається.

В експертних системах , описаних у цьому розділі мовою Турбо-Пролог, дані будуть завжди подані одним із двох способів. Перший спосіб – це класифікація і розміщення фактів і чисел (фрагментів фактичного знання) у правила Турбо-Прологу. Це представлення придатне для використання в експертних системах, що базуються на правилах. Інший спосіб – це організація фактів і числової інформації у твердженнях, що утворюють БД на твердженнях. Подання даних в твердженнях придатне для використання в експертних системах, які базуються на логіці.

Існують і інші системи подання даних. До них належать система на фреймах і розроблена нещодавно система на моделях. Система на фреймах використовує подання даних, основане на логічних групах атрибутів об'єкта. Для збереження та обробки логічні групи описують у фреймах. Для систем, що базуються на моделях, проект і структура системи засновані на знанні структури і поведінки пристрою, що є предметом дослідження. Наприклад – це експертна система, що моделює автомобіль.

Сьогодні системи, що ґрунтуються на правилах, найпопулярніші. Вони створені і використовуються в науковій та інженерній роботі, бізнесі. В цьому розділі розглядаються експертні системи, що базуються на логіці, які у природній спосіб укладаються в структуру мови Турбо-Пролог.

Конструювання експертної системи можна розпочати з таблиці, що складається з двох стовпчиків. Один стовпчик містить назви країн, а інший – назви відповідних столиць. Ця таблиця утворює маленьку БД:

```
Країна      Столиця
США         Вашингтон (Округ Колумбія)
АнгліяЛондон
Іспанія     Мадрид
```

Звичайно, таку таблицю використовують тільки для планування БД, в експертних системах способи подання даних більш відповідають мові програмування, яка застосовується для розробки системи. Твердження Турбо-Прологу, що відображає ці дані, можна записати в такий спосіб:

```
capital("Washington DC","USA").
capital("London","England").
capital("Madrid","Spain").
```

Подібні твердження утворюють базис експертної системи, основаної на логіці.

Ці самі дані можна подати у формі правил "якщо–то". Правила для попередніх трьох тверджень виглядають так:

```
capital_is("Washington DC") :-
country(is,"USA"),!.
capital_is("London") :-
country(is,"England"),!.
capital_is("Madrid") :-
country(is,"Spain"),!.
```


Ці правила можуть бути основою експертної системи, основаної на правилах. Як бачимо, подання даних в експертній системі те саме, що і подання фактів і правил.

Наведені приклади, звичайно, прості, однак, вони дуже корисні для демонстрації принципів побудови експертних систем мовою Турбо-Пролог. У наступному розділі буде показано, як можна використовувати таке подання даних на практиці.

2.4.3. Методи висновку

Метод висновку – це систематичний спосіб для доведення того, що з безлічі припущень випливає деякий висновок. Цей систематичний метод закладений у правилах висновку, що специфікують прийняту у логіці специфіку отримання висновку. Висновок здійснюється за допомогою пошуку і зіставлення зі зразком. Інші мови вимагають написання власних правил пошуку і порівняння зі зразком. У Турбо-Пролозі ці задачі виконуються за допомогою внутрішніх програм уніфікації, тому потрібно

тільки написати необхідну специфікацію. Як і у системах, що базуються на правилах, так і в системах, що ґрунтуються на логіці, користувач одержує відповіді на свої запити відповідно до логіки, закладеної в системі. У першому випадку запити користувача трансформуються у форму, що порівнюється з формою правил БД. Механізм висновку ініціалізує зіставлення, починаючи з "верхнього" правила. Звертання до правила називають "викликом". Виклик відповідних правил під час зіставлення триває доти, доки не відбулося зіставлення або не вичерпана вся БД, а зіставлення не знайдене. У другому випадку трансформовані запити є значеннями, що порівнюються зі значеннями, які містяться в БД.

Якщо механізм висновку виявляє, що можна викликати більше одне правило, то необхідно здійснити визначений вибір. Пріоритет віддається звичайно або правилам, що є більш вагомими, або правилам, що враховують більше поточних даних. Цей процес називають дозволом конфлікту.

Щоб продемонструвати процес висновку, припустимо, що потрібно з'ясувати, чи є Мадрид столицею Іспанії. Для питання "Мадрид – столиця Іспанії?" механізм висновку в системі, що базується на логіці, утворить:

```
capital("Madrid","Spain").
```

Якщо факт порівняння знайдений у системі, то вона видає відповідь "правильно".

Система на правилах використовує форму у вигляді правила для пошуку відповіді (В або ПРО) на питання (?) залежно від заданих мов (У або умова). Синтаксис правила такий: "якщо в БД є правило виду "якщо <у> тоді В ", то шукай <умову>, щоб отримати відповідь ПРО". Правило формулюється у вигляді:

```
capital_is("Madrid"):-
country(is,"Spain"), !.
```

Це приклад зворотного висновку. Висновок із правила описано і механізм висновку шукає в БД всі умови, що приводять до нього.

Експертні системи мовою Турбо-Пролог, розглянуті в цьому розділі, поєднують методи висновку, що базуються на логіці, і методи, що ґрунтуються на правилах.

2.4.4. Система користувацького інтерфейсу

Система користувацького інтерфейсу забезпечує взаємодію між експертною системою і користувачем. Ця взаємодія звичайно складається з кількох функцій:

1. Опрацювання даних, отриманих із клавіатури, виведення даних на екран.

2. Підтримка діалогу між користувачем і системою.

3. Розпізнавання ситуації нерозуміння між користувачем і системою.

4. Забезпечення "дружності" стосовно користувача.

Система інтерфейсу з користувачем повинна ефективно обробляти введення і висновок. Для цього необхідно обробляти дані, що вводяться чи виводяться, швидко, у зрозумілій і виразній формі. Необхідно також передбачити можливість роботи з додатковими – засобами такими, як друкувальні пристрої, магнітні диски і додаткові файли даних.

Крім того, система інтерфейсу повинна підтримувати відповідний діалог між користувачем і системою.

Діалог – це загальна форма консультації з експертною системою. Консультація повинна закінчуватися зрозумілим твердженням, виданим системою, з поясненням послідовності висновку, який наведений до цього твердження.

Система користувацького інтерфейсу повинна також розпізнати нерозуміння між користувачем і системою, що виникло або через помилку, або на принциповій основі. Система повинна відповідно реагувати на цю ситуацію. Наприклад, не повинно відбутися збою системи, якщо користувач вводить 1, коли очікується "так" або "ні", або коли користувач задасть безглузде питання.

Здатність експертної системи моделювати людину-експерта може змінюватися від простих пізнавальних процесів до введення нових даних або нових способів розв’язання задачі. Система інтерфейсу повинна інформувати користувача про методику роботи системи і її розвитку, якщо такий розвиток передбачено в системі.

Нарешті, система користувацького інтерфейсу має бути "дружньою" до користувача. Наприклад, послідовність меню, що показує задачі, які користувач може вибрати, є необхідною рисою експертної системи. Користувач також повинен мати можливість взаємодіяти з експертною системою у природний спосіб. В ідеалі користувач повинен мати змогу використовувати природну мову. Надрукувати "Що рекомендується від алергічного головного болю?" легше, ніж ввести:

```
Medication (Prescription,"allergy headache").
```

Хоча обробка природної мови дуже потрібна в експертних системах, таку можливість важко спроектувати і реалізувати. Експертні системи мовою Турбо-Пролог, описані в цьому розділі, використовують меню.

2.4.5. Контрольні питання та вправи

1. Структура експертної системи.
2. Означення експертної системи та типи експертних систем.
3. Що таке система інтерфейсу користувача?
4. Основні функції інтерпретатора експертної системи.
5. Способи подання даних в експертних системах.
6. Основні функції взаємодії експертної системи з користувачем.
7. На мові логіки написати найпростішу експертну систему, яка на запит користувача вилдає назву будь-якої країни Європи, її столицю, площу і густину населення на квадратний кілометр площі.
8. На мові логіки реалізувати експертну систему аналогічну п. 7 для країн СНД, яка , окрім вказаних функцій, здійснює порівняння рівня життя кожної країни СНД з Україною за рівнем доходу на душу населення.

2.5. Описання, проектування та реалізація експертних систем, що ґрунтуються на правилах та логіці

2.5.1. Описання експертних систем, що ґрунтуються на правилах і логіці

У всіх експертних системах існує залежність між вхідним потоком даних і даними в БД. Під час консультації вхідні дані порівнюються з даними в БД. Результатом зіставлення є негативна або позитивна відповідь. У системі, що базується на правилах, підтверджений результат є дією одного з продукційних правил. Ці продукційні правила визначаються вхідними даними.

Отже, експертна система, що базується на правилах (мовою Турбо-Пролог) містить безліч правил, які викликають за допомогою вхідних даних в момент зіставлення. Експертна система також містить інтерпретатор у механізмі виведення, що вибирає й активізує різні модулі системи. Роботу цього інтерпретатора можна описати послідовністю трьох кроків:

1. Інтерпретатор порівнює зразок правила з елементами даних у БД.
2. Якщо можна викликати більше ніж одне правило, то інтерпретатор використовує механізм дозволу конфлікту для вибору правила.

3. Інтерпретатор застосовує вибрані правила, щоб знайти відповідь на питання.

Цей процес інтерпретації є циклічним і називається циклом "розпізнавання дії".

У системі, що базується на правилах, кількість продукційних правил визначає розмір БД. Деякі найскладніші системи мають БД з більш ніж 5000 продукційних правил. Ви можете розпочати з невеликої кількості правил і додавати їх в БД у процесі розширення експертної системи.

Однак, важливішою, ніж розміри БД, є структура самих продукційних правил. Проектувальник БД відповідає за побудову сумісних правил. Сьогодні не існує строгих принципів, якими треба керуватися, проектуючи структуру правил. З цього приводу лише ведуться дискусії. Однак за останні кілька років деякі рекомендації стали очевидними і їх потрібно виконувати якомога точніше:

1. Використовувати якнайменшу кількість умов для визначення продукційного правила.

2. Уникати суперечливих продукційних правил.

3. Конструювати правила, спираючись на структуру, притаманну предметній галузі.

Перша експертна система мовою Турбо-Пролог в цьому розділі – система для ідентифікації породи собак. Вона допомагає потенціальному господареві вибрати породу собаки відповідно до певних критеріїв.

Припустимо, що користувач повідомив безліч характеристик собаки у відповідь на питання експертної системи. Інтерпретатор працює в циклі розпізнавання дії. Якщо характеристики порівняні з характеристиками породи собаки, що утворюють частину БД, тоді викликають відповідне продукційне правило і в результаті ідентифікується порода. Потім результат повідомляється користувачеві. Аналогічно, якщо порода не ідентифікована, це теж повідомляється користувачеві.

Тепер розглянемо характеристики двох порід собак, які містяться в БД. Гонча має коротку шерсть, зріст менше ніж 22 дюймів, довгі вуха і хороший характер. Датський дог має коротку шерсть, хвіст, що звисає, довгі вуха, добрий характер і вагу більше ніж 100 фунтів.

Як видно з цього описання, обидві породи мають коротку шерсть, довгі вуха і хороший характер. Зріст гончої менше ніж 22 дюймів тоді як нічого не сказано про зріст дога. Дог має хвіст, що звисає і вагу більш ніж 100 фунтів – характеристики, відсутні у гончої. Описання двох собак у термінах вказаних

характеристик достатньо, щоб розрізнити ці дві породи, і навіть відрізнити їх від будь-якої іншої породи в БД.

Необхідні продукційні правила можуть бути складені за зазначеними характеристиками:

```
dog_is ("Beagle") :-
it_is ("short-haired dog") ,
positive (has,"height under 22 inches") ,
positive (has,"long ears"),
positive (has,"good natured personality"), !.
dog_is ("Great Dane"):-
it_is("short-haired dog"),
positive(has,"low-set tail"),
positive(has,"longer ears"),
positive(has,"good natured personality"),
positive(has,"weight over 100 lb"), !.
```

У попередньому правилі довжина шерсті може бути описана за допомогою предиката **positive** у вигляді:

```
positive(has,"short-hair").
```

Але використання предиката it_is дає змогу обмежити "простір пошуку" (кількість даних, що перевіряють, шукаючи розв’язання) одним піддеревом деревоподібної структури, що містить інформацію про різні породи собак (див. рис. 11.1). Експертна система, що базується на правилах, дає змогу проектувальникові (програмістові) будувати правила, що у природний спосіб поєднують у групи зв'язані фрагменти даних. Кожне продукційне правило може бути незалежним від інших. Ця незалежність робить базу продукційних правил семантично модульною, тобто групи інформації не впливають один на одного. Більше того, модульність бази правил дає змогу розвивати БД, розширюючи її. Ця особливість украй необхідна в багатьох прикладах. Турбо-Прологу дає змогу легко її реалізувати в експертній системі.

В експертних системах, що базуються на логіці, БД складається з тверджень у вигляді пропозицій логіки предикатів. Такі пропозиції можуть групуватися, утворити БД Турбо-Прологу. Правила можуть або описувати дані, або управляти процесом внутрішньої уніфікації Турбо-Прологу.

Так само, як і у системі на правилах, експертна система, яка базується на логіці, має безліч правил, що можуть викликатися за допомогою даних із вхідного потоку. Система має також інтерпретатор, що може вибирати й активізувати модулі, які залучені до роботи системи.

Інтерпретатор виконує різні функції усередині системи на основі такої схеми:

1. Система має пропозиції в БД, що керують пошуком і порівнянням. Інтерпретатор зіставляє ці пропозиції з елементами даних у БД.

2. Якщо існує можливість виклику більше ніж одного правила, то система використовує можливості Турбо-Прологу для розв'язання конфлікту. Отже, користувачеві-програмістові не потрібно розглядати потенційно можливі конфлікти.

3. Система отримує результати уніфікованого процесу автоматично, тому вони можуть передаватися на потрібний пристрій виведення інформації.

Так само, як і у системі, що базується на правилах, цей циклічний процес є процесом розпізнавання – дії. Краса і великі можливості системи, основані на логіці, полягають у тому, що вона відображає структуру самого Турбо-Прологу. Цим пояснюється той факт, що вона дуже ефективна в роботі.

Найважливішим аспектом для БД у системі, основаній на логіці, є проектування БД, побудова її тверджень та вибір їхньої структури. БД повинна мати двохвимірну логічну організацію, і містить мінімум надлишкової інформації. Так само, як і у системі, що базується на правилах, мінімально достатня кількість даних утворює найефективнішу систему. Твердження БД для гончої і дога виглядають так:

```
rule(1,"dog","Beagle",[1,2,3,4]).
rule(2,"dog","Great Dane",[1,5,3,4,6]).
cond(1,"short-haired").
cond(2,"height under 22 inches").
cond(3,"longer ears").
cond(4,"good natured personality").
cond(5,"low-set tail").
cond(6,"weight over 100 lb").
```

Відзначимо, що в кожній пропозиції типу **rule** перший аргумент – номер правила, другий аргумент – тип об'єкта ("собака") і третій аргумент – порода собаки. У нас це гонча або дог. Список цілих чисел задає номери умов із прикладу типу ***cond (умова)***. Пропозиції типу **cond** містять усі характеристики для будь-якої породи, поданої в БД.

Списки номерів умов служать для збереження безлічі фактів, згідно з якими вибирають пропозиції типу **rule**. Інтерпретатор в експертній системі,

що базується на логіці, використовує ці номери умов, щоб зробити відповідний вибір.

Додавання і відновлення пропозицій БД є простими операціями. Експертні системи, що ґрунтуються на логіці, легко проектувати, розвивати і підтримувати в Турбо-Пролозі, тому що в міру розширення БД програма не потребує модифікації. Розширення насамперед полягає в поступовому додаванні нових відповідностей.

Розробка експертної системи вимагає великої організаційної уваги до деталей. Ця вимога змінюється, відповідно, зі зміною розміру і складності ї експертної системи, яка розробляється. Якщо експертна система, яку ви хочете створити, у остаточному підсумку може містити сотні продукційних правил, то важко визначити ефект від додавання додаткових правил. У Турбо-Пролозі продукційні правила містяться в програмі, і, отже, розміри програми збільшуються в міру додавання правил. Розміри пам'яті, зрештою, обмежують кількість правил. У такому разі використання системи на правилах є проблематичним. Водночас, у системі, що базується на логіці, де БД міститься у файлі на диску, обмеження на розміри БД не накладаються. Тому система, основана на логіці, переважає у такому разі.

Якщо ж наша експертна система буде містити не більш від декількох сотень правил, використання системи, що базується на правилах доцільніше. Оскільки продукційні правила майже не залежать один від одного, створення і тестування такої експертної системи простіше. Просто здійснюється і заміна правил щоб вивчити ефект, викликаний такою зміною. У системах, що ґрунтуються на логіці, зміна параметрів усередині БД повинна здійснюватися більшою обережно, тому що зміни менш помітні, а результат може бути руйнівним, а відновлення складним.

Якщо швидкість є головною вимогою до створюваної експертної системи, то можна вибирати або логічну систему, що повністю міститься в оперативній пам'яті, або систему, що базується на правилах, – обидві будуть працювати добре. Якщо, однак, експертна система повинна містити велику БД, то в розробника існує лише один варіант – логічна система, що розміщена на диску.

Після того, як вибір між системою, що базується на правилах і системою, яка ґрунтується на логіці, зроблений і ретельно вивчені дані, що ввійдуть у БД, можна розпочати проектування БД, яка містить специфіковані функції і має необхідні властивості.

Наступним кроком є розробка діаграм потоків даних і структурної схеми експертної системи. Це допоможе сконструювати модулі, що утворюють систему. Потім можна розпочати написання програми, з урахуванням з діаграми потоків даних і структурної схеми. Після закінчення програмування необхідно перевірити результати за допомогою людини-експерта, що бере участь у проекті.

Дуже добре ілюструє обидва розглянуті методи (метод побудови системи, що базується на правилах, і системи, що ґрунтується на логіці) програма для вибору породи собаки. У такий спосіб є можливість порівняти два різні підходи під час роботи з тими самими даними. У різних експертних системах для вибору породи собак основні дані однакові. Вибрані поширені породи собак. Класифікація в БД може мати деревоподібну структуру, як показано на рис. 2.11. Відповідно до цієї деревоподібної класифікації породи розділені на короткошерсті і довгошерсті. Англійський бульдог, гонча, дог і американський фокстер'єр є короткошерстими, а кокер-спаніель, ірландський сестер, колі і сенбернар – довгошерстими.

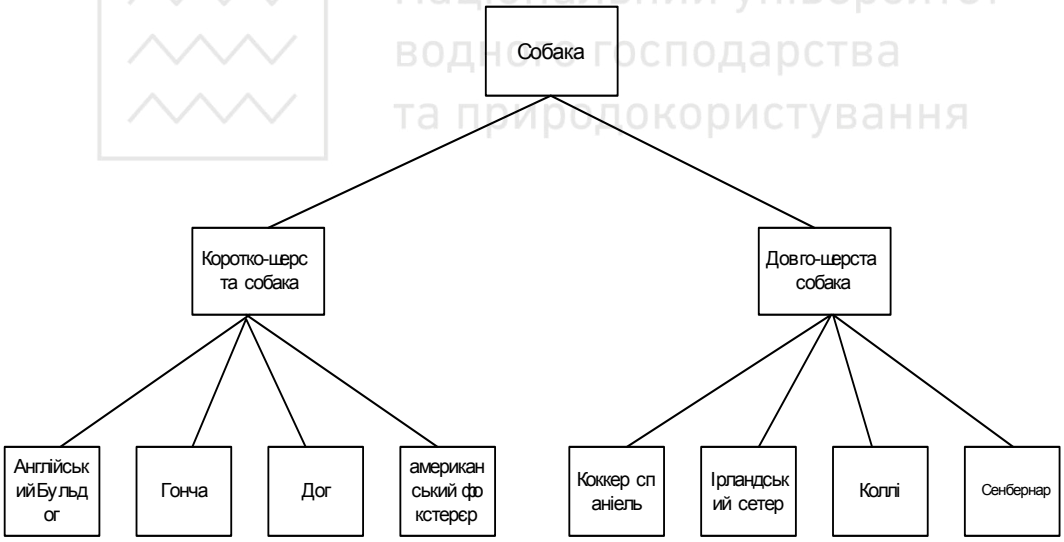


Рис. 2.11. Деревоподібна структура БД експертної системи для вибору породи собаки

Для ідентифікації породи усередині кожної підмножини можна використовувати список атрибутів. Кількість характеристик буде визначати ступінь точності класифікації. Для ідентифікації застосовують усю сукупність атрибутів, немає якої-небудь однієї характеристики розпізнавання. Усі вісім перерахованих нижче атрибутів є необхідними, тому що жоден з них не характерний для всіх порід одночасно. Атрибут характеризує всі

об'єкти одночасно і, можливо, не є необхідним у безлічі даних. В нашій експертній системі використовують такі характеристики:

1) коротка шерсть;
2) довга шерсть;
3) зріст менше ніж 22 дюймів;
4) зріст менше ніж 30 дюймів;
5) низько посаджений хвіст;
6) довгі вуха;
7) хороший характер;
8) вага більше за 100 фунтів.

Кожна характеристика для конкретної породи або правильна, або неправильна. Для кожної породи справедливі такі характеристики:

Порода	Характеристики
Англійський бульдог	1,3,5,7
Гонча	1,3,6,7
Дог	1,4,6,7,8
Американська гонча	1,5,6,7,
Кокер спанієль	2,3,5,6,7
Ірландський сетер	2,4,6
Колі	2,4,5,7
Сенбернар	2,5,7,8,

Спосіб використання цієї інформації залежить від реалізації експертної системи. У нашому випадку під час проектування БД деревоподібна структура, безліч ідентифікаційних характеристик і набори номерів характеристик для кожної породи утворюють робочу модель БД для вибору породи.

2.5.2. Проектування систем, що базуються на правилах

Коли спроектована БД, можна написати програму мовою Турбо-Пролог для маніпулювання нею. Для простоти і більшої зрозумілості, у програмі, що розглядається в цьому розділі, основна увага приділяється процесам, що виникають під час консультації з експертною системою. У ці процеси входить управління потоком даних, що надходять від користувача, одержання інформації з БД і видання результатів консультації. Для ілюстрації цих процесів на рис. 2.11 наведена діаграма потоку даних.

На цьому рисунку показані лінії потоку даних, що виходять із клавіатури. Ці лінії позначають потік вхідних даних від користувача. Лінії

потоків даних виходять також з БД. Ці лінії відповідають даним, отриманим з БД. Лінії потоку даних, що йдуть до монітора, відповідають виведеним на екран даним.

Тепер, ґрунтуючись на діаграмі потоків даних, можна розробити структурну схему, щоб зафіксувати організацію програмних модулів і правил. Рис. 2.12 демонструє остаточну схему.

На рис. 2.12 можна бачити, що головним модулем (або метою) є – **do_expert_job** (виконай експертну роботу). Модулі **ask(X,Y)** (запитай) і **remember(X,Y,Reply)** (запам'ятай) керують надходженням даних від користувача. Інші модулі оновлюють робочі дані і видають результати консультації.

Використовуючи цю схему, можна тепер розробити експертну систему мовою Турбо-Пролог, що базується на правилах. Спочатку необхідно зробити декларації БД. БД буде зберігати відповіді користувача на питання системи користувацького інтерфейсу (СКІ). Ці дані є ствердними або негативними відповідями. Далі потрібно оголосити предикати для виконання висновку (механізм висновку) і для взаємодії з користувачем (система користувацького інтерфейсу).

Усі разом це такі декларації:

```
database
xpositive(symbol,symbol)
xnegative(symbol,symbol)
predicates
do_expert_job
do_consulting
ask(symbol,symbol)
dog_is(symbol)
it_is(symbol)
positive(symbol,symbol)
negative(symbol,symbol)
remember(symbol,symbol,symbol)
```

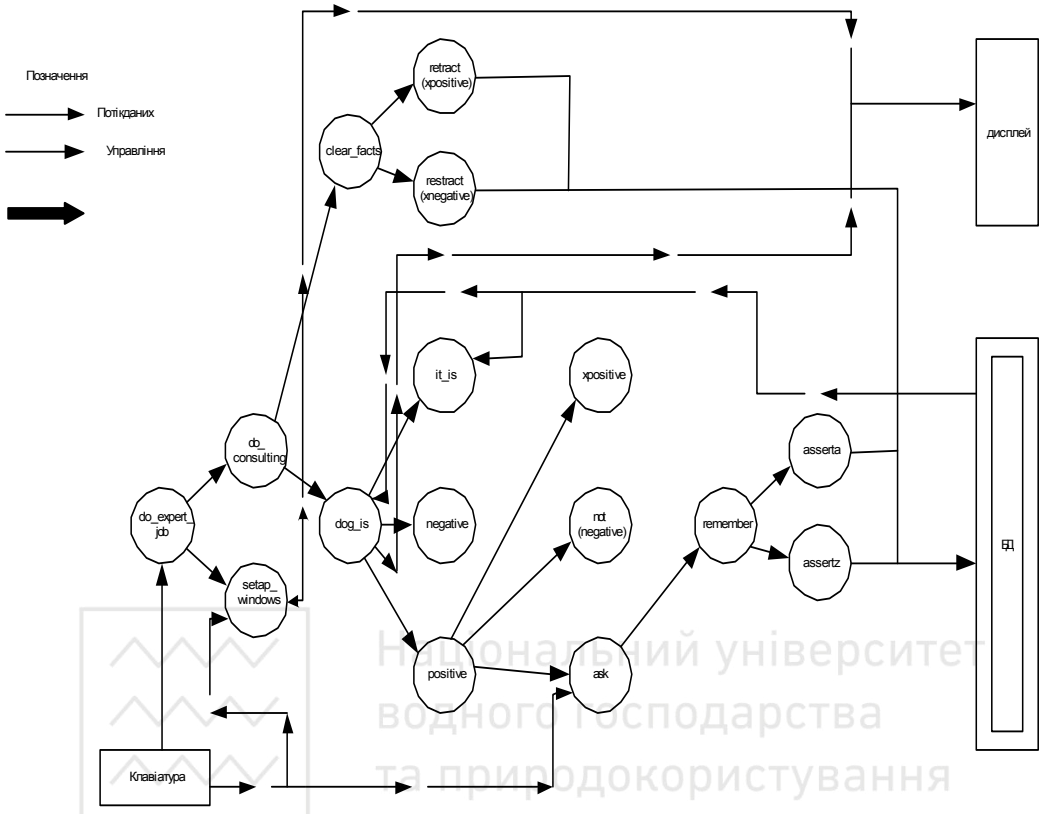


Рис. 2.12. Діаграма потоків даних для експертної системи, що базуються на правилах, для вибору породи собаки

`clear_facts`

Предикати БД **`xpositive`** і **`xnegative`** використовуються для збереження ствердних і негативних відповідей користувача. Перші чотири предикати потрібні для взаємодії з користувачем, а інші шість – для механізму висновку.

Повинні бути складені вісім продукційних правил: по одному для кожної породи. Кожне правило повинно описувати породу за ознакою належності до групи довгошерстих або короткошерстих. Це головні підкатегорії, як випливає з даних, і як показано на деревоподібній структурі (див. рис. 2.13). Правило **`it_is`** виконує цю ідентифікацію. Потім правило **`positive`** ідентифікує характеристики собаки в кожному випадку. І **`it_is`**, і **`positive`** використовують механізм висновку. Нижче наведено повне продукційне правило для кокер-спанієля:

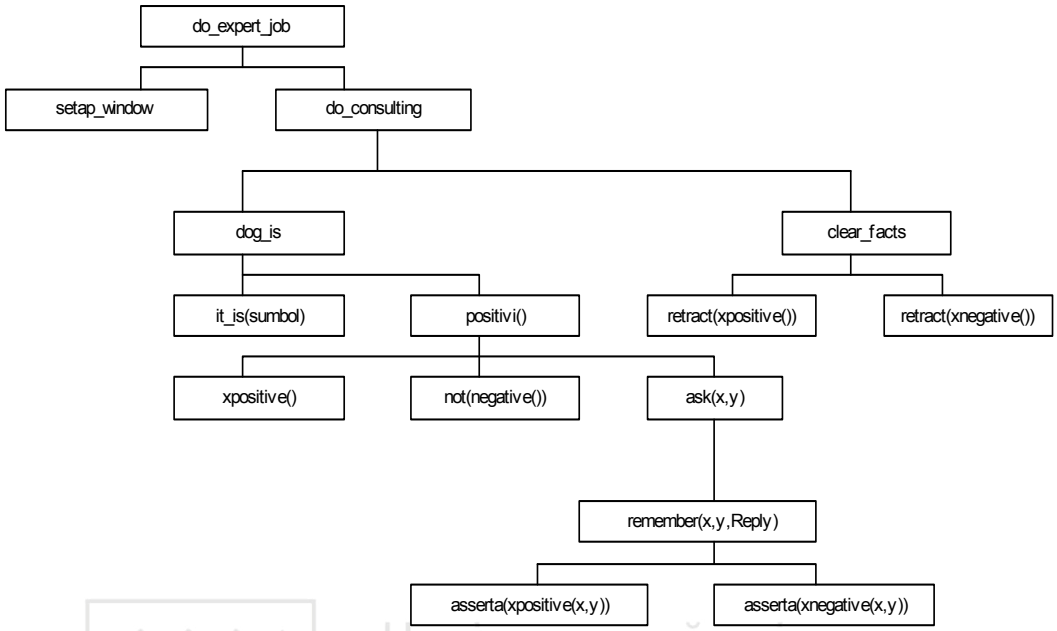


Рис. 2.13. Структурна діаграма для експертної системи, яка базується на правилах, для вибору породи собаки

```
dog_is("Cocker Spaniel") :-
it_is("long-haired dog"),
positive(has,"height under 22 inches"),
positive(has,"low set tail"),
positive(has,"longer ears"),
positive(has,"good natural personality"),!.
```

Механізм висновку повинен мати правила для керування даними, що вводить користувач, для порівняння їх із продукційними правилами і збереження "траси" (або запам'ятовування) негативних і позитивних відповідей. Правила **positive** і **negative** використовують для зіставлення даних користувача з даними в продукційних правилах. Правило **remember** (запам'ятовування) виконує додавання пропозицій з відповідями **yes** (так) і **no** (немає), котрі можуть використовуватись при зіставленні зі зразком:

```
positive(X,Y) :-
xpositive(X,Y),!.
positive(X,Y) :-
not(negative(X,Y)),!,
```

```
ask(X,Y).
negative(X,Y) :-
xnegative(X,Y),!.
remember(X,Y,yes) :-
asserta(xpositive(X,Y)).
remember(X,Y,no) :-
asserta(xnegative(X,Y)),
fail.
clear_facts :-
retract(xpositive(_,_)),
fail.
clear_facts :-
retract(xnegative(_,_)),
fail.
```

Призначення системи користувацького інтерфейсу (СКІ) – зв'язок даних, що вводить користувач, із системою логічного висновку. Головний модуль **do_expert_job** (виконує експертну роботу) і модуль **do_consulting** (виконує консультацію) здійснюють цей зв'язок. Модуль **ask(X,Y)** (запитай) формує запит до користувача і зберігає відповіді в БД. Крім того, вікно забезпечує додаткову зручність під час консультації. Система користувацького інтерфейсу наведена нижче:

```
do_expert_job :-
setup_window,
do_consulting,
write("Press space bar."),nl,
readch(_),
removewindow,
exit.
setup_window :-
makewindow(1,7,7,"AN EXPERT SYSTEM",1,16,22,58),
nl,write("* * * * * * * * * * * * * * * * * * * * * * *
*"),
nl,write("            A Dog Expert  "),
nl,write(" "),
nl,write("This is a dog identification system."),
nl,write("Please answer the question about    "),
nl,write("the dog you would like by typing in "),
```

```
    nl,write("'yes' or 'no'.  "),
    nl,write("* * * * * * * * * * * * * * * * * * * * *
*"),
    nl,nl.
    do_consulting :-
    dog_is(X),!,nl,
    write("the    dog    you    have    indicated    is
a(n)",X,"."),nl,
    clear_facts.
    do_consulting :-
    nl,write("Sorry I can't help you ! "),
    clear_facts.
    ask(X,Y) :-
    write("  Question :- ",X," it ",Y," ?"),
    readln(Reply),
    remember(X,Y,Reply).
```

Зверніть увагу, що головний модуль **do_expert_job** викликає модулі **setup_window** (установи вікно) і **do_consulting** (виконує консультацію). Консультаційний модуль має дві альтернативні форми. Перша взаємодіє з механізмом висновку; якщо результат циклу "розпізнавання – дія" позитивний, то результат повідомляють користувачеві. Друга форма повідомляє про негативний результат.

Тепер можна з'єднати окремі компоненти і сформувати повну експертну систему на правилах для вибору породи собаки. Лістинг 2.4 містить цю програму.

Лістинг 2.4

```
/* Програма: Експерт з порід собак Файл:prog1001.pro */
/* Призначення: Демонстрація роботи експертної системи. */
/* Це продукційна система, що базується на правилах     */
/* Зауваження: це система для ідентифікації породи.  */
/* Система використовує безліч продукційних правил   */
/* для висновку рішення  */
domains
database
xpositive(symbol,symbol)
xnegative(symbol,symbol)
```

```
predicates
do_expert_job
do_consulting
ask(symbol,symbol)
dog_is(symbol)
it_is(symbol)
positive(symbol,symbol)
negative(symbol,symbol)
remember(symbol,symbol,symbol)
clear_facts
goal
do_expert_job
clauses
/* Система користувацького інтерфейсу (СПИ) */
do_expert_job :-
makewindow(1,7,7,"AN EXPERT SYSTEM",1,16,22,58),
nl,write("* * * * * * * * * * * * * * * * * * * *"),
nl,write("  WELCOME TO A DOG EXPERT SYSTEM   "),
nl,write("      "),
nl,write("This is a dog identification system.    "),
nl,write("Please answer the question about   "),
nl,write("the dog you would like by typing in     "),
nl,write("'yes' or 'no'. "),
nl,write("* * * * * * * * * * * * * * * * * * * *"),
nl,nl.
do_consulting,
write("Press space bar."),nl,
readch(_),
removewindow,
exit.
do_consulting :-
dog_is(X),!,nl,
write("the dog you have indicated is a(n)",X,"."),nl,
clear_facts.
do_consulting :-
nl,write("Sorry I can't help you ! "),
clear_facts.
```

```
ask(X,Y) :-
write("  Question :- ",X," it ",Y," ?"),
readln(Reply),
remember(X,Y,Reply).
/*   МЕХАНІЗМ ВИСНОВКУ   */
positive(X,Y) :-
xpositive(X,Y),!.
positive(X,Y) :-
not(negative(X,Y)),!,
ask(X,Y).
negative(X,Y) :-
xnegative(X,Y),!.
remember(X,Y,yes) :-
asserta(xpositive(X,Y)).
remember(X,Y,no) :-
asserta(xnegative(X,Y)),
fail.
clear_facts :-
retract(xpositive(_,_)),
fail.
clear_facts :-
retract(xnegative(_,_)),
fail.
/*  ПРОДУКЦІЙНІ ПРАВИЛА  */
dog_is("English Bulldog") :-
it_is("short-haired dog"),
positive(has,"height under 22 inches"),
positive(has,"low-set tail"),
positive(has,"good natured personality"),!.
dog_is("Beagle") :-
it_is("short-haired dog"),
positive(has,"height under 22 inches"),
positive(has,"longer ears"),
positive(has,"good natured personality"),!.
dog_is("Great Dane") :-
it_is("short-haired dog"),
positive(has,"low-set tail"),
```

```
positive(has,"good natured personality"),
positive(has,"weight over 100 lb"),!.
dog_is("American Foxhound") :-
it_is("short-haired dog"),
positive(has,"height under 30 inches"),
positive(has,"longer ears"),
positive(has,"good natured personality"),!.
dog_is("Cocker Spaniel") :-
it_is("long-haired dog"),
positive(has,"height under 22 inches"),
positive(has,"low-set tail"),
positive(has,"longer ears"),
positive(has,"good natured personality"),!.
dog_is("Irish Setter") :-
it_is("long-haired dog"),
positive(has,"height under 30 inches"),
positive(has,"longer ears"),!.
dog_is("Collie") :-
it_is("long-haired dog"),
positive(has,"height under 30 inches"),
positive(has,"low-set tail"),
positive(has,"good natured personality"),!.
dog_is("St. Bernard") :-
it_is("long-haired dog"),
positive(has,"low-set tail"),
positive(has,"good natured personality"),
positive(has,"weight over 100 lb"),!.
it_is("short-haired dog") :-
positive(has,"short-haired"),!.
it_is("long-haired dog") :-
positive(has,"long-haired"),!.
/*КІНЕЦЬ ПРОГРАМИ        */
```


Ця програма просить користувача вибрати режим консультації або вихід із програми. Потім експертна система вибирає породу собаки на підставі відповідей користувача на питання або наприкінці невдалого пошуку видає

повідомлення **Sorry I can't help you** (вибачте, я не можу допомогти Вам). На рис. 2.14 показано екран під час консультації.

Програмні правила висновку, спроектовані відповідно до описаного процесу, гарантують нормальне виконання програми під час діалогу з користувачем і пошуку, забезпечуючи зручний і "дружній" до користувача діалог. Внутрішні програми уніфікації Турбо-Прологу, його можливості пошуку і порівняння зі зразком забезпечують ефективний і вичерпний процес.

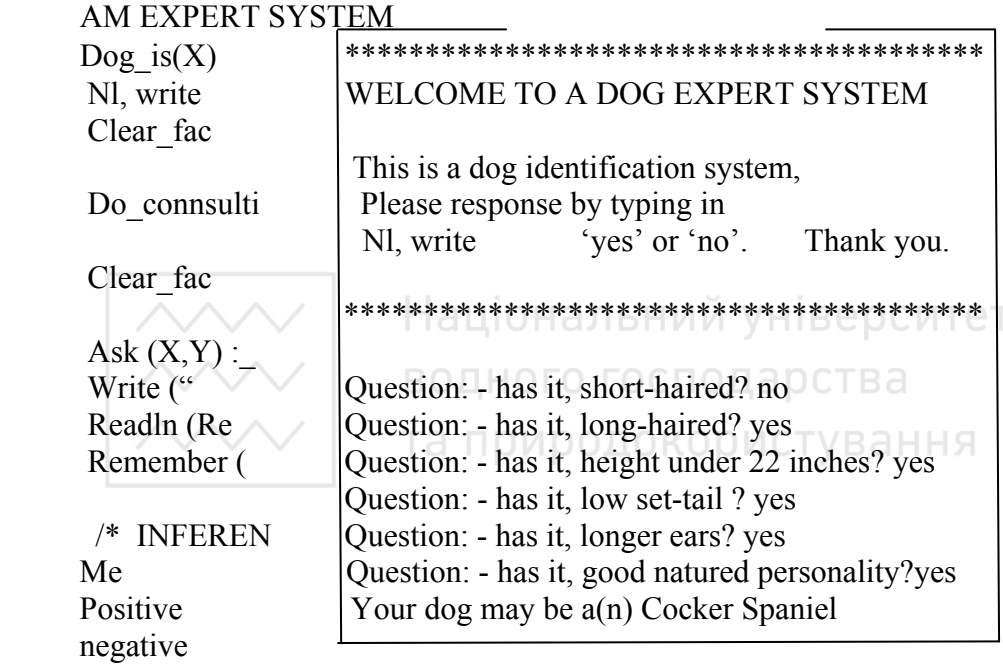


Рис. 2.14. Діалог з експертною системою, що базується на правилах, для вибору породи собаки

2.5.3. Реалізація систем, що ґрунтуються на логіці

Структура експертної системи, основаної на логіці, аналогічна до структури експертної системи, що базується на правилах. Знову для простоти і більшої зрозумілості у розглянутій у цьому розділі програмі основна увага відведена консультації з експертною системою. Програма повинна керувати потоком даних від користувача, виводити висновки з БД і видавати результати консультації. Діаграма, що ілюструє потоки даних, наведена на

рис. 2.15.

На рис. 2.15 вихідні з клавіатури лінії потоку даних являють собою вхідний потік даних від користувача. Інші лінії потоку даних, що виходять з БД, позначають вилучені дані. Лінії потоку даних також спрямовані до відеоекрана. Зважаючи на діаграму потоку даних, можна сконструювати структурну схему, як показано на рис. 2.16.

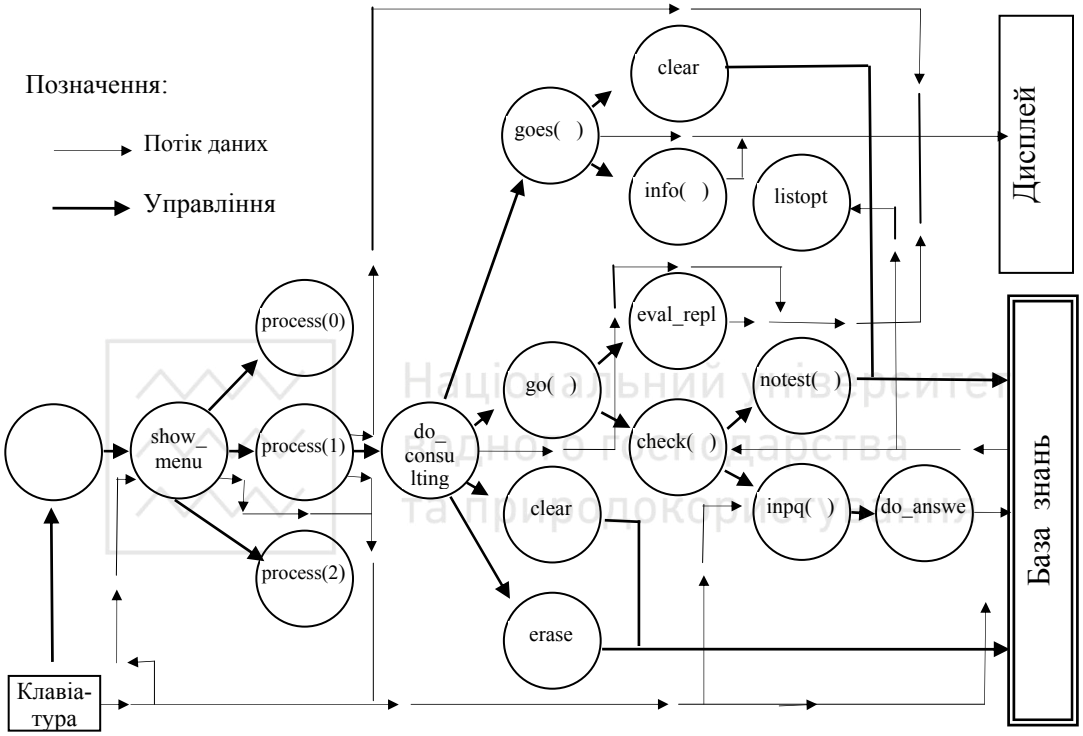


Рис. 2.15. Діаграма потоків даних для експертної системи, що базується на логіці, для вибору породи собаки

Структурна схема показує, що головний модуль **`do_expert_job`** викликає модуль **`show_menu`**. Цей модуль пропонує користувачеві вибрати програмну функцію. Відповідь користувача зчитується в цілочислову змінну **`Choice`** (вибір) і виклик **`process(Choice)`** приводить до виконання відповідної програмної функції. Модулі **`process(0)`** і **`process(2)`** служать для виходу з програми. Модуль **`process(1)`** викликає модуль **`do_consulting`** (виконуючий консультацію). Різні модулі, викликані **`do_consulting`**, видають породи собак, виконують висновок і оновлюють робочі дані. Модуль **`eval_reply`** забезпечує зручне закінчення діалогу консультації.

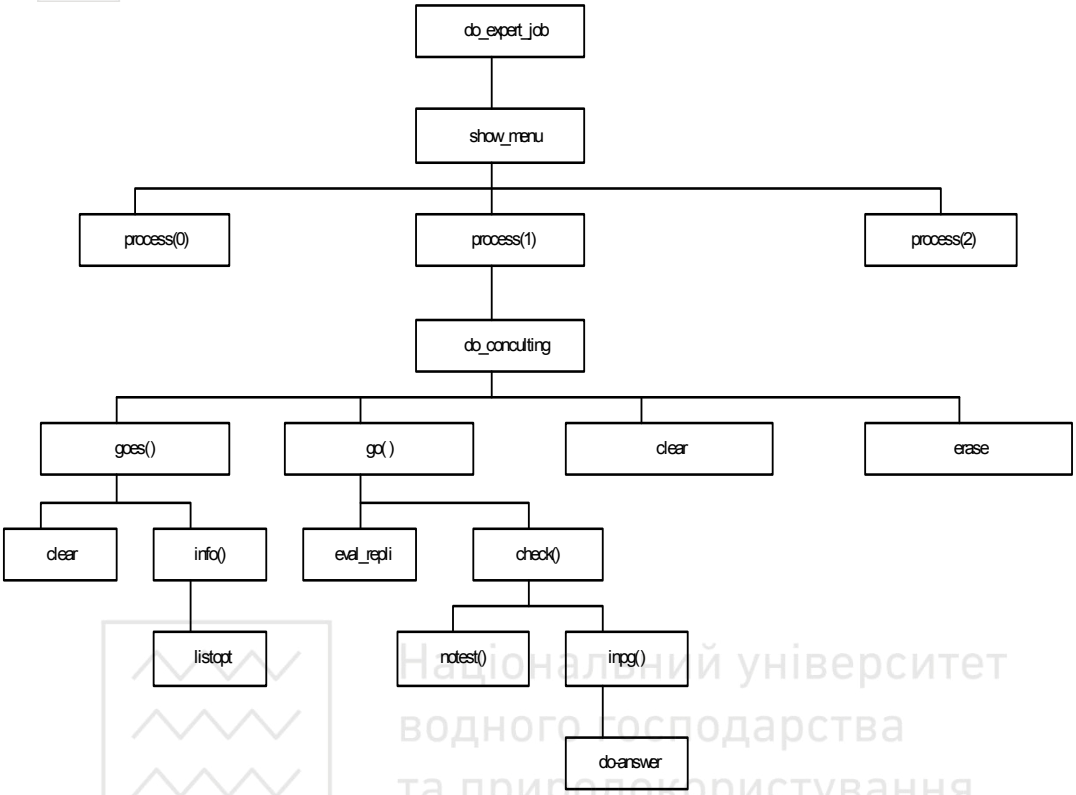


Рис. 2.16. Структурна діаграма для експертної системи, яка базується на логіці, для вибору породи собаки

Тепер можна розпочати розробку мовою Турбо-Пролог експертної системи, що базується на логіці. Перший розділ, який потрібно написати, – це розділ

```
domains.
domains
CONDITIONS  = BNO *
HISTORY     = RNO *
RNO, BNO, FNO   = INTEGER
CATEGORY    = SYMBOL
```

і декларації **database** для бази даних:

```
database
rule(RNO, CATEGORY, CONDITIONS)
cond(BNO, STRING)
yes(BNO)
no(BNO)
topic(string)
```

Предикат БД **rule** (правило) містить дані про собаку і предикат **cond** (умова) зберігає умови (або атрибути), що характеризують різні породи. Предикати **yes** (так) і **no** (немає) зберігають відповіді користувача. Предикат **topic** (тема) містить дані про типи собак (довго- або короткошерста порода).

Розділ **predicates** має вісім предикатів для інтерфейсу з користувачем:

```
do_expert_job
show_menu
do_consulting
process(integer)
info(CATEGORY)
goes(CATEGORY)
listopt
erase
clear
eval_reply(char)
```

Предикат **do_expert_job** (виконує експертну роботу) є метою програми. Правило **erase** (вилучення) вилучає дані з БД після закінчення циклу розпізнавання – дії. Правило **clear** (очистити) знищує в БД усі відповіді **yes** (так) і **no** (немає). Інші сім предикатів використовують у механізмі висновку.

```
go(HISTORY, CATEGORY)
check(RNO, HISTORY, CONDITIONS)
notes(BNO)
inpo(HISTORY, RNO, BNO, STRING)
do_answer(HISTORY, RNO, STRING, BNO, INTEGER)
```

Ці предикати і правила здійснюють пошук у БД і зберігають трасу значень об'єктів БД і введення користувача для цілей логічного висновку. Правило **check** (перевірка) шукає зразки даних, порівнянні з вхідними даними користувача. Альтернативні правила **notest** зберігають трасу відповідей **yes** (так) і **no** (немає). Предикат **do_answer** (оброби відповідь) додає дані від користувача в динамічну базу даних.

Експертна система, що ґрунтується на логіці, складається з БД, що містить твердження логіки предикатів. Твердження мають одну з двох форм: **rule** або **cond**. БД наведена нижче:

```
topic("dog").
topic("short-haired dog").
topic("long-haired dog").
```

```
rule(1, "dog",              "short-haired dog",    [1]   ).
rule(2, "dog",              "longt-haired dog",    [2]   ).
rule(3, "short-haired dog","English Bulldog ",    [3,5,7]  ).
rule(4, "short-haired dog","Beagle",  [3,6,7]     ).
rule(5, "short-haired dog","Great Dane",    [5,6,7,8]  ).
rule(6, "short-haired dog","American Foxhound",[4,6,7] ).
rule(7, "long-haired dog", "Cocker Spaniel", [3,5,6,7] ).
rule(8, "long-haired dog", "Irish Setter",   [4,6]     ).
rule(9, "long-haired dog", "Collie",         [4,5,7]   ).
rule(9, "long-haired dog", "St. Bernard",    [5,7,8]   ).
cond(1, "short-haired"     ).
cond(2, "long-haired" ).
cond(3, "height under 22 inches" ).
cond(4, "height under 30 inches" ).
cond(5, "low-set tail"     ).
cond(6, "longer ears" ).
cond(7, "good natured personality"    ).
cond(8, "weight over 100 lb"     ).
```

В розділі "Побудова БД" було сказано, що останній об'єкт у твердженні **rule** – список цілих чисел. Список містить номери умов, що характеризують кожну породу собаки в БД. Пропозиції **cond** містять усі можливі характеристики собак.

Тепер необхідно "сконструювати" предикати і правила для механізму висновку. Практично механізм висновку повинен мати початкове правило, таке, як **go**, для прийняття мети користувача й ініціалізації циклу розпізнавання – дії. Механізм переглядає твердження БД **rule** і **cond** для з'ясування існування або відсутності відповідних даних.

Початкове правило викликає правила, наприклад **check** (перевірка), для належного виконання підзадач. Це правило містить шлях номерів правил, номерів умов і класифікаційні об'єкти в БД. Воно намагається порівняти класифіковані об'єкти в термінах номерів умов.

Якщо порівняння відбувається, то цей модуль програми повинен додати в програму зіставлені значення і продовжити процес з новими даними, отриманими від користувача. Якщо зіставлення не відбувається, механізм зупиняє поточний процес і вибирає для порівняння іншу трасу. Пошук і зіставлення тривають доти, доки не вичерпані усі варіанти.

Після закінчення висновку початкове правило за допомогою інтерфейсу передає результати користувачеві.

Нижче наведена повна програма, що реалізує механізм висновку для експертної системи, що базується на логіці.

```
go(HISTORY, Mygoal) :-
rule(RNO,Mygoal,NY,COND),
check(RNO,HISTORY,COND),
go([RNO|HISTORY],NY).
check(RNO,HISTORY,[BNO|REST]) :-
yes(BNO),!,
check(RNO,HISTORY,REST).
check(_,_,[BNO|_]) :- no(BNO),!,fail.
check(RNO,HISTORY,[BNO|REST]) :-
cond(BNO,NCOND),
fronttoken(NCOND,"not",_,COND),
frontchar(_,COND,_,COND),
cond(BNO1,COND),
notest(BNO1),!,
check(RNO,HISTORY,REST).
check(_,_,[BNO|_]) :-
cond(BNO,NCOND),
fronttoken(NCOND,"not",_,COND),
frontchar(_,COND,_,COND),
cond(BNO1,COND),
yes(BNO1),
!,fail.
check(RNO,HISTORY,[BNO|REST]) :-
cond(BNO,TEXT),
inpo(HISTORY,RNO,BNO,TEXT),
check(RNO,HISTORY,REST).
check(_,_,[]).
notest(BNO) :- no(BNO),!.
notest(BNO) :- not(yes(BNO)),!.
do_answer(_,_,_,_,0) :- exit.
do_answer(_,_,_,BNO,1) :-
assert(yes(BNO)),
shiftwindow(1),
write(yes),nl.
do_answer(_,_,_,BNO,2) :-
assert(no(BNO)),
write(no),nl,
```

```
fail.
erase :- retract(_),fail.
erase.
clear :- retract(yes(_)),retract(no(_)),fail,!.
clear.
```

Для описання роботи механізму висновку далі наводиться приклад. Припустимо, що завданням механізму висновку є ідентифікація собаки, чиї характеристики відповідають кокер-спанієлю і містяться в БД за номерами 3, 5, 6, і 7.

Під час виконання програми мету програми забезпечує видання корисної інформації за допомогою інтерфейсу з користувачем. Після цього викликають модуль **do_consulting** (виконуючий консультацію), що у свою чергу викликає правило **go**. Це правило – початкове правило механізму висновку.

```
go(HISTORY, Mygoal) :-
rule(RNO,Mygoal,NY,COND),
check(RNO,HISTORY,COND),
go([RNO|HISTORY],NY).
```

Спочатку введення користувачем слова **dog** приводить до того, що змінна **Mygoal** (моя мета) одержує значення **"dog"** (собака). Застосовується твердження БД **rule(1,"dog","short-haired dog",[1])** і облікова змінна **COND** одержує значення типу список [1]. Далі правило **rule** передає цей параметр правилу **check** (перевірка). Своєю чергою, правило **check** здійснює доступ до правила **cond** БД з параметром **BNO**, що дорівнює 1. Правило **check** передає це значення предикатові **fronttoken**, щоб створити значення **_COND**. Це правило спричиняє невдачу.

Правило **check** повертається до правила **cond** і **COND** містить значення параметра. Потім правило **check** здійснює доступ до значення **"short-haired"** (короткошерстий) і передає його в змінній **TEXT** правилу **inpq**. Правило **inpq** видає на екран текстовий рядок: **Question: - short-haired?** (Запитання: – короткошерстий?)

Користувачеві повідомляється, що він повинен натиснути 1 для ствердної відповіді і 2 для негативної. Правило **inpq** приймає відповідь користувача, в нашому прикладі це 2, і інтерпретує її як негативну.

Процес триває з твердженням **rule** БД, тобто **RNO**, що дорівнює 2. Тепер правило **check** використовує значення **COND**, що дорівнює 2, для поточного правила **rule**, повторює цикл побудови списку значень **COND** і

запитує користувача про додаткове введення. Ґрунтуючись на відповідях користувача, що вводяться, (рис. 10.12), доступ до тверджень **rule** і **cond** відбувається у такій послідовності:

```
rule(1),    cond(1),
rule(2),    cond(2),
rule(7),    cond(3),
rule(7),    cond(5),
rule(7),    cond(6),
rule(7),    cond(7).
```

Наприкінці цього процесу змінна **COND** типу список має значення [3; 5; 6; 7]. Цей список зіставляється зі списком умови у правилі 7. Порівняння зв'язується з класифікованим об'єктом **"Cocker-Spaniel"** (кокер-спанієль) у твердженні **rule**, і, отже механізм висновку знайшов необхідний результат.

Система користувацького інтерфейсу має три частини. Найбільша частина складається здебільшого з правил для організації меню і закриття вікна, коли воно більше не потрібне. Робота другої частини СКІ залежить від вибору користувачем програмної функції. Підправило **process(1)** викликає правило **do_consulting**, яке у своєю чергою викликає **goes(Mygoal)**. Це підправило видає список порід собак і викликає правило **go(Mygoal)**, яке ініціалізує пошук і зіставлення зі зразком. Третя частина **CKI** запитує й отримує відповіді **yes** і **no** від користувача. Вона реалізована в такий спосіб:

```
inpo(HISTORY,RNO,BNO,TEXT) :-
write("Question :-",TEXT," ? "),
makewindow(2,7,7,"Response",10,54,7,20),
write("Type 1 for 'yes' ,"),nl,
write("Type 2 for 'no' : "),nl,
readint(RESPONSE),
clearwindow,
shiftwindow
do_answer( HISTORY,RNO,TEXT,BNO,RESPONSE).
```

Це правило взаємодіє і з користувачем, і з механізмом висновку. Предикати **write** і **readint** використовують для взаємодії з користувачем, а правило **do_answer** взаємодіє з МВ.

Тепер можна з'єднати окремі компоненти разом, щоб сформувати повну експертну систему для вибору собаки, яка базується на логіці. Лістинг 2.5 – це реалізація даного проекту.

Лістинг 2.5

```
/* Програма: Експерт з порід собак Файл:prog1002.pro */
/* Призначення. Демонстрація роботи експертної системи, */
/*  базується  на логіці                                */
/* Зауваження: це система для ідентифікації породи. Система */
/* складається з бази даних (БД), механізму висновку (МВ)*/
/* і системи користувацького інтерфейсу (СКІ).          */
/* База даних розташовується в оперативній пам'яті      */
domains
CONDITIONS = BNO *
HISTORY    = RNO *
RNO, BNO, FNO   = integer
CATEGORY   = symbol
database
/* Предикати бази даних     */
rule(RNO, CATEGORY, CONDITIONS)
cond(BNO, string)
yes(BNO)
no(BNO)
topic(string)
predicates
/* Предикати системи користувацького інтерфейсу  */
do_expert_job
show_menu
do_consulting
process(integer)
info(CATEGORY)
goes(CATEGORY)
listopt
erase
clear
eval_reply(char)
/* Предикати механізму висновку  */
go(HISTORY, CATEGORY)
check(RNO, HISTORY, CONDITIONS)
notes(BNO)
inpo(HISTORY, RNO, BNO, STRING)
do_answer(HISTORY, RNO, STRING, BNO, INTEGER)
goal
do_expert_job
clauses
/* База даних (БД)    */
topic("dog").
topic("short-haired dog").
topic("long-haired dog").
```

```
rule(1,    "dog",              "short-haired dog",[1]   ).
rule(2,    "dog",              "longt-haired dog",[2]   ).
rule(3,    "short-haired dog","English Bulldog ",[3,5,7] ).
rule(4,    "short-haired dog","Beagle",     [3,6,7]    ).
rule(5,    "short-haired dog","Great Dane", [5,6,7,8]  ).
rule(6,    "short-haired dog","American Foxhound",[4,6,7] ).
rule(7,    "long-haired dog", "Cocker Spaniel", [3,5,6,7] ).
rule(8,    "long-haired dog", "Irish Setter",     [4,6] ).
rule(9,    "long-haired dog", "Collie",           [4,5,7] ).
rule(9,    "long-haired dog", "St. Bernard",      [5,7,8] ).
cond(1,    "short-haired"  ).
cond(2,    "long-haired"   ).
cond(3,    "height under 22 inches"   ).
cond(4,    "height under 30 inches"   ).
cond(5,    "low-set tail"  ).
cond(6,    "longer ears"   ).
cond(7,    "good natured personality" ).
cond(8,    "weight over 100 lb"  ).
/* Система користувацького інтерфейсу */
do_expert_job :-
makewindow(1,7,7," DOG EXPERT SYSTEM ",0,0,25,80),
show_menu,
nl,write(" Press space bar. "),
readchar(_),
exit.
show_menu :-
write("    "),nl,
write(" * * * * * * * * * * * * * * * *  *     "),nl,
write(" *          DOG EXPERT             *     "),nl,
write(" *                                 *     "),nl,
write(" *    1. Consultation              *     "),nl,
write(" *                                 *     "),nl,
write(" *                                 *     "),nl,
write(" *    2. Exit the system           *     "),nl,
write(" *                                 *     "),nl,
write(" * * * * * * * * * * * * * * * *  *     "),nl,
write("    "),nl,
write("Please enter your choice: 1 or 2 : "),nl,
readint(Choice),
process (Choice).
process(1) :-
do_consulting.
process(2) :-
removewindow,
exit.
do_consulting :-
goes(Mygoal),
```

```
go([],Mygoal),
!.
do_consulting :-
nl, write(" Sorry I can't help yuo."),
clear.
do_consulting.
goes(Mygoal) :-
clear,
clearwindow,
nl,nl,
write("    "),nl,
write("   WELCOME TO THE DOG EXPERT SYSTEM  "),nl,
write("    "),nl,
write("This is a dog identification system. "),nl,
write("To begin the process of choosing a   "),nl,
write("dog, please type in 'dog'. If you    "),nl,
write("wish to see the dog types, please    "),nl,
write("type in a question mark (?).    "),nl,
write("    "),nl,
readin(Mygoal),
info(Mygoal),!.
info("?") :-
clearwindow,
write("Reply from the KBS."),nl,
listopt,
nl,write("Please any key. "),
readchar(_),
clearwindow,
exit.
info(X) :-
X >< "?".
listopt :-
write("The dog types are : "),nl,nl,
topic(Dog),
write("    ",Dog),nl,
fail.
listopt.
inpo(HISTORY,RNO,BNO,TEXT) :-
write("Question :- ",TEXT," ? "),
makewindow(2,7,7,"Response",10,54,7,20),
write("Type 1 for 'yes' ,"),nl,
write("Type 2 for 'no' : "),nl,
readint(RESPONSE),
clearwindow,
shiftwindow(1),
do_answer(HISTORY,RNO,TEXT,BNO,RESPONSE).
eval_reply('y') :-
```

```
write("I hope you have found this helpful !").
eval_reply('n') :-
write(" I am sorry I can't help you !").
go(_,Mygoal) :-
not(rule(_,Mygoal,_,_)),!,
nl,write(" The dog you have indicated is a(n) ",
Mygoal,".").nl,
write("Is a dog you would like to have (y/n) ?"),
nl,readchar(R),
eval_reply(R).
/*   Механізм висновку     */
go(HISTORY, Mygoal) :-
rule(RNO,Mygoal,NY,COND),
check(RNO,HISTORY,COND),
go([RNO|HISTORY],NY).
check(RNO,HISTORY,[BNO|REST]) :-
yes(BNO),!,
check(RNO,HISTORY,REST).
check(_,_,[BNO|_]) :- no(BNO),!,fail.
check(RNO,HISTORY,[BNO|REST]) :-
cond(BNO,NCOND),
fronttoken(NCOND,"not",_,COND),
frontchar(_,COND,_,COND),
cond(BNO1,COND),
notest(BNO1),!,
check(RNO,HISTORY,REST).
check(_,_,[BNO|_]) :-
cond(BNO,NCOND),
fronttoken(NCOND,"not",_,COND),
frontchar(_,COND,_,COND),
cond(BNO1,COND),
yes(BNO1),
!,fail.
check(RNO,HISTORY,[BNO|REST]) :-
cond(BNO,TEXT),
inpo(HISTORY,RNO,BNO,TEXT),
check(RNO,HISTORY,REST).
check(_,_,[]).
notest(BNO) :- no(BNO),!.
notest(BNO) :- not(yes(BNO)),!.
do_answer(_,_,_,0) :- exit.
do_answer(_,_,BNO,1) :-
assert(yes(BNO)),
shiftwindow(1),
write(yes),nl.
do_answer(_,_,_,BNO,2) :-
assert(no(BNO)),
```

```
write(no),nl,
fail.
erase :- retract(_),
fail.
erase.
clear :- retract(yes(_)),
retract(no(_)),
fail,!.
clear. /*        Кінець програми        */
```

Ця програма видає початкове меню, пропонуючи користувачеві вибір між **consultation** (консультацією) і **exit from the system** (вихід із системи). Якщо користувач вибирає консультацію, то між користувачем і системою відбувається діалог. Потім користувачеві повідомляють результат. Результатом є або вибрана порода, або повідомлення **Sorry I can't help you** (Вибачте, я не можу допомогти вам).

На рис. 2.14, *а* показано діалог під час консультації. Зверніть увагу, цей діалог приводить до ствердного результату. Тобто БД містить інформацію про породу собаки, що задовольняє специфікації користувача.

DOG EXPERT SYSTEM

WELCOM TO THE DOG EXPERT SYSTEM
This is a dog identification system
To begin the process of choosing a
Dog, please type in ‘dog’. If yuo
Wish to see the dog types, please
Type in a question mark (?)

Dog

Question: - has it, short-haired? no
Question: - has it, long-haired? yes
Question: - has it, height under 22 inches? yes
Question: - has it, low set-tail ? yes
Question: - has it, longer ears? yes
Question: - has it, good natured personality?yes
The dog you have indicated is a(n) cocker –spaniel,
Is this a dog you wold like to have (y/n) ?

Рис. 2.14, *а*. Діалог експертної системи, що базується на логіці, для вибору породи собаки

2.6. Розширена експертна система

2.6.1. Алгоритм побудови експертної системи медичної діагностики

Дві експертні системи, розглянуті в попередніх розділах, доволі прості і легко реалізуються. Важливою рисою обох систем є можливість їхнього розширення для підтримки великих БД. Механізми висновку в цих експертних системах структуровані так, щоб їх можна було легко розширювати, додаючи специфічніші твердження. Ці твердження можуть формувати великі і неоднорідні БД.

Медична діагностична експертна система може проілюструвати розширення системи у описаний спосіб. Призначення цієї системи, що базується на логіці – ідентифікація ймовірної хвороби. Користувач подає інформацію про симптоми у відповідь на запитання експертної системи.

Розгляд проекту

Проект медичної діагностичної експертної системи аналогічний експертній системі, що базується на логіці, для вибору породи собаки. Додатково в неї входять модулі для завантаження і збереження БД.

Діаграма потоків даних для цієї системи наведена на рис. 2.15, *а*. Зверніть увагу, що діаграма подібна на діаграму потоків даних, наведену на рис. 2.15, тому деталі потоків даних на рис. 2.15, *а* не специфіковані. Додано модулі process(1) для завантаження файла БД у пам'ять і **`process(3)`** для збереження БД на диску. (Зверніть увагу на лінії потоків даних між БД у пам'яті і файлом БД на диску).

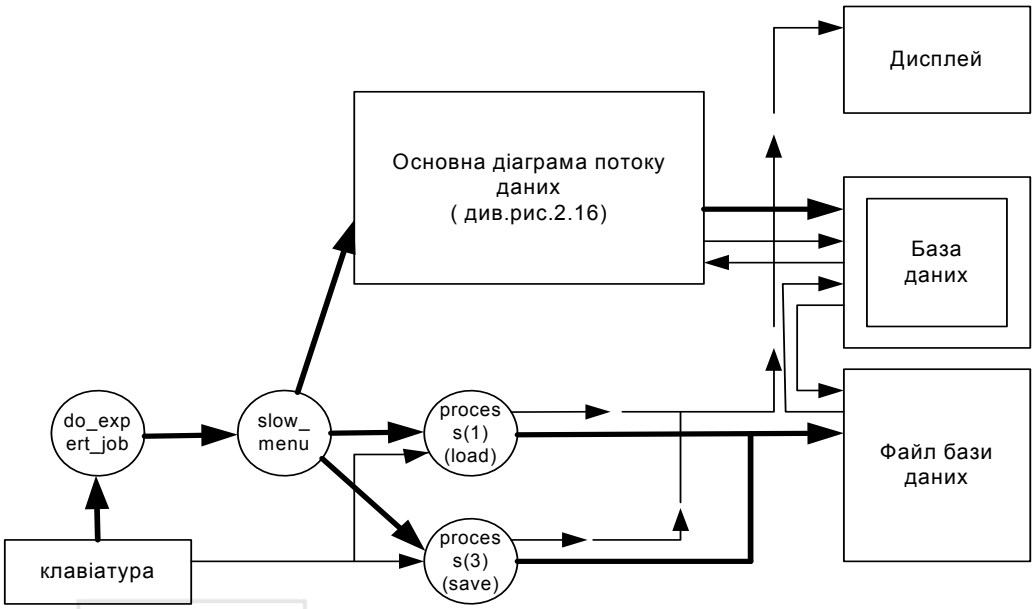


Рис 2.15, *а*. БД для медичної діагностичної експертної системи

Структурна схема, отримана з діаграми потоків даних, показана на рис. 2.16, *а*. Ця структурна схема аналогічна до схеми, наведеної на рис. 2.16, для системи вибору породи собак, що базується на логіці. Однак зверніть увагу на два додаткові модулі **process(1)** і **process(3)**, що завантажують і зберігають БД.

БД має інформацію про 15 хвороб; ця інформація зберігається в 15 твердженнях **rule** БД. Список цілих чисел, зв'язаний з кожним твердженням **rule**, характеризує хворобу. Твердження **rule** і **cond** зв'язані за допомогою списків наприкінці кожного правила для медичної експертної проблеми. Наприклад, правило:

```
rule(11,"illness","social anxiety",[12,15,16,17,18,13])
```

зв'язано з такими симптомами (умовами):

```
cond(12,"feel anxious most of time")
cond(15,"anxious since giving up tobacco,alcohol, drugs") cond(16,"recently had a major upset in file") cond(17,"loss weight or eyes bulding") cond(18,"have sex life problem")
cond(13,"feel anxious in meeting, particles, interviews")
```

Твердження **`topic("illness")`** надає користувачеві вибір БД. У нас вибір **– `"illness".`** Якщо треба, наприклад, створити БД для інфекційних захворювань, треба додати твердження **`topic("infectious diseases")`** (інфекційні хвороби). Потім потрібно спроектувати і реалізувати **`rule`** і **`cond`** твердження для побудови БД про захворювання.

Рис. 2.16, *а*. Структурна діаграма для медичної експертної системи

2.6.2. Програмна реалізація експертної системи медичної діагностики

Лістинг програми, що реалізує систему для вибору собаки, яка базується на правилах, може бути використаний як шаблон для програми експертної системи медичної діагностики. Структури двох програм аналогічні. Якщо необхідно, можна одержати медичну діагностичну експертну систему. Повна програма, показана на лістингу 2.6, – реалізація такої модифікації.

Лістинг 2.6

```
/* Програма: Медична експертна система Файл: prog1003.pro */
/*                                                         */
/* Призначення. Демонстрація роботи експертної системи, що
/* базується на диску */
/* Зауваження:  Це медична експертна система.  Система */
/* складається з бази даних (БД), механізму висновку (МВ)*/
/* і системи користувацького інтерфейсу (СПИ).    */
/* База даних завантажується з диска  */
domains
CONDITIONS = BNO *
HISTORY    = RNO *
RNO, BNO, FNO   = integer
CATEGORY   = symbol
database
/* Предикати бази даних     */
rule(RNO, CATEGORY, CONDITIONS)
cond(BNO, string)
yes(BNO)
no(BNO)
topic(string)
predicates
/* Предикати системи користувацького інтерфейсу  */
do_expert_job
show_menu
do_consulting
process(integer)
listopt
evalans(char)
info(CATEGORY)
goes(CATEGORY)
/* Предикати механізму висновку  */
go(HISTORY, CATEGORY)
check(RNO, HISTORY, CONDITIONS)
notes(BNO)
inpo(HISTORY, RNO, BNO, string)
do_answer(HISTORY, RNO, string, BNO, integer)
erase
clear
goal
do_expert_job
```

```
clauses
/* Система користувацького інтерфейсу (частина 1)*/
do_expert_job :-
makewindow(1,7,7,"MEDICAL EXPERT SYSTEM ",0,0,25,80),
show_menu,
nl,write(" Press space bar. "),
readchar(_),
exit.
show_menu :-
write("    "),nl,
write(" * * * * * * * * * * * * * * * * *  *     "),nl,
write(" *      MEDICAL EXPERT              *     "),nl,
write(" *                                  *     "),nl,
write(" *    1. Load KBS                   *     "),nl,
write(" *    2. Consultation               *     "),nl,
write(" *    3. Save KBS                   *     "),nl,
write(" *    4. Exit the system            *     "),nl,
write(" *                                  *     "),nl,
write(" * * * * * * * * * * * * * * * * *  *     "),nl,
write("    "),nl,
write("Please enter your choice: 1,2,3 or 4 :    "),nl,
readint(Choice),
process (Choice).
process(0).
process(1) :-
consult("illness.dba").
process(2) :-
do_consulting.
process(3) :-
save("ILLNESS.DBA").
process(4) :-
exit.
do_consulting :-
goes(Mygoal),nl,nl,
go([],Mygoal),
!.
do_consulting :-
nl, write(" Sorry I can't determine that one"),
nl,write(" Please see a physician.    "),nl,
clear.
goes(Mygoal) :-
```

```
clear,
clearwindow,
nl,nl,
write("    "),nl,
write(" WELCOME TO THE MEDICAL EXPERT SYSTEM   "),nl,
write("    "),nl,
write("This is an illnes identification system."),nl,
write("To start the consultation process,   "),nl,
write("please type in 'illness'. "),nl,
readin(Mygoal),
info(Mygoal),!.
go(_,Mygoal) :-
not(rule(_,Mygoal,_,__,!,nl,
write("    I think it is ",MYgoal,"."),nl,nl,
write("    Is my diagnosis right (y/n) ?"),nl,
readchar(Answer),
evalans(Answer).
/*  МЕХАНІЗМ ВИСНОВКУ */
go(HISTORY,Mygoal) :-
rule(RNO,Mygoal,NY,COND),
check(RNO,HISTORY,COND),
go([RNO|HISTORY],NY).
check(RNO,HISTORY,[BNO|REST]) :-
yes(BNO),!,
check(RNO,HISTORY,REST).
check(_,_,[BNO|_]) :- no(BNO),!,fail.
check(RNO,HISTORY,[BNO|REST]) :-
cond(BNO,NCOND),
fronttoken(NCOND,"not",_,COND),
frontchar(_,COND,_,COND),
cond(BNO1,COND),
notest(BNO1),!,
check(RNO,HISTORY,REST).
check(_,_,[BNO|_]) :-
cond(BNO,NCOND),
fronttoken(NCOND,"not",_,COND),
frontchar(_,COND,_,COND),
cond(BNO1,COND),
yes(BNO1),
!,fail.
check(RNO,HISTORY,[BNO|REST]) :-
```

```
cond(BNO,TEXT),
inpo(HISTORY,RNO,BNO,TEXT),
check(RNO,HISTORY,REST).
check(_,_,[]).
notest(BNO) :- no(BNO),!.
notest(BNO) :- not(yes(BNO)),!.
do_answer(_,_,_,0) :- exit.
do_answer(_,_,BNO,1) :-
assert(yes(BNO)),
shiftwindow(1),
write(yes),nl.
do_answer(_,_,_,BNO,2) :-
assert(no(BNO)),
write(no),nl,
fail.
erase :- retract(_),fail.
erase.
clear :- retract(yes(_)),retract(no(_)),fail,!.
clear.
/* СИСТЕМА КОРИСТУВАЦЬКОГО ІНТЕРФЕЙСУ (частина 2)  */
inpo(HISTORY,RNO,BNO,TEXT) :-
write("Question :-",TEXT," ? "),
makewindow(2,7,7,"Response",10,54,7,20),
write("Type 1 for 'yes' ,"),nl,
write("Type 2 for 'no' : "),nl,
readint(RESPONSE),
clearwindow,
shiftwindow(1),
do_answer(HISTORY,RNO,TEXT,BNO,RESPONSE).
info("?") :-
clearwindow,
write("Reply from the KBS."),nl,
listopt,
nl,write("Please any key. "),
readchar(_),
clearwindow,
show_menu.
info(X) :-
X >< "?".
listopt :-
write("The illnesses are : "),nl,nl,
```

```
topic(Ins),
write("    ",Ins,"  "),nl,
fail.
listopt.
evalans('y') :-
write(" I am glad I can help you !"),nl,nl,
write(" Press the space bar."),
readchar(_),
clearwindow,
show_menu.
evalans('n') :-
write(" I am sorry I can't help you !"),nl,nl,
write(" Please press space bar ."),
readchar(_),
clearwindow,
show_menu.
/*          КІНЕЦЬ ПРОГРАМИ              */
```


Відзначимо, що БД не наведена на лістингу. Вона зберігається у файлі на диску. Цей файл створюється за допомогою редактора Турбо-Прологу. Предикати роботи з файлами consult і save використовують для завантаження файла БД і запису його на диск. Меню в програмі для користувача забезпечує зручне завантаження і збереження модулів БД.

Під час виконання програми користувачеві надається меню програмних функцій. Потім видається консультація на основі запитань і відповідей користувача. Відповіді дають експертній системі інформацію для зіставлення за зразком. Модулі користувацького інтерфейсу забезпечують відповідну графіку і зручний для користувача діалог природною мовою. Коли закінчується цикл розпізнавання – дія експертної системи, система видає кінцеве повідомлення: або ймовірний діагноз, або, пораду звернутися до лікаря. Консультація допомагає ідентифікувати хворобу.

Ядро медичної діагностичної експертної системи, яка ґрунтується на логіці, – механізм висновку. Робота механізму висновку аналогічна до роботи механізму висновку в системі, яка базується на логіці експертної системи для вибору собаки.

Відправною точкою роботи механізму висновку є отримані дані користувача illness. Перше питання, що задається користувачеві, випливає з першого твердження cond.

На екрані з'являється: Question: - continually on edge?

Питання, що залишилися, базуються на твердженнях cond з другого номера по восьмий. Відповіді на питання 5 і 8 негативні, тому змінна COND типу список має значення [1, 2, 3, 4, 5, 6, 7]. Цей список порівнюється зі списком умов правила 6. Це зіставляння зв'язується з класифікованим об'єктом "hypothyroidism" у твердженні rule, і, отже, механізм висновку знайшов, таким чином, рішення – збільшення щитовидної залози.

2.6.3. Контрольні запитання і вправи

1. Дати означення експертної системи.
2. Основні принципи побудови експертних систем.
3. Навести загальну структуру експертної системи.
4. Класифікація способів подання знань в експертних системах.
5. Навести загальну структурну схему експертної системи.
6. Суть методу висновку у разі його застосування до побудови експертних систем в середовищі Турбо-Прологу.
7. Порівняльна характеристика експертних систем, що ґрунтуються на правилах та на логіці.
8. Характеристика вбудованих предикатів Турбо-Прологу для поновлення бази

знань експертної системи.

9. Записати фрагмент програми для підтримки усного діалогу мовою у вигляді правил та мовою логіки.
10. У створену базу діагностики захворювань ввести динамічним способом діагноз двох нових захворювань і переконатися у правильності встановлення діагнозу системою.

2.7. Діалог з комп’ютером на мові логіки

2.7.1. Підходи до спілкування з комп'ютером на природній мові

Говорячи про природну мову, можна мати на увазі будь-яку мову, за допомогою якої люди спілкуються між собою: англійську, французьку, японську, російську. Усі ці мови мають структури і правила, відповідно до яких люди конструюють зі слів фрази, фрагменти речення, цілі речення. Коли говорять, що хтось "спілкується природною мовою", то мають на увазі не тільки розуміння ним речень, але і його здатність формулювати речення самостійно.

Наука про спілкування з комп’ютером природною мовою така сама давня, як і сама проблема штучного інтелекту. Лісп, перша мова штучного інтелекту, була створена для цілей обробки символів, слів, списків, а також складніших облікових структур. Лісп доволі часто використовується в розробках з проблеми спілкування природною мовою. Створення другої мови для розв’язання проблем штучного інтелекту Прологу стало можливе завдяки дослідженням в області природних мов, мов програмування і комп’ютерного перекладу.

Вченими було створено низку систем, що розуміють природну мову. У 1979 році Ларрі Харріс з фірми AI Corporation розробив систему INTELLECT; вона здійснювала взаємодію з базами даних і інформаційними системами в галузі фінансів, маркетингу, виробництва і керування. Система SCRIPTS, створена фірмою Cognitive Systems Instruments, є іншим прикладом подібних систем. Програма NaturalLink, спроектована під керівництвом Харрі Теннанта в Texas Instruments, служить інтерфейсом з мікрокомп'ютерами. Комп’ютерна система ELIZA, природно-мовний пакет, що імітує роботу психолога, реагує на ключові слова, що трапляються в реченнях пацієнта, і дає запрограмовані рекомендації, ґрунтуючись на них. Якийсь час ELIZA вважалася найсерйознішою програмою у даній галузі. Автор системи ELIZA Джозеф Вайценбаум стверджував навіть, що його дітище може виконувати роль консультанта-лікаря.

Метою всіх робіт, що виконуються в галузі штучного інтелекту, незалежно від розходжень в підходах до вирішення проблеми, є створення програм для сприйняття природної мови, яка б не дуже відрізнялася від сприйняття мови людиною. Створювані програми, як правило, працюють з

текстовим матеріалом, який або вводиться з клавіатури, або зчитується спеціальним сканувальним пристроєм. Звукові розпізнавання і синтез мови, як бачимо, відіграватимуть все важливішу роль для розв'язання проблеми спілкування з комп'ютером на природній мові.

Одним з підходів до спілкування з комп'ютером на природній мові є метод ключових слів. Аналіз ключових слів – метод аналізу речень на предмет наявності ключових слів, які може розпізнавати комп'ютер. Ключові слова стають значеннями об'єктів предикатів. При такому підході важлива не граматична структура речення, оскільки програма не аналізує зв'язків між словами і реагує однаково на різні варіанти вхідного тексту, а лише наявність ключових слів.

Добрі результати, як було показано, дає контекстно-вільний аналіз (КВ-аналіз). Побудова будь-якої природної мови підпорядковується деякому набору правил, який називають граматикою. Правила граматики диктують послідовність проходження символів і рядків символів, що утворюють дозволені мовою речення. У КВ-граматиках фрази класифікують залежно лише від їхньої внутрішньої структури. У контекстно-залежних граматиках, навпаки, аналіз конструкції залежить від контексту, в якому міститься конструкція.

Наприклад, припустимим є речення

Мері читає книжки.

Воно відповідає вже знайомій суб'єктно-вербально-об'єктній структурі. Значення цього речення можна зрозуміти, не звертаючись ні до яких інших речень. Таким чином, воно піддається аналізу як окрема незалежна одиниця.

Природна мова, однак, містить багато прикладів контекстно-залежних речень. Розглянемо речення

Джон розмовляв із Джо, перед тим як він пішов на сніданок.

У даному реченні займенник "він" неоднозначний. Незрозуміло, займенник "він" заміщає "Джона" чи "Джо". Хоча природні мови зовсім не страждають від такої неоднозначності, можна створити мову, усі речення якої мають сенс, зрозумілий з тексту саме цього речення, поза залежністю від контексту. Саме такі речення використовують в підході, орієнтованому на КВ-граматики.

Пролог є зручним засобом використання підходу, зв'язаним із застосуванням КВ-граматик, до обробки конструкцій природної мови. КВ-правила можуть бути записані у вигляді тверджень Прологу. Задача

розпізнавання речень зводиться до задачі доказу набору тверджень, що задають граматичну структуру мови.

Під час синтаксичного аналізу елементи речення аналізують відповідно до граматичних правил. Такий аналіз вимагає розділення речення на складові частини: ця операція виконується за допомогою граматичного розбору речення.

Техніка граматичного розбору передбачає розширений розбір мережі переходів, розбір конструкцій знизу-вгору і зверху-вниз. Розширений розбір мережі переходів полягає в розділенні речення на все менші і менші частини доти, доки воно не буде розібрано цілком. При розборі нагору аналізові піддаються всі слова, починаючи із самого лівого і кінчаючи самим правим. Діючи, таким чином, можна вивести будь-яку можливу синтаксичну структуру. При аналізі зверху-униз речення аналізується, виходячи з припущення щодо його структури. Техніка аналізу зверху-униз використовується в одній із програм даного параграфу.

Прагматичний аналіз має справу з вивченням того сенсу, що люди вкладають у реченні, а не того, який виражають слова цього речення самі по собі. Так, наприклад, відповіддю на питання "Чому Джон сьогодні такий тихий?" є все-таки не "Сьогодні він ще не говорив". За цим питанням стоїть зовсім інший зміст, який звучить приблизно так: "Чим так сьогодні стурбований Джон?". Здатність дати відповідь на це питання, що представляє дійсний інтерес, а не на його буквальне формулювання, вимагає наявності величезного обсягу інформації про особливості людського сприйняття, родинних відношень і почуттів людей. Такий підхід до проблеми спілкування природною мовою є одним з найбільш складних, тому, тут поки важко досягти яких-небудь значних результатів.

Аналіз ключових слів дозволяє програмі, що відповідає за інтерфейс між машиною і людиною, правильно реагувати на досить широкий клас варіацій запитів до системи. Розглянемо три таких запити:

Розкажіть мені про Джорджа Вашингтона.

Дайте мені інформацію про Джорджа Вашингтона.

Хто такий Джордж Вашингтон?

Кожне з цих речень могло б виступити в ролі команди бази даних, що запитує відомості про цю видатну людину. Хоча речення розрізняються за структурою, можна сказати, що функціонально вони еквівалентні. Кожне з них містить ім'я Джордж Вашингтон. Програма може вичленувати це ключове словосполучення і використовувати його як значення об'єкта

твердження БД. Діючи таким чином, можна спробувати сконструювати нескладний природно-мовний інтерфейс із БД.

Визначаючи реакцію програми, варто приділити увагу словам, що знаходяться на початку кожної фрази: розкажіть (tell), дайте (show), хто (who). Перші два з них можна трактувати як запит на всю інформацію, яка є в наявності про Джорджа Вашингтона. Третє, навпаки, може вказувати на те, що користувачеві потрібна лише частина цієї інформації. Як бачимо, розгляд цих ключових слів дозволяє визначити кількість і вид виданої інформації.

При розробці простого природно-мовного інтерфейсу можна прийняти наступні припущення щодо командних речень, що вводяться:

1. Структура команди не впливає на вибір використовуваних у ній слів.

2. Для визначення кількості і характеру запитуваної користувачем інформації потрібен аналіз лише декількох ключових слів.

Обидва припущення можна пояснити на прикладі порівняння двох таких командних речень:

```
Опишіть Джорджа Вашингтона.
Розкажіть мені все, що відомо про життя і діяльність Джорджа Вашингтона.
```

Друга пропозиція містить багато сторонніх слів. Такі слова ігноруються, відповідь системи від них не залежить.

Головною задачею при розробці інтерфейсу, що використовує словник ключових слів, є аналіз речень, що вводяться, з метою виділення ключів. Ця задача легко вирішується в Турбо-Пролозі. Процедура складається з трьох кроків:

1. Введення команди як рядка символів.

2. Перетворення цього рядка в список слів.

3. Ідентифікація ключових слів.

Перший крок легко запровадимо в життя через те, що рядок є базисним типом даних Турбо-Пролога. Другий крок вимагає написання правила, що перетворить рядок у список. Така операція може бути прооблена за допомогою вбудованого предиката Турбо-Пролога **fronttoken** (див. додаток А). Третій крок може бути реалізований двома способами. Один з них полягає в заданні всіх ключових слів (усіх їхніх варіантів, якщо такі існують) у твердженнях бази даних. Тоді внутрішні уніфікаційні процедури Турбо-Пролога зможуть зіставити елементи речення зі значеннями, взятими з БД. Якщо зіставлення вдається, відбувається перехід до наступної стадії. Інший спосіб полягає в тому, що ключові слова визначаються позицією, що вони

займають у реченні. Цей спосіб цілком ґрунтується на частоті застосування визначених граматичних конструкцій. Так, наприклад, ключовими можуть вважатися перше й останнє слова речення, як у питанні "Де знаходиться Бостон?" Саме ці слова уніфікаційної процедури намагаються порівняти з наявними в базі. Так само, як і в першому випадку, успішне зіставлення дає програмі змогу почати виконання наступних кроків.

Цей вид аналізу ключових слів є прикладом дуже простого граматичного розбору команд користувача. Якщо кількість ключових слів буде доволі великим, то з БД можна буде отримувати докладнішу і точнішу інформацію.

2.7.2. Алгоритми та програми створення списків та ідентифікації ключових слів

2.7.2.1. Програма створення списків

Ознайомившись з основами методу аналізу ключових слів, можна спробувати спроектувати і написати мовою Турбо-Пролог просту програму, що аналізує слова вхідної пропозиції. Ця програма буде сприймати командні пропозиції, перетворювати ці пропозиції в список слів і виводити цей список на екрані. Вона також буде виділяти зі списку останнє слово, а потім видавати це слово на екран.

Введене користувачем речення привсвоюється змінній **Sentence**. Перетворення рядка в список здійснюється правилом **convers**, що було приведено в 2.6. Звертання до **convers** має вигляд

```
convers(Sentence,list)
```

а саме правило виглядає як

```
convers(Str,[Head|Tail]):-
fronttoken(Str,Head,Str1),!,
convers(Str1,Tail).
convers(_,[ ]).
```

У разі успішного виконання **convers** змінна **List** міститиме список слів речення. Кожен елемент списку – слово.

Далі за допомогою нижченаведеного правила здійснюється ідентифікація останнього елемента списку

```
last_element([Head],Last_element):-
Last_element = Head.
last_element([_|Tail],Last_element):-
last_element(Tail,Last_element).
```

Перший варіант цього рекурсивного правила присвоює змінній **Last_element** останній елемент списку. Другий варіант здійснює пересування в напрямку кінця списку.

Програма "Створення списку" (лістинг 2.8) реалізує метод аналізу ключових слів. Вона може послужити базисом для створення інших, детальніше розроблених практичних програм. У ній використовують засоби Турбо-Прологу для роботи з вікнами. Застосоване також правило **print_list**, яке роздруковує список.

Лістинг 2.8

```
/* Програма: Створення списку. Файл: PROG1101.PRO */
/* Призначення: Перетворення вхідного речення */
/*                  у список слів і витяг з нього */
/*                  останнього слова. */
domains
str_list = symbol * str  = string
predicates
convers(str,str_list)
print_list(str_list)
do_convert_and_print
find_last_word(str_list)
last_element(str_list,symbol)
goal
do_convert_and_print.
clauses
/* правило, що перетворює вхідний рядок у список слів */
convers(Str,[Head|Tail]):-
fronttoken(Str,Head,Str1),!,
convers(Str1,Tail).
convers(_,[ ]).
/* правило для роздруку списку */
print_list([ ]).
```

```
print_list([Head|Tail]):-
write("                          ",Head), nl,
print_list(Tail).
/* правило для пошуку останнього елемента списку */
last_element([Head],Last_element):-
Last_element = Head.
last_element([_|Tail],Last_element):-
last_element(Tail,Last_element).
/* правило для пошуку і роздруку останнього елемента
списку */
find_last_word(List) :-
last_element(List,LName),
nl, write(" Here is the last word : ",Lname), nl.
/* правило для цільового затвердження */
do_convert_and_print :-
makewindow(1,7,7,"",0,0,25,80),
makewindow(2,7,7," A List Maker ",1,10,23,40),
nl, write(" PLEASE TYPE IN A SENTENCE."),
cursor(3,6), readln(Sentence),
nl, write(" THE INPUT SENTENCE IS "), nl, nl,
write("                    ",Sentence), nl,nl,
write(" THE OUTPUT LIST OF WORDS IS "), nl, nl,
convers(Sentence,List),
print_list(List),
find_last_word(List),
nl, write(" PRESS THE SPACE BAR."),
readchar(_),
clearwindow,
exit.
/*****   кінець програми   *****/
```

Уявлення про характер діалогу з цією програмою можна одержати з рис. 2.17. На вхід програми тут подане речення:

a man loves a woman

```
┌──────── A List Maker ────────┐
│ PLEASE TYPE IN A SENTENCE.   │
│     a man loves a woman      │
│ THE INPUT SENTENCE IS        │
│     a man loves a woman      │
│ THE OUTPUT LIST OF WORDS IS  │
│     a                        │
│     man                      │
│     loves                    │
│     a                        │
│     woman                    │
│ THE OUTPUT LIST OF WORDS IS  │
│ Here is a last word: woman   │
│ PRESS THE SPACE BAR.         │
└──────────────────────────────┘
```

Рис. 2.17. Діалог з програмою "Створення списку"

Зверніть увагу на відсутність наприкінці речення крапки '.'. Якби крапка стояла, то вона трактувалася б програмою як окремий атом; і предикат **last_element** присвоїв би крапку змінній **Last_element**. Отже, крапка повинна бути випущена, тому що вона не може зробити ніякого внеску в процес розпізнавання наказу.

2.7.2.2. Програма ідентифікації ключових слів

Як вже було сказано в попередньому розділі, інтерфейсна система, ідентифікуючи слова, може мати потребу в доступі до БД. Покажемо, як створити мовою Турбо-Пролог програму, що порівнює введений користувачем текст із даними, що зберігаються в базі.

Запит до БД у формі команди складається зі слів, деякі з яких повинні бути розпізнані як ключові. Для цього рядок, що містить текст команди, розділяють на слова, що містяться в списку. Кожне слово списку використовується під час спроби порівняти його зі значеннями об'єктів визначених тверджень бази даних. Якщо порівняння вдається, то ці значення присвоюються змінним, які використані для виклику правил. Описана техніка демонструється на прикладі програми "Ключові слова".

Припустимо, що необхідно створити інтерфейс до "Бази даних гри у

футбол", розробленої в розд. 3. Потрібно виділити прізвища гравців так, щоб можна було давати відповіді на запити про них. У такому разі програма повинна містити правила, що дають змогу отримувати повне ім'я гравця з твердження БД dplayer і вилучати з цього імені (що зберігається як один-єдиний елемент даних) прізвище.

Для здійснення другої з перерахованих операцій програма повинна перетворювати повне ім'я гравця в список, що містить окремо ім'я і прізвище гравця, а потім шукати в цьому списку останній елемент. Найістотніша частина правила, яке реалізує цю операцію, виглядає як

```
do_trick :-
dplayer(P_name,T_name,_,_,_,_,_,_),
D_name = P_name,
write(" ",D_name," ",T_name),nl,
convers(D_name,D_list),
last_element(D_list,L_name),
write(" Last name = ",L_name),nl,nl,
fail.
```


Повне ім'я гравця привласнюється змінній **D_name**. Значення змінної обробляється правилом **convers**, що створює список, який складається з імені і прізвища гравця. Правило **Last_element** привласнює значення останнього елемента списку змінній **L_name**.

```
──────── FootBall Database Query ────────┐
│  Dan Marino  Miami Dolphins             │
│  Last name = Marino                     │
│  Richart Dent  Chicago Bears            │
│  Last name = Dent                       │
│  Bernie Kosar  Cleveland Brown          │
│  Last name = Kosar                      │
│  Doug Cosbie  Dallas Cowboys            │
│  Last name = Cosbie                     │
│  Mark Malone  Pittsburgh                │
│  Last name = Malone                     │
│no more names                            │
│Finished                                 │
└─────────────────────────────────────────┘
```

Рис. 2.18. Результат роботи програми “Ключові слова”

Програма "Ключові слова" наведена у лістингу 2.9. Програма використовує засоби Турбо-Прологу для роботи з вікнами, що дає змогу подати у наочнішій формі функції програми, а також роздрукувати її відповіді. Результат роботи наведено на рис. 2.18. Відзначимо, що повне ім'я гравців, їхні прізвища та назви команд можна побачити на екрані у спеціальному вікні.

Лістинг 2.9

```
/* Програма: Ключові слова. Файл: PROG1102.PRO */
/* Призначення: Одержання списку об'єктів БД.       */
domains
str_list = string *
p_name, t_name, pos, height, college = string
p_number, weight, nfl_exp                = integer
database
dplayer(p_name, t_name, p_number, pos,
height, weight, nfl_exp, college)
predicates
do_trick
assert_database
clear_database
player(p_name, t_name, p_number, pos,
height,        weight,        nfl_exp,        college)
convers(string,str_list) last_element(str_list,string)
goal
assert_database,
makewindow(1,7,7,"",0,0,25,80),
makewindow(2,7,7," FootBall Database Query ",
2,10,22,50),  nl,nl,  do_trick,  write("Finished"),nl,
readchar(_), clear_database.
clauses
assert_database :-
player(P_name,T_name,P_number,Pos,Ht,Wt,Exp,
College),
assertz( dplayer(P_name,T_name,P_number,Pos,Ht,
```

```
Wt,Exp,College) ), fail.
assert_database :- !.
clear_database :-
retract( dplayer(_,_,_,_,_,_,_,_) ),
fail.
clear_database :- !.
do_trick :-
dplayer(P_name,T_name,_,_,_,_,_,_),
D_name = P_name,
write(" ",D_name," ",T_name),nl,
convers(D_name,D_list),
last_element(D_list,L_name),
write(" Last name = ",L_name),nl,nl,
fail.
do_trick :- write("no more names"),nl.
/* правило, що перетворить вхідний рядок у список */
/* слів                                            */
convers(Str,[Head|Tail]):-
fronttoken(Str,Head,Str1),!,
convers(Str1,Tail).
convers(_,[]).
/* правило для перебування останнього елемента списку */
last_element([Head],Last_element):-
Last_element = Head. last_element([_|Tail],Last_element):-
last_element(Tail,Last_element).
/* база даних */
player("Dan Marino","Miami Dolphins",13,"QB",
"6-3",215,4,"Pittsburgh").
player("Richart Dent","Chicago Bears",95,"DE",
"6-5",263,4,"Tennessee State").
player("Bernie Kosar","Cleveland Browns",19,"QB",
"6-5",210,2,"Miami").
player("Doug Cosbie","Dallas Cowboy",84,"TE",
"6-6",235,8,"Santa Clara").
player("Mark Malone","Pittsburgh Steelers",16,"QB",
"6-4",223,7,"Arizona State").
/*****   кінець програми   *****/
```

2.7.3. Програмний інтерфейс природного спілкування з комп'ютером у базі даних гри в футбол

Програма "Створення списку" продемонструвала використання аналізу ключових слів для опрацювання запитів користувача, а програма "Ключові слова" показала, як ключове слово може бути отримане з бази даних. Тепер ми з'єднаємо ці два прийоми і створимо програму природно-мовного інтерфейсу з БД.

Призначенням такої програми буде пошук у БД заданого користувачем прізвища гравця. Якщо на першій позиції в запиті стоять слова **Tell**, **Show** або **Who**, а на останній – прізвище гравця, то внутрішні уніфікаційні процедури Турбо-Прологу виконають швидкий пошук гравця з відповідним прізвищем у базі. Якщо такий гравець буде знайдений, то відомості про нього висвітяться на екрані дисплея. У разі відсутності гравця користувач буде проінформований про те, що потрібних даних немає.

На рис. 2.19 наведена структурна схема програми "Інтерфейс з базою даних "Гра у футбол". Головний модуль, **do_query**, викликає шість інших. Перший і останній з цієї шістки здійснюють відповідно завантаження й очищення БД. Другий модуль, convers, перетворить у список вхідний рядок. Наступні два модулі **first_element** і **last_element** виділяють із вхідного речення перше й останнє ключові слова. Перше з них аналізують на належність до набору припустимих програмою ключових слів, друге (останнє слово речення) задає прізвище футболіста.

П'ятий модуль з викликаних **do_query** – це **do_right_ form**. Він відповідає за перевірку правильності першого ключового слова, що повинне бути або **SHOW**, або **TELL**, або **WHO**. Друге ключове слово передається модулеві **access_database**, що своєю чергою викликає модуль **search_player** для пошуку в БД гравця із заданим прізвищем. **search_player** використовує два інші модулі, **convers** і **last_element**, для ідентифікації прізвища гравця.

Головний модуль має вигляд

```
do_query :-
assert_database,
makewindow(1,7,7,"",0,0,25,80),
 makewindow(2,7,7,"",1,3,4,71),
cursor(1,22), write("PRO FOOTBALL DATABASE"),
```

```
makewindow(3,7,7," Input Info ",6,3,6,37),
repeat, nl,
write(" Please type in your question."),
 cursor(3,2), readln(Sentence),
makewindow(4,7,7," List Maker ",6,42,6,32),
convers(Sentence,List),
first_element(List,Kname),
last_element(List,Lname),
do_right_form(Kname,Lname),
clear_database.
```

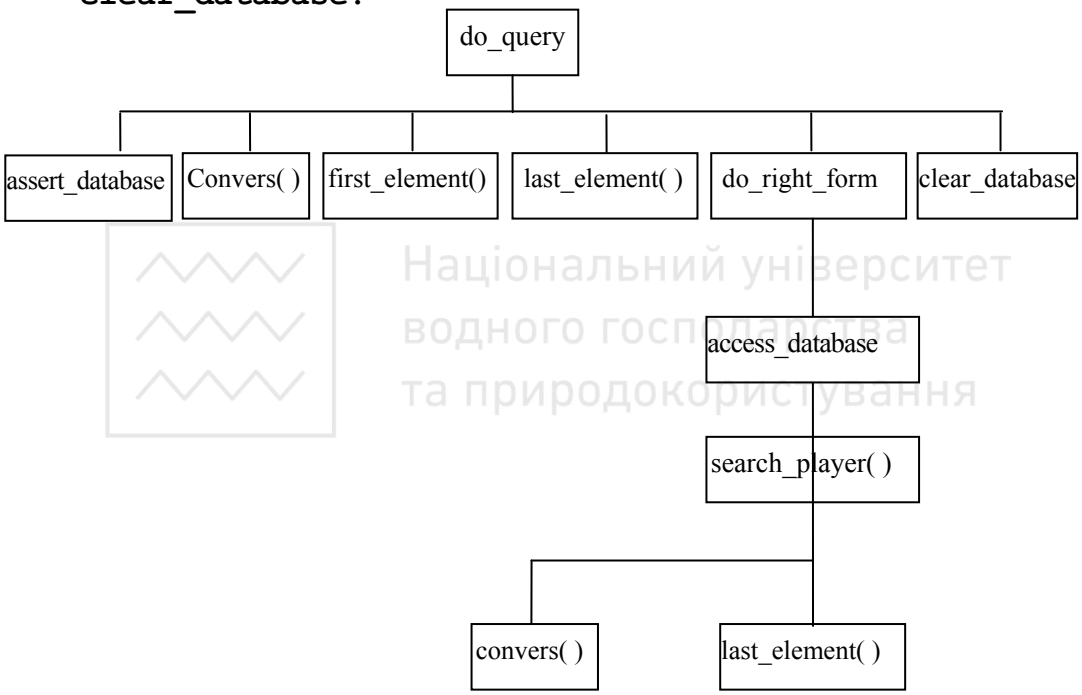


Рис. 2.19. Структурна схема програми "Інтерфейс із базою даних "Гра у футбол"

Правила, що трапляються в цьому модулі, **convers** і **last_element** у точності повторюють однойменні правила з попередньої програми. Правило **first_element** витягає перший елемент зі списку, отриманого з вхідного речення.

```
first_element([Head|_],Fname):-
Fname = Head.
```

Модуль **do_right_form** перевіряє перше ключове слово і витягає з БД прізвище гравця:

```
do_right_form(Kname,Lname) :-
```

```
    keyword(Kname),
    nl, write(" THE KEY WORDS ARE "), nl,
    nl, write("               ",Kname," ",Lname),
    makewindow(5,7,7," Player Info ",12,3,12,71),
    access_database(Lname),
    nl, write(" ANY KEY CONTINUES."),
    readchar(_),
    removewindow,
    gotowindow(4),
    removewindow,
    gotowindow(3),
    clearwindow,!,
    fail.
```

Перевірку ключового слова здійснюють за допомогою виклику правила **keyword:**

```
    keyword(Key) :-
    upper_lower(Upkey,Key),
    Upkey = "SHOW",!.
```


Цей предикат успішний, якщо верхньорегістровий еквівалент першого слова команди – **SHOW**. Аналогічні варіанти правила написані і для ключових слів **WHO** і **TELL**. Розширення словникового запасу системи не становить практично ніяких труднощів.

Якщо ж виконання **keyword** закінчується неуспішно, то випробовують інші варіанти **do_right_form:**

```
    do_right_form(Kname,_) :-
    Kname="Done",!, exit.
    do_right_form(Kname,_) :-
    nl, write(" This keyword is unknown:"), nl,
    nl, write("   ",Kname),
    nl, write(" Press any key to try again."),
    readchar(_),
    removewindow,
    clearwindow,!,
    fail.
```

Перший із наведених варіантів викликає закінчення роботи програми; другий інформує користувача про виявлення незнайомого ключового слова.

Перший варіант правила **do_right_form** викликає модуль **access_database**, а другий - модуль **search_player**, який шукає в БД гравця за його прізвищем.

Модуль **search_player** виглядає так:

```
    search_player(Lname) :-
    dplayer(Name,Team,Number,Position,
    Height,Weight,NFL_Exp,College),
    Team_name    =    Name,    convers(Team_name,Dlist),
last_element(Dlist,Dname), Lname = Dname,
    nl, write("Available information:"),
    nl, nl, write(" ",Name," is a player "),
    write("on the ",Team," team."),
    nl, write(" His number is ",Number),
    write(" and his position is ",Position,"."),
    nl, write(" His weight and height are ",Weight),
    write(" lb. and ",Height," ft-inch."),
    nl, write(" He has ",NFL_Exp,
    " years of NFL experience."),
    nl, write(" He is a graduate of ",College,"."),
    nl, !.
    search_player(_) :-
    nl, nl,
    write("That player is not in the database."),
    nl.
```

У правилі **search_player** твердження БД **dplayer** містять вісім об'єктів. Серед них – змінна **Name,** що визначає повні імена гравців із БД. Змінна **Temp_name** використовується для роботи з копією значення **Name**. Правило **convers** перетворить повне ім'я гравця у список слів **Dlist**. Правило **last_element** отримує зі списку прізвище гравця.

Програма "Інтерфейс із базою даних "Гра у футбол" (лістинг 2.10) реалізує розглянутий проект. Програма використовує кілька вікон; графічні засоби Турбо-Прологу роблять це доволі складне подання інформації простим у реалізації і доволі ефективним.

Лістинг 2.10

```
/* Програма: Природно-мовний інтерфейс з БД          */
/*         "Гра у футбол".                          */
/*         Файл: PROG1103.PRO  */
/* Призначення: Створення природно-мовного інтерфейсу*/
/*                   до БД.    */
domains
str_list = symbol *
str                  = string
p_name, t_name, pos, height, college = string
p_number, weight, nfl_exp = integer
database
dplayer(p_name, t_name, p_number, pos,
height, weight, nfl_exp, college)
predicates
repeat
do_query
assert_database
clear_database
player(p_name, t_name, p_number, pos,
height, weight, nfl_exp, college)
convers(string,str_list)  first_element(str_list,string)
last_element(str_list,string)
access_database(string)
search_player(string)
do_right_form(string,string)
keyword(string)
goal
do_query.
clauses
repeat.
repeat :- repeat.
assert_database :-
player(P_name,T_name,P_number,Pos,Ht,Wt,Exp,
College),
assertz( dplayer(P_name,T_name,P_number,Pos,Ht,
```

```
Wt,Exp,College) ), fail.
assert_database :- !.
clear_database :-
retract( dplayer(_,_,_,_,_,_,_,_) ),
fail.
clear_database :- !.
/* правило, що перетворюють вхідний рядок у список */
/* слів                                              */
convers(Str,[Head|Tail]):-
fronttoken(Str,Head,Str1),!,
convers(Str1,Tail).
convers(_,[]).
/* правило для знаходження першого елемента списку */
first_element([Head|_],Fname):-
Fname = Head.
/* правило для знаходження останнього елемента списку
*/
last_element([Head],Last_element):-
Last_element                    =                    Head.
last_element([_|Tail],Last_element):-
last_element(Tail,Last_element).
/* доступ до бази даних і роздрук інформації */
access_database(Lname) :-
search_player(Lname).
/* пошук гравця */
search_player(Lname) :-
dplayer(Name,Team,Number,Position,
Height,Weight,NFL_Exp,College),
Team_name     =     Name,     convers(Team_name,Dlist),
last_element(Dlist,Dname), Lname = Dname,
nl, write("Available information:"),
nl, nl, write(" ",Name," is a player "),
write("on the ",Team," team."),
nl, write(" His number is ",Number),
write(" and his position is ",Position,"."),
nl, write(" His weight and height are ",Weight),
write(" lb. and ",Height," ft-inch."),
nl, write(" He has ",NFL_Exp,
" years of NFL experience."),
nl, write(" He is a graduate of ",College,"."),
nl, !.
```

```
search_player(_) :-
nl, nl,
write("That player is not in the database."),
nl.
/* правило для цільового твердження */
do_query :-
assert_database,
makewindow(1,7,7,"",0,0,25,80),
makewindow(2,7,7,"",1,3,4,71),
cursor(1,22),
write("PRO FOOTBALL DATABASE"),
makewindow(3,7,7," Input Info ",6,3,6,37),
repeat,
nl,
write(" Please type in your question."),
cursor(3,2),
readln(Sentence),
makewindow(4,7,7," List Maker ",6,42,6,32),
convers(Sentence,List),
first_element(List,Kname),
last_element(List,Lname),
do_right_form(Kname,Lname),
clear_database.
/* правило для перевірки ключового слова */
do_right_form(Kname,Lname) :-
keyword(Kname),
nl, write(" THE KEY WORDS ARE "), nl,
nl, write("                    ",Kname," ",Lname),
makewindow(5,7,7,"     Player     Info     ",12,3,12,71),
access_database(Lname),
nl, write(" ANY KEY CONTINUES."),
readchar(_),
removewindow,
gotowindow(4),
removewindow,
gotowindow(3),
clearwindow,!,
```

```
fail.
do_right_form(Kname,_) :-
Kname="Done",!, exit.
/* інформація про ключові слова */
do_right_form(Kname,_) :-
nl, write(" This keyword is unknown:"), nl,
nl, write("                    ",Kname),
nl, write(" Press any key to try again."),
readchar(_),
removewindow,
clearwindow,!,
fail.
/* перевірка правильності ключових слів */
keyword(Key) :-
upper_lower(Upkey,Key),
Upkey = "SHOW",!.
keyword(Key) :-
upper_lower(Upkey,Key),
Upkey = "WHO",!.
keyword(Key) :-
upper_lower(Upkey,Key),
Upkey = "TELL",!.
/* Футбольна база даних */
player("Dan   Marino","Miami   Dolphins",13,"QB",   "6-
3",215,4,"Pittsburgh").
player("Richart Dent","Chicago Bears",95,"DE",
"6-5",263,4,"Tennessee State").
player("Bernie Kosar","Cleveland Browns",19,"QB",
"6-5",210,2,"Miami").
player("Doug Cosbie","Dallas Cowboy",84,"TE",
"6-6",235,8,"Santa Clara").
player("Mark Malone","Pittsburgh Steelers",16,"QB",
"6-4",223,7,"Arizona State").
/*****   кінець програми   *****/
```


Приклад діалогу з програмою "Інтерфейс із базою даних "Гра у футбол" можна побачити на рис. 2.20. У вікні у верхній частині екрана міститься назва

БД. Вікно Input Info призначене для введення команди користувача, у вікні List Maker наведено значення обох ключових слів. Вікно Player Info містить усю наявну інформації про гравця. Занотуймо, що ця інформація виводиться у вигляді закінчених речень, що робить програму ще більш зручнішою для користувача.

Якщо попрацювати з програмою певний час, то можна побачити, що ні розмір БД, ні збільшення кількості ключових слів не впливають істотно на час роботи внутрішніх уніфікаційних процедур Турбо-Прологу, робота яких реалізується без особливих труднощів. Виведення інформації на екран потребує куди більшого часу, ніж будь-який інший процес у системі.

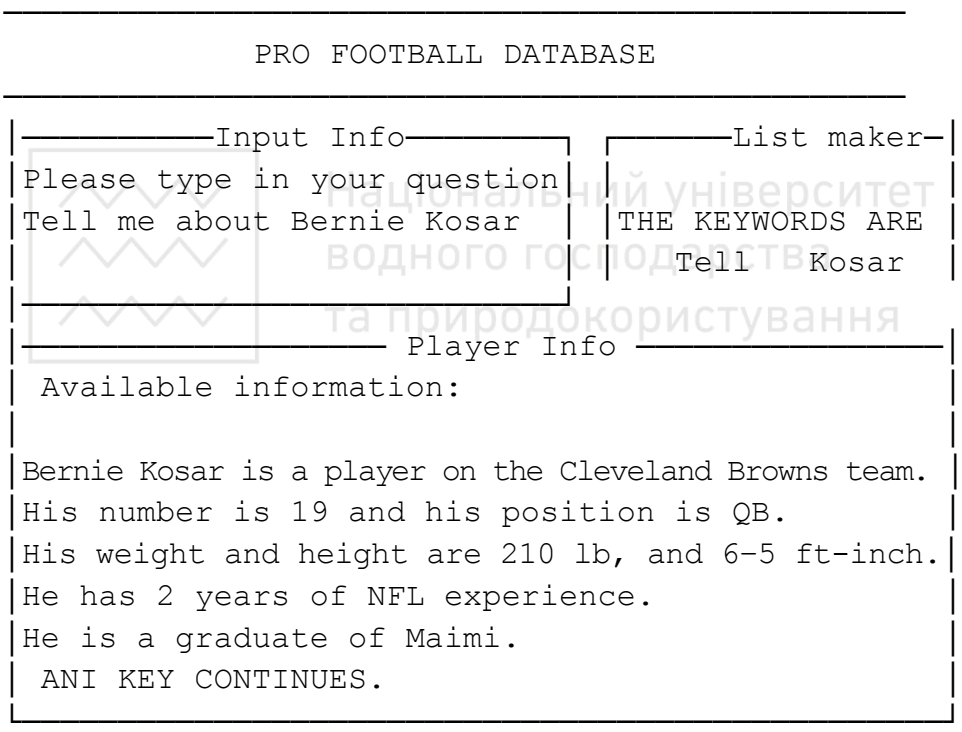


Рис. 2.20. Приклад діалогу з програмою "Інтерфейс із базою даних "Гра у футбол "

2.7.4. Контрольні питання і вправи

1. Підходи до спілкування природною мовою в середовищі Турбо-Пролог.
2. Методи аналізу ключових слів під час побудови речень.
3. Доцільність застосування фактів-капканів для захисту від зациклювання та зависань програми.

4. Найпростіші моделі для контекстно-вільного аналізу.
5. Навести структурну схему для лексичного аналізу речень.
6. Реалізувати фрагмент Пролог-програми швидкого аналізу синтаксису на основі заданої синтаксично правильної бази даних..

2.8. Реалізація ігрових підходів мовою логіки

2.8.1. Основні підходи до розв'язання ігрових задач

Наше щоденне життя складається з багатьох дій, які повторюються, таких, як приготування їжі, харчування, сон та багато іншого. Для звичайної людини це потребує великих затрат енергії, але невеликого напруження розумових здібностей.

Але ігри та головоломки потребують того, щоб над ними замислились. Це одна з причин їхньої популярності. Задачі ігор та головоломок добре формалізовані. Тому значна кількість перших робіт в галузі штучного інтелекту були орієнтовані саме на них.

Використовуючи Турбо-Пролог, можна розробити та реалізувати завершену програму для розв'язання головоломки чи будь-якої гри. Внутрішні уніфікаційні підпрограми Турбо-Прологу можуть виконувати багаторівневий пошук і порівняння зі зразком, що дає змогу легко та ефективно реалізувати методи розв'язання задач.

Для багатьох задач метод розв'язання потребує виконання ланцюга суджень, починаючи з бажаного результату і закінчуючи заданими умовами. В ШІ цей підхід називається **зворотним виведенням**. Турбо-Пролог має можливості для зворотного виведення, яка є його фундаментальною властивістю.

Загалом в Пролозі та в Турбо-Пролозі зазвичай ціль правила містить параметри, значення яких є невідомими. Підцілі тіла цього правила намагаються обчислити параметри цілей. Саме це і є фундаментальною властивістю Прологу, і отже зворотне виведення є дуже характерним Прологу.

Під час розв'язання задач іншого виду можуть знадобитися всі необхідні проміжні кроки і відповідні умови для того, щоб досягти будь-якого з можливих розв'язків чи відповідей. У такому разі розв'язування задачі починають з відомих фактів або умов і виконують як прямий пошук допустимого рішення. Цей підхід називають **прямим виведенням**.

2.8.2. Інтелектуальна гра "23 сірники" та її реалізація

Гра в "23 сірники" є однією з багатьох ігор для двох осіб. Ця гра дуже цікава, оскільки вона може бути реалізована як проста гра або як доволі складна з багатьма варіантами.

Гра в "23 сірники" полягає в такому. Гравці починають з 23 сірників (або паличок). Кожен гравець по черзі видаляє 1, 2 або 3 сірники. Той, хто візьме останній сірник, програв. Отже, мета гри – змусити іншого гравця взяти останній сірник.

Розробка першої програми мовою Турбо-Пролог для цієї гри показана в розділі 2.8.2.1.

2.8.2.1. Найпростіша програма (розгляд проекту)

Існують два варіанти гри в 23 сірника:

Варіант з однією купкою

Номер купки	Стан купки	Кількість сірників
1	1,2,3,...22,23	23

Варіант з чотирма купками

Номер купки	Стан купки	Кількість сірників
1	1 2 3 4 5	5
2	6 7 8 9 10 11	6
3	12 13 14 15 16 17	6
4	18 19 20 21 22 23	6

У першому та другому варіанті гри кожен гравець по черзі бере 1, 2 або 3 сірники з кожної купки. В двох варіантах сумарна кількість сірників, які залишаються після ходу кожного гравця, згодом визначить того, хто програв.

Спочатку розглянемо простіший варіант з однією купкою. Для реалізації цієї гри мовою Турбо-Пролог комп’ютер стає одним із гравців, а користувач – другим. Основна процедура передбачає декілька кроків:

1. Вибрати 23 сірники як початкову кількість та помістити їх в одну купку.

2. Попросити користувача видалити 1, 2 або 3 сірники. Порахувати кількість сірників, які залишилися в купці. Можливі два варіанти:

a. Якщо залишився лише один сірник, то комп’ютер повинен видалити його. У такому разі комп’ютер програв.

b. Якщо залишилося більше від одного сірника, то перейти до третього кроку.

3. Комп'ютер видаляє 1, 2 або 3 сірники. Підрахувати кількість сірників, які залишилися в купці.

a. Якщо залишився лише один сірник, то користувач повинен взяти його. У такому разі користувач програв.

b. Якщо залишилося більше від одного сірника, то перейти до другого кроку.

Сама по собі процедура є дуже простою. Тим не менше кількість сірників, які видаляє гравець, визначає переможця і з цього видно, що стратегія є важливим компонентом в структурі гри.

2.8.2.2. Програма простої гри в "23 сірники"

Під час реалізації цієї простої гри можна дозволити користувачу вибирати стратегію гри самостійно. Піклуватися тут треба тільки про комп'ютер. Комп’ютер повинен генерувати числа 1, 2 або 3 випадково та використовувати це число у модулі гри. Для написання програми гри в "23 сірники" необхідно реалізувати три кроки, які ми розглядали в розділі 2.8.2.1. Перший крок – це встановлення початкової кількості сірників, яке дорівнює 23. Потім можна передати це число з головного модуля в підмодуль у співвідношенні із структурою програми (див. рис. 2.21).

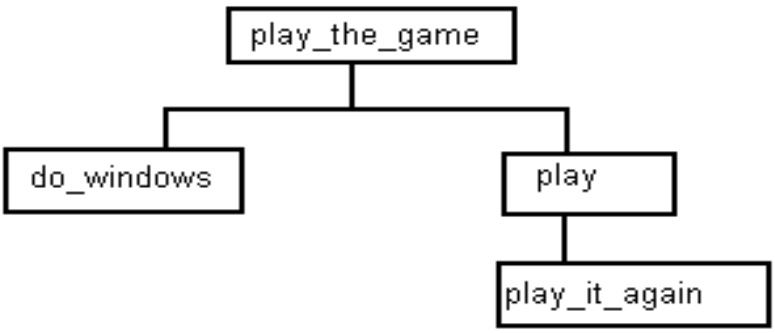


Рис. 2.21. Структура програми простої гри в "23 сірники"

Головний модуль має вигляд

```
play_the_game : -
do_windows,
play(23,0,0).
```

У цьому модулі правило do windows створює вікна і видає допоміжну інформацію. Правило **play (23,0,0)** фактично виконує всю гру. Це правило викликає інші модулі і передає їм вихідне число 23.

Правило **play(M,H,C)** складається з підправила, що забезпечує виконання ігрової послідовності доти, доки не визначиться переможець:

```
play(M,H,C)
play_it_again(M,H,C).
```

Змінні **M, H** і **C** використовують для позначення **matches (сірники)**, **human (людина)** і **computer (комп'ютер)** відповідно.

Правило **play_it_again** рекурсивне. Воно має три альтернативи, які реалізують три можливі закінчення процесу гри: виграє людина, виграє комп'ютер, гра продовжується.

Якщо виграє людина, то правило має вигляд

```
Play_it_again (M,H,C)
M <= 0,
nl, write(" Ти виграв, людина!"), nl ,
nl, write(" Натисніть клавішу пробілу."),
readchar( ),
clearwindow, !.
```

Предикат **M<=0** визначає, що останній хід у грі зробив комп'ютер. У такому разі він "узяв" або всі сірники **(M=0)**, або більше сірників, ніж фактично є **(M<0)**. У тому і іншому разі комп'ютер програє.

Якщо виграє комп'ютер, то відповідне правило має вигляд

```
Play_it_again (M,H,C)
M=1,
nl, write(" Я, комп'ютер, виграв !"),nl,
readchar( ),
clerawindow, !.
```

У цьому варіанті правила предикат рівності **M=1** означає, що поточна кількість сірників дорівнює 1 і що цей останній сірник повинен бути вилучений людиною під час наступного ходу, тобто комп'ютер виграє. Як перший, так і другий варіанти правила **play_it_again** у разі успішного закінчення виконують вихід із програми.

Третій варіант правила є серцевиною ігрової послідовності:

```
Play_it_again(M,H,C)
nl,write(" Ваш хід."),
nl,write(" Скільки сірників ви хочете забрати?"),
readint(Hn),
M2 = M - Hn,
H2 = Hn,
write(" Тепер  маємо",M2,"сірників"),
nl,nl,write(" Мій хід."),
nl,write(" Я думаю !"),
random(F),
Rea = 1 + 3 * F,
real int(Rea,Rint),
M3 = M2 - Rint,
nl write(" Я видалив ",Rint,"."),
n1,write(" Тепер маємо ",M3,"сірників"),nl,
M7 = M3,
H7 = H2,
C7 = Rint,
Play_it_again(M7,H7,C7).
```

Цей варіант правила зчитує відповідь, введену користувачем, обчислює кількість сірників, що залишилися, і видає це число на екран. Потім правило зчитує значення комп'ютера (випадкове ціле число, що дорівнює 1, 2 або 3), модифікує кількість сірників і видає змінене значення. Третій варіант правила є рекурсивним правилом.

Описані правила використовуються в програмі простої гри в 23 сірники (лістинг 2.11). Ця програма використовує засоби віконного інтерфейсу Турбо-Пролог і видає допоміжні повідомлення і запрошення.

Лістинг 2.11

```
    /* Програма: Простий варіант гри в "23 сірники" */
    /* Призначення: Демонстрація гри в „23 сірники" */
    /* Вказівка: Запустіть програму. Ціль утримується в
програмі */

    domains
    predicates
    play_the_game
    do_windows
    play(integer, integer, integer)
    play_it_again(integer, integer, integer)
    real_int(rea1, integer)
    goal
    play_the_game.
    clauses
    /* ціль є правилом */
    play_the_game : -
    do_windows,
    p1ay(23,0,0).
    /* правило для створення вікон    */
    do_windows :—
    makewindow(1,7,7,"",0,0,25,80),
    makewindow(2,7,7," Game of 23 Matches ",
    1,5,22,40),
    nl,write("Welcome to the Game of 23 Matches. ),
    M = 23,
    H= O,
    C=0,
    nl,write("There are ",M," matches to begin. ),
    nl,write("Me     will     take     turns     removing
matches."),nl,
    nl,write("Each time you may remove 1, 2, or 3"),
    nl,write(matches and then 1' 11 do the same."),nl,
    nl,write("The p1ayer who has to remove"),
    nl, write("the last match will lose. ),
```

```
    nl,write("To begin, there are 23 matches. ),
    nl write("The human has removed ",H," match. ),
    nl,write("   The   computer   has   removed   ",C,"
match."),nl.
    play(M,H,C)
    play_it_again (M,H,C).
    /* правило для визначення переможця */
    Play_it_again (M,_,_ ) : -
    M <= 0,
    nl, write(" You've won !"), nl,
    nl, write(" Press the SPACE bar. ),
    readchar(),
    clerawindow, !,
    exit.
    Play_it_again (M,_ ,_ ) : -
    M=1,
    nl, write(" 1, the computer, won !"),nl,
    readchar( ),
    clerawindow, !.,
    exit.
    Play_it_again(M,_,_ ) : -
    nl,write(" Your turn. ),
    nl,write(" How many do you want to remove?"),
    readint(Hn),
    M2 = M2 - Hn,
    H2 = Hn,
    write(" Now there are ",M2," match(es). ),
    nl,nl,write(" My turn. ),
    nl,write(" 1 am deciding !"),
    random(F),
    Rea = 1 + 3 * F,
    Real_int(Rea,Rint),
    M3 = M2 - Rint,
    n1,write(" 1 removed ",Rint,". ),
    nl,write("   Now   there   are   is(are)   ",M3,"
match(es)."),nl,
    M7 = M3,
```

```
H7 = H2,
C7 = Rint,
Play_it_again(M7,H7,C7).
/* допоміжне правило  */
readint(Re,In) Re = In.
/*          кінець програми          */
```

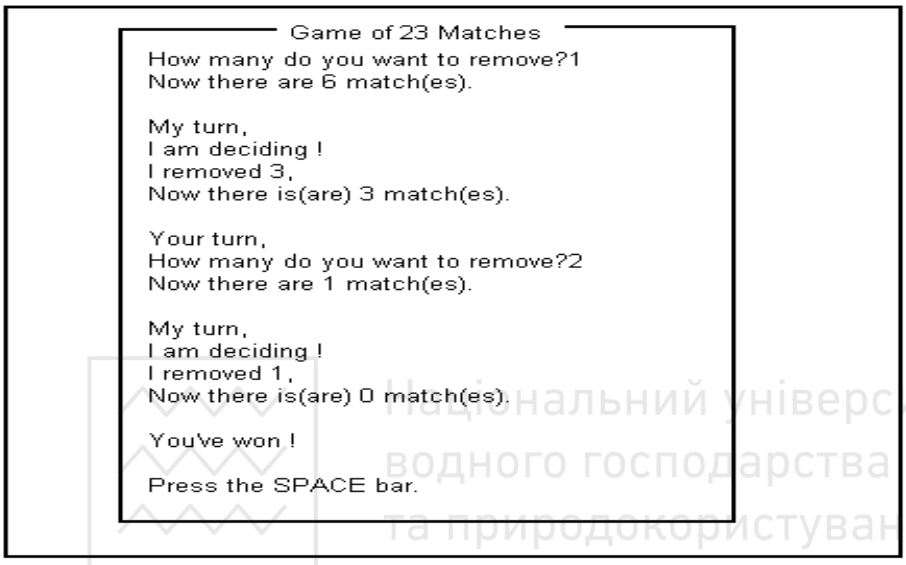


Рис. 2.22. Сеанс із програмою простої гри в "23 сірники"

На рис. 2.22 наведено діалог між користувачем і програмою. Як бачите, діалог простий і зрозумілий, так що користувач може легко проаналізувати ситуацію, що склалася, і вирішити, яке число ввести. Користувач може використовувати власну виграшну стратегію, на відміну від комп'ютера, що у випадковий спосіб вибирає числа з ряду 1, 2, 3. Ці дії не є стратегією, швидше це сподівання на успіх. Тому користувач має більше шансів виграти, ніж комп'ютер.

2.8.2.3. Інтелектуальна гра в "23 сірники"

У простій грі в "23 сірники" комп'ютер як гравець був запрограмований на вибір кількості сірників, що вилучаються, у випадковий спосіб. Комп'ютер грав, не маючи виграшної стратегії. Програма інтелектуальної гри в "23 сірники" показує, як реалізувати виграшну стратегію для комп'ютера.

У грі "23 сірники" кожен гравець намагається залишити іншому

гравцеві тільки один сірник, щоб саме він його взяв. Початкова кількість сірників 23, кінцева мета — один сірник, а кожен гравець може вилучити 1, 2 або 3 сірники. Отже, можна побачити послідовність початкових, проміжних і кінцевих станів купки з погляду кожного гравця.

Для того щоб продемонструвати ситуації, що виникають під час гри, у табл. 9 показані ігрові послідовності. Для першої гри ці послідовності наступні:

```
S =[23,20,17,14,13,10,9,6,5,4,1]
S = [23,17,13,9,5,1] — залишені користувачеві
S =[ 20,14,10,6,4] – залишені комп'ютерові.
```

Таблиця 2.9

Ігрові послідовності для гри в "23 сірники"

	Вилучена користувачем кількість сірників	Кількість сірників, що залишилося	Вилучена комп'ютером кількість сірників	Кількість сірників, що залишилося
Послідовність № 1	3	20	3	17
	3	14	1	13
	3	10	1	9
	3	6	1	5
	1	4	3	1
				комп’ютер виграє
Послідовність № 2	1	22	1	21
	2	19	2	17
	3	14	1	13
	2	11	2	9
	1	8	3	5
	2	3	2	1
				комп’ютер виграє

Для другої гри ці послідовності такі:

```
S = [23,22,21,19,17,14,13,9,8,5,3,2,1]
S = [23,21,17, 13,9,5, 1] — залишені користувачеві
S = [22, 19,14,11,8,3] — залишені комп'ютерові.
```

Аналіз показує, що деякі числа в послідовності для кожного з гравців небажані, якщо вони залишені супротивником. Це числа: 2, 3, 4, 6, 7 і 8. З іншого боку, числа 1, 5 і 9 можна спокійно залишити супротивникові.

Відзначимо, що в послідовностях, показаних для цих двох ігор, обидві послідовності користувача містять такі "небезпечні" числа. Перша послідовність для комп'ютера містить 6 і 4, і ці два числа "безпечні".

Цей аналіз також показує, що кожен гравець намагається залишити "небезпечні" числа супротивникові. Тому доцільно знайти співвідношення між кількістю сірників, що вилучаються, і "небезпечною" кількістю сірників, що залишається супротивникові. Ентузіасти і теоретики вивчили цей аспект гри і запропонували різні рішення.

Спрощена версія рішення наводиться нижче. Якщо **M** – кількість сірників, які залишилися, то треба вилучити **C** сірників. **C** обчислюють за формулою

```
C = (R + 3) — 4 * integer((R + 3)/ 4),
де R = M - (4 * integer(M/4)).
```

Тут **integer** повертає цілу частину значення виразу в дужках.

Наприклад, якщо M = 8, то C = 3; якщо M = 6, то C = 1.

Ці формули можуть бути записані як правила Турбо-Прологу. Послідовність їхнього запису така:

```
A_r = M2/4 - 0.45,
Real_int(A_r, A_int)
Al_int = M2 - (4 * A_int),
A2_r = (Al_int + 3)/4 - 0.45,
кeal_int(A2_r, A2_int),
A3_int = (Al_int + 3) - (4 * A2_int).
```

Правила присвоюють **A3_int** ціле значення кількості сірників, які повинні бути вилучені. (Правило **real_int** необхідне для перетворення дійсного числа в ціле).

Запропонована виграшна стратегія може бути використана (у вигляді правил Турбо-Прологу) для заміни простої випадкової стратегії на основі випадкового числа, що використовувалася в попередній програмі гри в "23 сірники". У результаті одержимо програму інтелектуальної гри в "23 сірники".

Лістінг 2.12

```
/* Програма: Інтелектуальний варіант гри "23 сірники" */
/* Призначення: Демонстрація гри в "23 сірники" */
/*          Використовується виграшна стратегія      */
/* Вказівка: Запустіть програму.               */
/*                Ціль утримується в програмі        */

domains
predicates
play_the_game
do_windows
play(integer, integer, integer)
play_it_again(integer, integer, integer)
real_int(real, integer)
goal
play_the_game.
clauses
/* ціль є правилом */
Play_the_game :-
do_windows,
play(23,0,0).

/* правило для утворення вікон */
do_windows :—
makewindow(1,7,7, ',0,0,25,80),
makewindow(2,7,7," Game of 23 Matches ",
                    1,5,22,40),
makewindow(3,7,7, Observer ",1,50, 10,20),
cursor(2,2),
write("RESULT UPDATE"),
gotowindow(2),
n1,write(" Welcome to the game of 23 Matches.”),
M = 23,
H=O,
C=O,
n1,write(" There are ",M," matches to begin.),
nl,write(" The human has removed ",H," match.),
```

```
nl,write(" The computer has remo ed ",C," match."),nl.

/* правило для виконання гри */
play(M,H,C)
play_ it_again(M,H,C).

/* правило для видачі повідомлення, */
/* що виграла людина */
Play_it_again(M,_,_ )
M <= O,
n1, write(" You the human won !"),nl,
makewindow(4,7,7,"Result ",13,50, 10, 15),
nl,write(" YOU WON ! ),nl,nl,nl,

/* звуковий сигнал для користувача */
sound(4,392),
sound(4,440),
sound(4,494),
sound(4,440),
sound(4,494),
sound(4,392),
nl, write(" Press the SPACE bar. ),
readchar( ),
clerwindow, !,
exit.

/* правило для видачі повідомлення, */
/* що виграв комп'ютер */
Play_it_again (M,_,_ ) ;-
M=1,
n1   write(" 1, the computer, won !"),nl,
makewindow(5,7,7, "Resuit",13,50,10,20),
nl,write(" 1 WON ! " ), nl, nl, nl,

/* звуковий сигнал для користувача */
sound(4,394),
sound (4,440),
```

```
sound (4,494),
sound(4,440),
sound (4,494),
sound(12,392),
nl, write(" Press the SPACE bar. ),
readchar(),
clerawindow, !,
exit.
/* основне правило, що виконує гру */
/* і повторюване декілька раз */
Play_it_again(M,_,_ ) :-
n1,write(" Your turn. ),
nl,write(" How many do you want to remove?"),
readint(Hn),
M2 = M2 - Hn,
H2 = Hn,
write(" Now there are ",M2," match(es) ),
gotowindow(3),
cursor(4,3),
write(M2, " match(es) "),
gotowindow(2),
nl,nl,write(" My turn. ),
nl,write(" 1 am thinking !"),
A_r = M2/4 — 0.45,
Real_int(Ar, Aint),
Al_int = M2 — (4 * A_int)
A2_r = (Al_int + 3)/4 — 0.45,
Real_int(A2_r, A2_int)
A3_int (Al_int + 3) — (4 * A2_int),
M3 = M2 - A3_int,
nl,write(" I removed ",A3_int,"."),nl,
nl,write("    Now    there    are    is(are)    ",M3,"
match(es)."),nl,
gotowindow(3),
cursor(4,3),
write(M3, " match(es) "),
gotowindow(2),
```

```
M7 = M3,
H7 = H2,
C7 = A3  int,
Play_it_again(M7,H7,C7).
/* правило перетворення дійсного */
/* числа в ціле */
Real_int(Re,In) :-  Re = In.
/*          кінець програми                  */
```

Інтелектуальна програма гри в "23 сірники" видає кількість сірників у купці. Потім вона зчитує введене користувачем значення кількості сірників, які видаляються, модифікує число сірників у купці і видає їхню нову кількість.

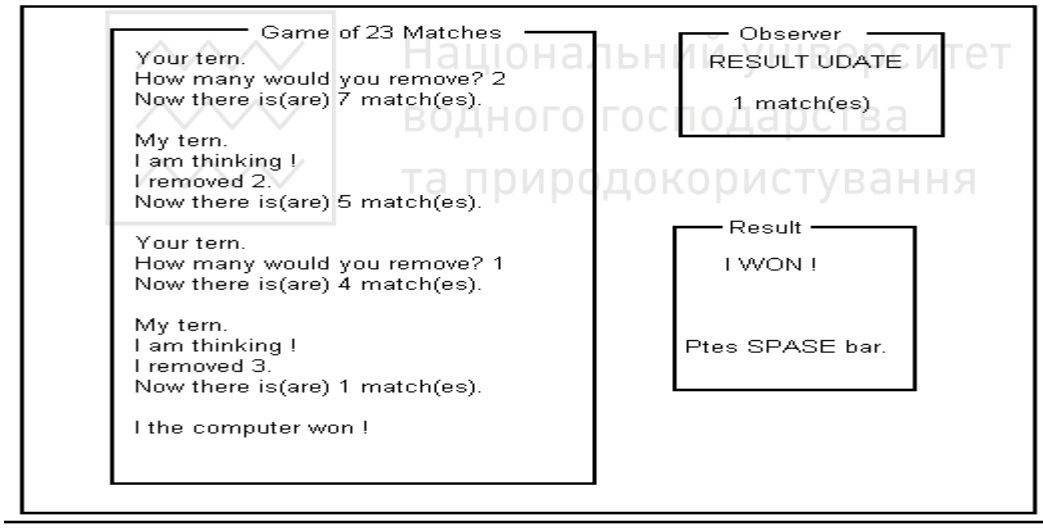


Рис. 2.23. Сеанс інтелектуальної гри в "23 сірники"

Під час гри комп'ютера програма використовує формулу виграшної стратегії для обчислення кількості сірників, що видаляються. Потім програма видаляє знайдену кількість сірників з купки і видає нову кількість сірників, що залишилися.

Гра повторюється доти, доки один із гравців не виграє. Потім визначається переможець і звучить музичний сигнал.

Зверніть увагу, що програма має два додаткові вікна в правій половині екрана. Верхнє вікно показує модифіковану кількість сірників після кожного

ходу. Нижнє вікно демонструє переможця. На рис. 2.23 показано сеанс інтелектуальної гри в "23 сірники". Відзначимо, що комп'ютер виграв.

2.9. Алгоритм та реалізація мовою логіки гри “Мавпа і банани”

Задача "Мавпа і банани" – це класична задача ШІ, яка демонструє добре відомі методи висновку під час програмування мовою Турбо-Пролог. Задача формулюється в такий спосіб.

Мавпа перебуває в зачиненій кімнаті, на підлозі лежить коробка, а в'язка бананів звисає зі стелі так, що дістати її мавпа не може.

Розв’язання задачі – це послідовність дій, що допоможуть мавпі дотягтись до бананів. Послідовність дій мавпи може бути організована в такий спосіб.

Мавпа підходить до коробки, ставить її під в’язку бананів, стає на коробку і хапає банани.

2.9.1. Розробка програми

Програма, що реалізує задачу "Мавпа і банани", повинна містити у базі даних деякі конкретні факти. Ними є місце знаходження мавпи в кімнаті, розташування коробки в кімнаті, позиція на підлозі кімнати, над якою висить в'язка бананів, і інформація про те, чи сидить мавпа на коробці, чи ні. Для цієї задачі місце розташування бананів, коробки і мавпи змінюється і може бути представлене змінними Турбо-Прологу. Причому місце перебування мавпи і коробки може бути будь-яким. Попередній аналіз приводить до таких початкових умов:

- банани над позицією 1 – 10;
- коробка в позиції 1 – 10;
- мавпа в позиції від 1 – 10;
- мавпа не на коробці;
- у мавпи немає бананів.

Ці факти описують початковий стан. Якщо мавпа діє успішно, то остаточний результат буде описуватися за допомогою таких фактів:

- банани в первісній позиції;
- коробки в тій самій позиції, що і банани;

- мавпа в тій самій позиції, що і банани;
- мавпа на коробці;
- мавпа схопила банани.

Тепер необхідно побудувати правила трансформації вихідного стану в кінцевий. Нижче наводяться можливі формати для цих правил:

```
move_to(box,position, position).          /* пересунути*/
move_to(monkey, position, position).      /*пересунути*/
go_to(position, position, position).            /*перейти*/
push_box(position, position, position).  /*перемістити коробку*/
climb_box.                              /*піднятися на коробку*/
grasp_bananas                           /*схопити банани*/
```

Перше правило **move_to** призначене для пересування коробки в нове місце, а друге правило **move_to** призначене для переміщення мавпи в нову позицію. Правило **go_to** призначене для логічного переміщення коробки або мавпи з однієї позиції в іншу, Правило **push_box** призначено для пересування мавпою коробки з одного положення в інше. Правило **climb_box** призначене для того щоб мавпа вилізла на коробку. Правило **grasp_bananas** призначене для хапання бананів.

2.9.2. Реалізація програми "Мавпа і банани"

Використовуючи базу даних і правила, розглянуті в попередньому розділі, можна написати програму мовою Турбо-Пролог, що розв'язує задачу про мавпу і банани. Блок-схема цієї програми показана на рис. 14.1.

Основний модуль на цій блок-схемі — це **solve_the_problem** (розв'язати задачу). Цей модуль викликає підмодулі **request_positions** (запросити позиції) і **go_and_get_bananas** (йди і візьми банани). Перший з них запрошує користувача ввести місце розташування мавпи, коробки і бананів. Ці координати присвоюються змінним **M**, **Bo** і **B.** Модуль **monkey_works** (мавпа працює) викликає чотири модулі: **move_to(box,B,Bo)**, **move_to(monkey,Bo,M)**, **climb_box** і **grasp_bananas**. Модуль m**ove_to(box,B,Bo)** викликає модуль **push_box(C,B)**, що своєю чергою, викликає модуль **move_to(monkey,B,M).** Останній модуль викликає модуль **go_to(M,B)** для виконання переміщення мавпи з однієї позиції в іншу. Модуль **climb_box** виконує переміщення мавпи, а модуль **grasp_bananas** – хапання бананів мавпою.

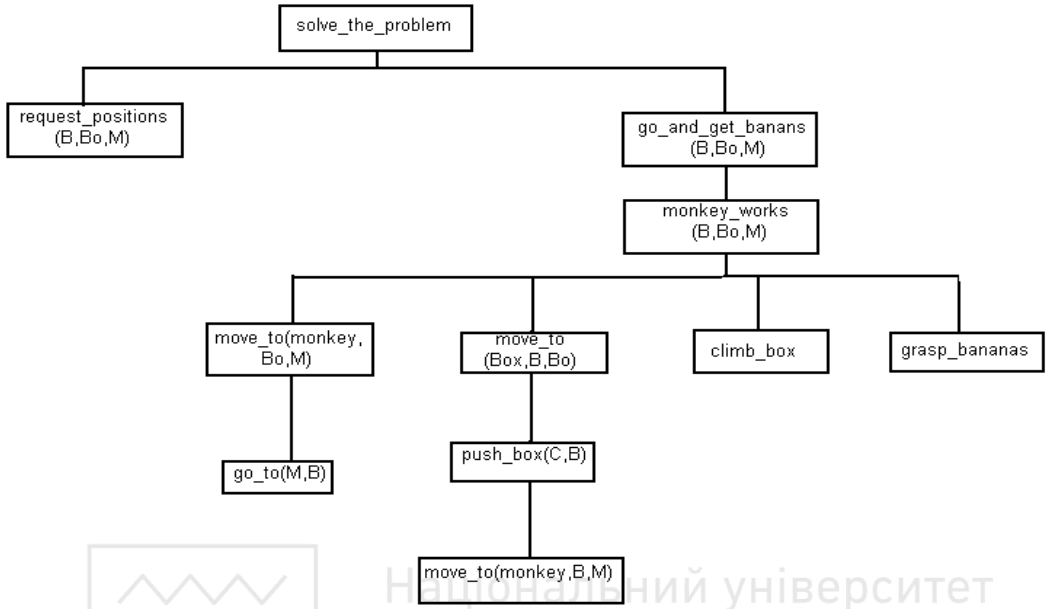


Рис. 2.24. Структура програми "Мавпа і банани"

На лістингу 2.13 наведена програма "Мавпа і банани", яка є реалізацією розглянутого проекту.

Лістинг 2.13

```
/* Програма: Мавпа і банани PROG1204.PRO */
/* Призначення: Демонстрація розв'язання          */
/*              головоломки про мавпу і банани*/

domains
database
is_at(symbol, integer)
monkey_is_off_box
predicates
solve_the_problem
request_position(integer, integer, integer)
go_and_get_bananas(integer, integer, integer)
monkey_works(integer, integer, integer)
```

```
move_to(symbol, integer, integer)
go_to(integer, integer)
push_box(integer, integer)
climb_ box
grasp_bananas
goal
solve_the_problem
clauses

/* база даних */
monkey_is_off_box.
is_at(bananas,1).          is_at(bananas,2).
is_at(bananas,3).          is_at(bananas,4).
is_at(bananas,5).          is_at(bananas,6).
is_at(bananas,7).          is_at(bananas,8).
is_at(bananas,9).          is_at(bananas, 10).
is_at(box,1).              is_at(box,2).
is_at(box,3).              is_at(box,4).
is_at(box,5).              is_at(box,6).
is_at(box,7).              is_at(box,8).
is_at(box,9).              is_at(box,10).
is_at(monkey,1).           is_at(monkey,2).
is_at(monkey,3).           is_at(monkey,4).
is_at(monkey,5).           is_at(monkey,6).
is_at(monkey,7).           is_at(monkey,8).
is_at(monkey,9).           is_at(monkey, 10).

/* правила досягнення бананів */
solve_the_problem :-
make_window(1,7,7,"",0,0,25,80),
make_window(2,7,7," MONKEY AND BANANAS ",
1,10,23,50),
request_position(B,Bo,M),
nl,write(" Monkey is thinking. "),
nl,write(" '1 want those bananas"),
write(" hanging up there !'"), nl,
go_and_get_ bananas(B,Bo,M),
```

```
write(" Monkey gets the bananas !"), nl,nl,
write(" THREE CHEERS !!!"), nl,nl,
write(" Press the SPACE BAR."),
readchar( ).
request_position(B,Bo,M) :-
n1, n1,
write(" Enter position of the bananas(1-10). "),
readint(B),
write(" Enter position of box(1-10). "),
readint(Bo),
write(" Enter positin of monkey(1-10). "),
readint(M).
go_and_get_bananas(B,Bo,M)
is_at(bananas,B),
is_at(box,Bo),
is_at(monkey,M),
monkey_works(B,Bo,M).
monkey_works(B,Bo,M) :-
n1, write(" Monkey looks around and spies the box. "),
move_to(monkey, Bo,M),
n1, write(" monkey goes from ""',M," to ",B,"."), n1,
move_to(box,B,Bo),
n1, write(" Monkey pushes the box from ",Bo,
" to ",B,"."), n1,
climb_box,
grasp_bananas.
move_to(monkey, B,M)
is_at(monkey, B),
is_at(monkey,1),
go_to(M,B),
fail.
Move_to(box,B,M),
is_at(box,B),
is_at(box,C),
push_box(C,B).
go_to(B,C)  :-
monkey_is_off_box,
```

```
retract(is_at(monkey,B)),
assert(is_at(monkey,C)),
nl,write(" Monkey goes ",B," to ",C,").
push_box(B,C)
monkey_is_off_box,
move_to(monkey, B,M),
retract(is_at(monkey, B)),
retract(is_at(box,B)),
assert(is_at(monkey,C)),
assert(is_at(box,C)).
climb box :-
monkey_is_off_box,
retract(monkey_is_off_box),
write(" Monkey climbs the box."), n1
 grasp_bananas :- .
write(" Monkey grasp the bananas."), n1.
/* кінець програми */
```

База даних "Мавпа і банани" складається з наступних тверджень:

```
monkey_is_off_box.
is_at(bananas, 1).        is_at(bananas,2).
is_at(bananas,3).         is_at(bananas,4).
is_at(bananas,5).         is_at(bananas,6).
is_at(bananas,7).         is_at(bananas,8).
is_at(bananas,9).         is_at(bananas,10).
is_at(box,1).             is_at(box,2).
is_at(box,3).             is_at(box,4).
is_at(box,5).             is_at(box,6).
is_as(box,7).             is_at(box,8).
is_at(box,92).            is_at(box,10).
is_at(monkey,1).          is_at(monkey,2).
is_at(monkey,3).          is_at(monkey,4).
is_at(monkey,5).          is_at(monkey,6).
is_at(monkey,7).          is_at(monkey,8).
is_at(monkey,9).          is_at(monkey,10) .
```

Перше твердження – це простий факт: мавпа не на коробці. Інші твердження відображають припустиме місце розташування бананів, коробки і мавпи.

Цільовий модуль має вигляд

```
solve_the_ problem :-
make_window(1,7,7,"",0,0,25,80),
make_window(2,7,7," MONKEY AND BANANAS ",
1,10,23,50),
request_position(B,Bo,M),
nl,write(" Monkey is thinking. "),
nl,write(" 'I want those bananas"),
write(" hanging up there ! ), nl,
go_and_get_bananas(B,Bo,M),
write(" Monkey gets the bananas !"), nl,nl,
write(" THREE CHEERS !!!"), nl,n1,
write(" Press the SPASE BAR. ),
readchar( ).
```


Цей модуль видає інформаційний текст у вікно на екрані. Потім він викликає два модулі, **request_position** і **go_and_get_bananas**. Другий з них виконує "захоплення бананів", викликаючи підмодуль **monkey_works**. Основний модуль видає деякий підсумковий текст.

Правило **monkey works** реалізоване в такий спосіб.

```
monkey_works(B,Bo,M) :-
n1, write(" Monkey looks around and spies the box. "),
move_to(monkey, Bo,M),
n1, write(" Monkey goes from ",M," to ",Bo,"."), n1,
move_to(box,B, Bo),
n1, write(" Monkey pushes the box from ",Bo,
“to” ’B’ ”.” ), n1
climb_box ,
grasp_bananas .
```

Основна задача цього модуля розділена на послідовність менших підзадач. Правила **move_to** і **go_to** разом переводять мавпу з одного

положення в інше:

```
move_to(monkey, B,M)
is_at(monkey, B),
is_at(monkey,M),
до_to(M,B),
fail.

move_to(box,B,M),
is_at(box,B),
is_at(box,C),
push_box(C,B).

go_to(B,C)
monkey_is_off_box,
retract(is at(monkey, B)),
assert(is_at(monkey,C)),
n1, write(" monkey goes ",B," to ",C,").
```


Правило **move_to** перевіряє допустимість позицій **B** і **C**. Правило **go_to** фактично вилучає з бази даних попереднє розташування мавпи і додає туди її поточне положення.

Правила **move_to** і **push_box** спільно виконують переміщення коробки з одного положення в інше:

```
move_to (box ,B, M) :-
is_at(box, B),
is_at(box, C),
push_box(C ,B).

push_box(B, C) :-
monkey_is_off_box,
move_to(monkey, B, M),
retract(is_at(monkey, B)),
retract(is_at(box, B)),
assert(is_at(monkey, C)),
assert(is_at(box, C)).
```

Правило **move_to** перевіряє допустимість параметрів **B** і **C** і передає ці параметри правилу **push_box.** Правило **push_box** викликає правило **move_ to(monkey, B, M)** для того, щоб включити мавпу в процес пересування коробки в нове місце. Правило **push_box** видаляє з бази даних попередні твердження про мавпу і коробки і додає на їх місце нові.

Якщо досягнуто кінцеве положення про мавпу і коробки, то правила **climb_box** і **grasp_ bananas** змушують мавпу залізти на коробку і схопити банани:

```
climb_box :-
monkey_is_off_box,
retract(monkey_is_off_box),
write (" Monkey climbs the box."), nl.
grasp_bananas:-
write (" Monkey grasp the bananas."), nl.
```

Тіло правила **climb_box** вилучає твердження **monkey_is_off_box**, що відповідає за те, що мавпа залазить на коробку. Правило **grasp_bananas** не має параметрів і імітує хапання мавпою бананів.

На рис. 2.25 показаний сеанс із програмою "Мавпа і банани". На екрані, подана інформація про початкове положення мавпи і коробки, а також позиція, над якою висять банани. Короткий текст сценарію показує дії мавпи і роботу правил програми.

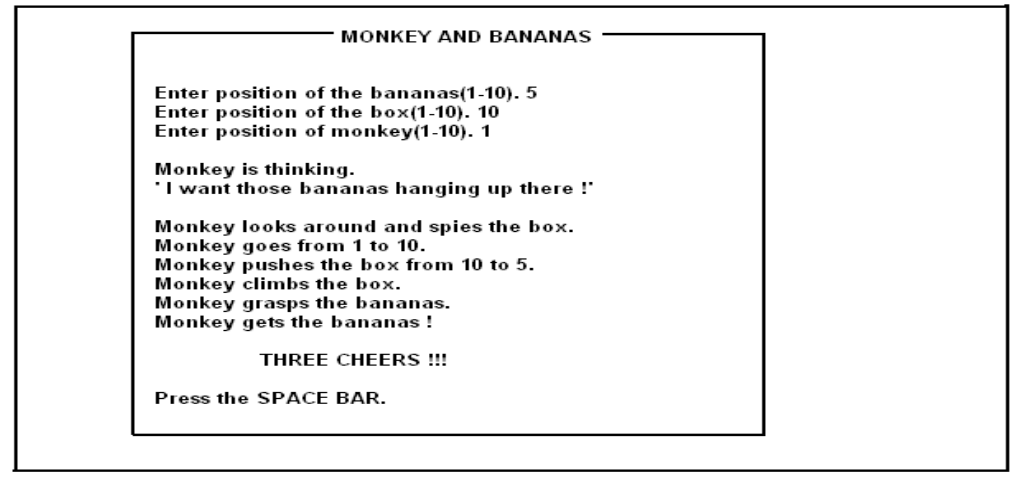


Рис. 2.25. Сеанс із програмою "Мавпа і банани"

Задача "Мавпа і банани" розв'язується за допомогою спеціальної системи відображення місць розташування предметів у просторі, тобто стан системи (мавпа, коробка і банани) відображається в термінах положення мавпи, коробки і бананів. Для цієї задачі стан визначається параметрами:

`стан(B, Bo, M, C, G),`

де **B** – розташування бананів, **Bo** – розташування коробки, **M** – розташування мавпи, **C** – визначає, чи стоїть мавпа на коробці, а **G** – визначає, чи схопила мавпа банани (0 – не схопила, 1 – схопила). Початковий стан такий:

`стан(5, 10, 1,0,0),`

а кінцевий стан має вигляд

`стан(5,5,5, 1, 1).`

Правила Турбо-Прологу в цій програмі – це оператори, що діють над простором станів, перетворюючи їх на нові. Для цієї задачі початковий стан обробляється цими правилами, для того, щоб перевести всю систему в кінцевий стан. На рис. 2.26 показана діаграма можливих станів задачі.

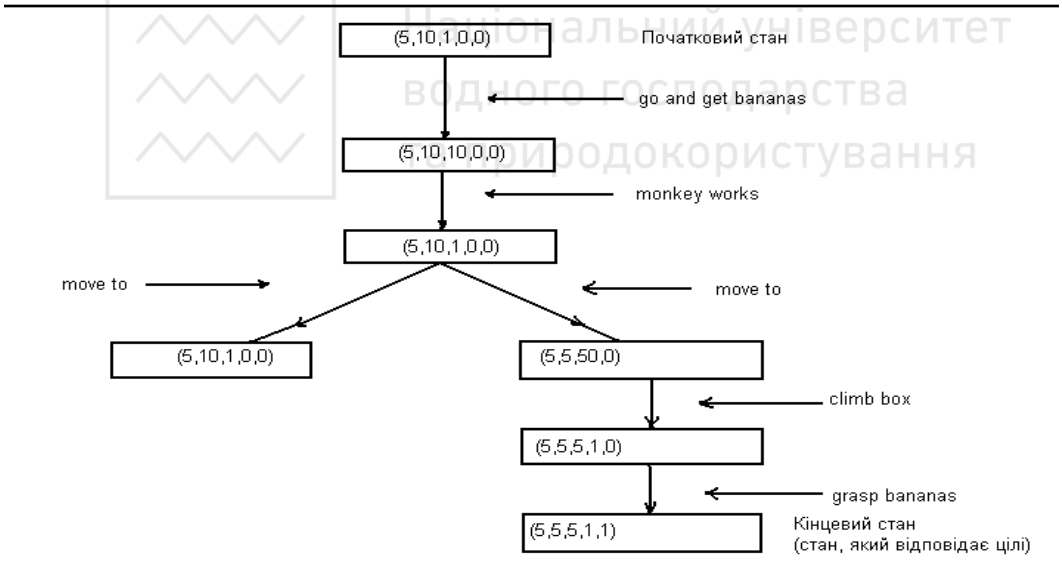


Рис. 2.26. Діаграма стану для задачі "Мавпа і банани"

Метод, який використаний у програмі "Мавпа і банани", є прикладом зворотного висновку. Програма починається з мети і рухається назад за допомогою підцілей, що виникають під час уніфікації. У Пролозі ліва частина правила (голова правила) є метою. Права частина правила (тіло правила з підцілями) використовується для порівняння з даними в базі даних. Якщо

процеси зіставлення виявляються успішними, то ліва частина правила також успішна. З цієї причини системи зі зворотним виведенням називаються системами, керованими метою.

2.9.3. Контрольні питання та вправи

1. Порівняльна характеристика функціональних можливостей простої та інтелектуальної гри "23 сірники".

2. Запропонувати стратегію виграшу людини з ймовірністю 40%.

3. Суть методу зворотного виведення під час створення комп'ютерних ігор.

4. Характеристика вбудованих предикатів Турбо-Прологу для роботи з графічними об'єктами.

5. Модифікувати програму "Мавпа і банани" так, щоб мавпі була необхідна ще й палиця для збивання бананів.

6. Програмним способом забезпечити супроводження гри "Мавпа і банани" у вигляді послідовності картинок, які відображають зміну місця знаходження мавпи.

7. Реалізувати програму відгадування задуманого числа із 100 можливих за мінімальну кількість кроків.

2.10. Основні тенденції розвитку мов логічного програмування

2.10.1. Методологія логічного програмування

Істотна відмінність логічного програмування від традиційного полягає в тому, що ми повинні описувати логічну структуру задачі, а не вказувати комп'ютеру послідовність дій, яку необхідно виконати для отримання розв'язку. Люди, які не мають попереднього досвіду роботи на комп'ютері, схильні вважати, що програмування є за своєю природою логічною задачею і часто, опанувавши процедурні мови програмування – такі, як Бейсік чи Паскаль, дуже здивовані і до певної міри розчаровані, коли переконуються, що це не так. Адже використання традиційних сучасних мов програмування для написання програм багато в чому залежить від внутрішніх вбудованих механізмів комп'ютера. З одного боку, це розумно, але з іншого не завжди відповідає початковому уявленню про механізм знаходження розв'язку задачі. Програмісти, які звикли мати справу з традиційними мовами

програмування, під час першого ознайомлення з декларативними мовами можуть зіткнутися з проблемами пристосування до логічного мислення. Бажання ефективно управляти ресурсами комп'ютера часто викликає почуття втрати, коли освоюються логічні мови (Пролог чи Лісп), які не є машинно-орієнтованими. Необхідно докласти певних зусиль для того, щоб подолати усталену звичку до одного способу розрахунків (наркотичної прихильності до оператора присвоювання) і навчитися виводити наслідки з довільної множини логічних речень на основі логічного інтерпретатора. Фактично у будь-якій іншій (не логічній) мові програмування для кожної нової задачі необхідно заново писати програму, яка буде розв'язуватися на основі заданого обсягу знань. Відсутність гнучкості процедурних програм у разі зміни мети значно знижує продуктивність програм. Водночас ми завжди отримуємо логічний розв'язок задачі на основі логічних речень, що зберігаються в пам'яті комп'ютера у разі застосування механізму логічного висліду.

Відзначимо, що було б неправильним хибне враження про те, що логіка звільняє програміста від прагматичних міркувань. В реальних застосуваннях для досягнення максимальної ефективності програми часто виникає потреба в структуруванні вхідних речень, належного урахування дедуктивної стратегії інтерпретатора та конкретного вигляду сформульованого логічного запиту. Програмісту необхідно пам'ятати і про алгоритмічні аспекти, і про логічні властивості створюваних ним програм. Тим не менше, треба зазначити, що твердження програми мовою логіки завжди описують задачу, а не процес її розв'язання: будь-які гіпотези, твердження, критерії чи алгоритми завжди безпосередньо зрозумілі з тексту програми.

Методологія програмування охоплює ті аспекти розробки програмного забезпечення, які зв'язані з безпосереднім складанням програм для комп'ютерів. Якщо для більшості професій знаряддя праці створюються для задоволення практичних інтересів, то в галузі програмування спостерігається зворотний процес: мови програмування та інструментальні засоби визначалися здебільшого особливостями архітектури комп'ютера (колись – електронної обчислювальної машини), а не пошуком найкращих способів подання користувачем числових задач та методів їх розв'язання. Ця традиція зумовлена ще й обмеженими технічними та економічними можливостями у виробництві логічних схем.

Традиційні комп'ютери, до яких зараховували перші чотири покоління комп'ютерів, називають машинами фон Неймана, визнаючи їх залежність від

моделі механічного розрахунку. В такій моделі машинна операція є послідовністю переходів станів, що впливають на дискретні комірки пам’яті, причому кожен такий перехід заміняє поточний стан комірки на новий. Мета розрахунків в тому, щоб перетворити деякий початковий стан в деякий кінцевий, змушуючи машину виконувати послідовність вказаних переходів. Програма і визначає цю послідовність переходів і сама зберігається в пам’яті комп’ютера. Отже, програма є послідовністю дискретних вказівок, які детально описують послідовність переходів станів комірок пам'яті. Майже всі широковживані мови сьогодні є фон-нейманівської структури: Паскаль і Бейсік, Ада і Фортран, Кобол і мови символічного Асемблеру. Єдина принципова особливість всього спектра мов фон-нейманівської структури – це ступінь, в якому описання переходів станів не прив’язане до архітектури реальної машини.

Переваги, які забезпечують мови декларативного програмування, належно оцінені спеціалістами в царині штучного інтелекту і відіграють істотну роль в таких масштабних проектах, як проєкт створення комп’ютерів п’ятого покоління (Японія), програма комітету Альві (Великобританія), програма ESPRINT (Європейське Співтовариство). Зміщення акцентів до декларативних мов відбулося завдяки усвідомленню необхідності підвищувати строгість і продуктивність програмного забезпечення, розробляти нові архітектури обчислювальних машин та значним здобуткам в галузі обчислювальної логіки і штучного інтелекту.

Оцінку основних можливостей декларативних мов програмування та відповідних їм системних структур подано в роботі Дарлнгтона і Ковальського, яка опублікована в 1983 році. Однією з істотних особливостей, які відрізняють ці мови від процедурних , є їх двояка функція: широкий набір моделей подання знань для розв’язання прикладних задач та розрахунків, а також притаманний процедурним мовам паралелізм і максимальна пристосованість до систем програмного забезпечення, які піддаються постійним модифікаціям. Ці особливості декларативних систем обіцяють великі можливості для обчислювальної діяльності. До них належать комерційні програми опрацювання даних; природні і штучні мови, які прямо приводять до декларативного описання; графічні і людино-машинні інтерфейси; декларативне подання об’єктів, складових частин, планів і цілей проектування, які дають змогу оптимізувати процедуру автоматизації проектування і виробництва.

2.10.2. Напрямки застосування мов логічного програмування

2.10.2.1. Інтелектуальні системи, що ґрунтуються на знаннях (ІС)

Інтелектуальні системи – це такі системи, в яких механізми інтелектуальних міркувань застосовують до явних моделей подання знань. Прикладами таких систем є системи баз даних, системи, що ґрунтуються на правилах, експертні системи. Спеціалісти визначають галузь цих досліджень як прикладний штучний інтелект. Технологія розроблення ІС була виділена в проекті Альві (Великобританія) як одна із чотирьох технологій, яка необхідна для експлуатації в повному обсязі п’ятого покоління комп’ютерних систем. До останніх трьох технологій належить людино-машинний інтерфейс (ЛМІ), надвеликі інтегральні схеми (НВІСи) і технологія програмного забезпечення. Відзначимо, що всі ці технології знайшли своє місце у сфері прикладного застосування

Відзначимо, що інтелектуальні системи, що ґрунтуються на знаннях, неможливо побудувати лише на основі створення бази знань з відповідним механізмом опрацювання інформації для неї. Хоча правомірно назвати будь-яке складання прологівської програми прикладом програмування, що ґрунтується на знаннях. Тим не менше неправильно вважати, що будь-яка Пролого-подібна реалізація програми приведе до ІС. Інтелектуальна обробка інформації передбачає істотніші механізми, ніж дають базисні засоби побудови логічних висновків і пошуку. Як мінімум, для створення ІС необхідні стратегічні механізми (евристика), які дають змогу скоротити непотрібний пошук, використовуючи, наприклад, професійну обізнаність про клас досліджуваних задач. Інтелектуальні стратегії можуть вбудовуватися безпосередньо в інтерпретатор, що дає значний ефект, коли ці стратегії є універсальними і застосовними до широкого класу задач. З іншого боку, прикладні програми об’єктного рівня можуть викликатися за допомогою програм метарівня, які містять описання стратегій і які можуть бути, якщо необхідно, видаленні їх інтерпретатором і перепрограмуванні відповідно до обставин. Такий підхід дає змогу вповні використати потужні можливості об’єктно-орієнтованої мови та метамови. Слоумен, відомий спеціаліст в галузі штучного інтелекту, відзначає, що ІС набуватимуть все більшого значення, оскільки вони не тільки вводять в практику відомі методи штучного інтелекту, а й дають змогу зробити певні узагальнення, пов’язані з побудовою моделей подання знань та розвитком інтелекту. Ці системи є інструментальним засобом для дослідження таких задатків, як вміння робити

відкриття, здібності до творчої діяльності, самоаналізу і самовдосконалення, навчання і узагальнення. На нашу думку, ці системи доцільно експлуатувати для низки інших структур подання знань, а не тільки логіки.

2.10.2.2. Системи баз даних (СБД)

Найчіткішим застосуванням ІС, якому нині приділена значна увага, є їх використання в системах баз даних. У звичайних СБД дані традиційно розглядають як множину співвідношень, які екстенсивно зберігаються у вигляді таблиці.

Поява реляційних баз даних уможливила створення багатьох запитів завдяки реляційним розрахункам, в яких є стандартні оператори, такі, як оператор з'єднання і проектування.

Можливості застосування логічного програмування до подання даних і опрацювання запитів досліджені в ранніх роботах Емдена, Ковальського і Тернлунда. Саме вони встановили факт відповідності процедури пошуку даних в моделі традиційної бази даних стандартним механізмам побудови висновків в логічних інтерпретаторах.

У реляційній базі даних, сформульованій мовою логіки, пошук кортежів для відповіді на запит стає процесом доведення теорем, в яких база даних розглядається як множина припущень, а запит (деяке цільове твердження) – як теорема. Пошук кортежів (можливих розв'язків) стає автоматизованим способом доведення теорем. В логіці не роблять різниці між даними, поданими у вигляді фактів і даними, поданими у вигляді загальних правил. В ній також не розрізняються правила, які трактуються як дані, і правила, які трактуються як процедури, що забезпечує можливість логічного описання всієї задачі: формування структур даних, введення фактів, правил і процедур, застосування механізмів перегляду, поповнення, видалення, повернення, відсікання та логічного виведення. Отже, мова логіки дає змогу подолати штучні відмінності між мовами програмування, мовами запитів і мовами описання даних.

Фундаментальна невідповідність, що існує між реляційними базами даних і підходом, що ґрунтується на доведенні теорем, розглянута в роботах Нікола і Галера в 1978 році. За другим підходом, де використовують узагальнені правила, база даних розглядається як теорія, яка дає змогу робити широкий спектр інтерпретацій. Водночас в першому варіанті під час роботи з таблицями основних кортежів вона трактується як одна інтерпретація. Характерні недоліки реляційного підходу стають очевидними внаслідок тих

ускладнень, що виникають у разі організації рекурсивного доступу та під час опрацювання неповністю визначених (частково означених) кортежів в запитах та відповідях.

Перевага логіки веде до відмови від використання реляційних обчислень і ведуться активні пошуку того, як змусити логіку відповідати тим вимогам високого рівня, які ставить методологія СБД. Основний напрямок цих зусиль пов'язаний з оптимізацією запитів та їх аналізом. Загальноприйнята мова запитів в СБД може бути придатною для використання непрофесіоналами, які не зобов'язані мати справу з механізмами збереження і пошуку даних. З цього аспекту запит трактується як специфікація, яка не має процедурних якостей. Завдання розробника – в оптимальному формулюванні запитів до бази даних. Проблема у разі стандартного формулювання запитів в тому, що Пролог сліпо впорядковує підцілі в цільовому твердженні, що може виявитися доволі неефективним під час розгляду великих баз даних і повторного використання параметра в запиті. Частково ця проблема може бути розв'язана плануванням запитів (впорядкуванням підцілей) та оптимізацією запитів, яка забезпечується взаємною незалежністю підцілей.

Для організації ефективної вибірки даних, поданих у вигляді диз'юнктів, доцільно використовувати методи індексації. Цей підхід ефективніший для СБД, ніж для логічних програм. Методи ефективного індексування, зокрема, багато – ключового кешування розглядаються в статті Ллойда, яка опублікована в 1983 році.

Окрім цих прагматичних проблем залишаються відкритими ціла низка теоретичних проблем, пов'язаних з використанням логіки мови баз даних. Добре відома проблема фреймів, яка виникає у разі подання часу, стану і можливих змін, а також взаємодії немонотонних змін (міркування над неповністю визначеними даними за замовчуванням). Ці проблеми розглядають в цілій низці робіт Ковальського, Сергота, Мінера упродовж останніх двадцяти років.

2.10.2.3. Експертні системи (ЕС)

Експертна система (ЕС) – це система, яка моделює людські знання і досвід в деякій конкретній предметній галузі, для якої неможливе строге математичне описання. ЕС є різновидом ІС і володіють великою базою знань, складених з фактів, правил і евристичних тверджень, що належать до предметної галузі, а також можливостями в діалоговому режимі спілкуватися

зі своїм користувачем майже на рівні людини-експерта. Окрім цього, передбачається можливість роз'яснення всіх рекомендацій, розв'язків, порад, що видаються користувачу, а також система має здатність до самовдосконалення на основі такого спілкування. Детальне описання властивостей та напрямків застосування експертних систем наведено в книзі Мічі, яка опублікована в 1979 році.

Експертні системи мають очевидну і надзвичайно привабливу галузь застосувань мовою логічного програмування. Їх необхідно відрізняти від простих систем, що ґрунтуються на використанні правил, яким часто приписують властивості експертних тільки на підставі того, що вони містять факти об'єктного рівня і правил, що адекватно відображають деяку підмножину експертних знань. Відмінність полягає в тому, що експерт-людина, знаючи факти і правила, що стосуються проблемної галузі, крім того, розуміє, як ними ефективно скористатися.

Отже, експертні системи не піддаються повній формалізації і тому ймовірніше чекати, що введення експертних знань в програму для комп'ютера, на противагу точній розробці добре специфікованих звичайних програм, буде відбуватися методом проб і помилок. В міру того, як база знань розвивається і до системи ставляться нові вимоги, в неї повинні вноситись відповідні корективи. За своєю природою логічне програмування найкраще пристосоване до розв'язування задач виявлення суперечностей (наприклад, за допомогою процедур спростування) і неповноти (наприклад, за допомогою правил виводу за замовчуванням) для мотивації необхідності удосконалення експертної системи. Водночас метамова логічного програмування забезпечує автоматичне описання критеріїв і механізмів для вдосконалення.

Експертні системи повинні володіти системою логічного виведення як базису для обробки запитів, можливістю поповнювати базу знань тою інформацією, котра була уже отримана, можливістю пояснювати користувачу, як і чому було одержано кожен результат, можливістю отримувати знання від людини (принцип діалогової симетрії). Карк і Маккейба реалізували ці вимоги додаванням спеціальних управляючих можливостей до правил об'єктного рівня, що містяться в базі знань. Надалі складніший спосіб реалізації вказаних вимог за допомогою метамови був описаний Хаммондом і Серготом. Система реалізована мовою мікро-Пролог і дає доволі гнучкий інструмент, що називається APE-the-User і призначений для побудови експертних систем. Після цього система розглядає експерта як розширення наявних в ній внутрішньої бази знань і дає йому запитання, щоб

отримати додаткову інформацію про вказане відношення, яка потім буде зберігатися в базі знань. Отже, відбувається кооперація системи і експерта для побудови бази знань. Запити користувача на об'єктному рівні можуть оброблятися безпосередньо звертанням до інтерпретатора мікро-Прологу, який міститься в ядрі системи, в той час як різноманітні запити типу "як", чому", "чому ні", що вимагають пояснення, обробляються зверненням до інтерпретатора метарівня. Він, своєю чергою, будує і видає пояснення у вигляді відредагованих доведень. В системі передбачені також засоби для структурованих запитів природною мовою. Поєднання всіх цих можливостей створює сприятливе універсальне середовище для побудови, модифікації та експлуатації експертних систем.

Система APES була використана з великим успіхом для автоматизації юридичних експертних систем. В 1981 році система APES була застосована для кодування ухваленого закону про громадянство в Великобританії, котрий визначає категорії британського громадянства і містить правила, необхідні для власної інтерпретації.

2.10.2.4. Опрацювання інформації природною мовою

Обробка повідомлень природною мовою відіграє дуже важливу роль в розробленні засобів для забезпечення людино-машинного інтерфейсу і, зокрема, в побудові зовнішніх рівнів інтелектуальних систем.

Для реалізації природної мови на комп'ютері потрібно формалізувати як її синтаксис, так і семантику.

Більшість із властивостей природної мови найзручніше подавати за допомогою контекстно-залежних граматик, що містять правила, з допомогою яких певні структури виразів класифікують залежно від контексту їх входження в речення природної мови, а не лише залежно від їх будови. Вперше логіка хорнівських диз'юнктів для представлення контекстної залежності була використана Колмероє (1978) для розробки "метаморфозних граматик", для яких граматики певних диз'юнктів утворюють нормальну форму. Він показав, що хорнівських диз'юнктів достатньо для вираження довільної контекстно-вільної граматики (КВГ). Проблему задання структури речень природної мови можна формулювати як цільові твердження, а різні процедури доведення, що використовують в логічному поданні природної мови, відповідають різним стратегіям синтаксичного аналізу.

Найзагальнішою структурою подання, що пристосована як до природно-мовних так і до інших конструкцій є семантичні мережі. Деліяні і

Ковальський показали, що традиційне формулювання семантичних мереж можна було би узагальнити для подання речень у вигляді диз'юнкт. Такі системи забезпечують доволі компактне і типове подання і до них можна застосовувати операційні схеми видобування інформації.

Логічне програмування може використовуватися і для навчальних потреб, зокрема, для ознайомлення користувачів з обчисленнями на комп'ютері, теорією ігор, графічними побудовами, розв'язування ребусів і головоломок. Його також можна використовувати для збагачення викладання всіх академічних предметів із шкільного чи вузівського навчального плану. Логіка є єдиною академічною дисципліною, спільною для всіх навчальних предметів, оскільки вона скрізь сприяє викладанню та розумінню матеріалу.

Логіку, як правило, легше сприймають ті, в кого немає попереднього досвіду роботи з комп'ютером ніж ті, хто звик мати справу з традиційними системами програмування. Того, хто упродовж десятків років звик до процедурного формалізму і заздалегідь визначених стандартів, треба переконувати, що логіка має привабливі перспективи. Власне не набір вдало підібраних прикладів, які демонструють фантастичні можливості логіки (пошук континууму розв'язків, властивість оберненості), а загальні принципи методології програмування, колосальна важливість прикладних застосувань, пов'язаних з базами знань і відповідна експлуатація майбутнього нового покоління архітектури обчислювальних машин – ось ті проблеми, які є вмотивованим доказом реального інтересу до логіки.

2.10.3. Логіка – не фоннеймановська мова програмування

Основна властивість, завдяки якій логіка та інші декларативні мови програмування стають незалежними від неймановського поняття розрахунків, є їх семантична незалежність від стратегії їх виконання. Якщо якась задача розв'язується на комп'ютері за допомогою логічної програми, то це наслідок, який може бути виведений з логічного змісту програми і він ніяк не пов'язаний зі способом дослідження програми комп'ютером.

Стратегія Прологу була задумана так, щоб з урахуванням відносної простоти його реалізації дати програмісту деякі можливості для керування вибором викликів і набором процедур в умовах експлуатації єдиного процесора. Практичні зусилля з реалізації вільнішої стратегії пов'язані з використанням управління потоком даних у співпрограмному режимі. Ця схема розрахунків реалізується за рахунок використання загальних змінних у викликах, а не передбаченої заздалегідь процедурі або неявно заданої

послідовності керування. Такий співпрограмний режим реалізований в системі IC-Пролог, який описаний Кларком, Макейбом і Григорі в 1982 році і розроблений на основі досліджень Стівенсона схем лінивого розрахунку в Пролозі [11]. В IC-Пролозі кожна із змінних може бути анотована одним із символів "?" або "^", які вказують на спосіб виконання виклику в точці його активації. За наявності у виклику анотації x? управління переходить до цього виклику, якщо на деякому кроці змінній x присвоюється якесь значення або значення присвоюється якійсь іншій змінній, що входить в терм зі змінною x. Інтерпретатор рухається цим шляхом доти, доки не побачить, що на наступному кроці виконання змінній x необхідно передати якісь дані. Після цього виконання розрахунків припиняється, а керування повертається до попередньо відкладеної точки. Друга можлива анотація x^ означає, що керування одразу переходить до виклику, що містить цю змінну, призупиняє поточні розрахунки щоразу, як тільки на наступному кроці виконання для змінної x необхідно передати якісь дані. Пізніше керування повернеться до цієї точки, як тільки необхідні дані успішно будуть передані x. Отже, дія цих двох анотацій немов би доповнює одна другу. Якщо говорити неформально, то така стратегія розрахунків дає змогу перестрибувати з одного частково закінченого обчислення на інше відповідно до потоку даних через анотовані змінні.

Оскільки виклик з анотацією x? поповнює дані (принаймні частково), то його називають "шаленим споживачем", а виклик з анотацією x^ дає змогу передавати додаткові дані для x, то його називають "лінивим виробником". Протокол споживачів-виробників лежить в основі більшості моделей виконання задачі у спів-програмному режимі під керівництвом потоку даних. Формуючи різними способами виклики програми, можна отримати широкий спектр методів виконання від "лінивого" виробництва даних до їх активного споживання.

Виконання під керівництвом потоку даних – це лише один із способів реалізації паралелізму між індивідуальними процесами. Можливості для паралельного виконання логічних програм можна розділити на кілька категорій. До першої категорії належить АБО-паралелізм, в якому використовується той факт, що на активний виклик відповідають одразу кілька процедур і їх можна застосувати паралельно для дослідження цього виклику. Отримані результати паралельних обчислень можна будувати незалежно, окрім випадку, коли в них є посилання на одні і ті самі незв’язні змінні у спільному батьківському середовищі, внаслідок чого виникають

конкуруючі присвоювання. Щоб позбутися цієї завади, можна застосовувати спеціальні схеми організації стеків.

До другої категорії належить І-паралелізм, в якому використовується той факт, що поміщені в запит виклики можна розраховувати паралельно. Для цього необхідний набір паралельних процесорів з встановленням пріоритетів зв'язування. Ефективність таких моделей істотно залежить від точності керування передаванням даних і синхронізацією процесів. Незалежний І-паралелізм можливий лише якщо паралельні виклики не містять спільних змінних. За наявності спільних викликів для ефективності паралельних розрахунків можна один з І-паралельних викликів в наказовому порядку призначити виробником цієї змінної, а всі інші споживачами.

До третьої категорії паралельних обчислень належить паралелізм з використанням потоків або "конвейєрне опрацювання структур". У такому разі виклику дозволяється розпочати споживання даних, які вироблені іншим викликом за умови, що перший виклик отримає певну підструктуру цих даних. Тим часом виклик-виробник продовжує породження наступної структури даних. Власне кажучи, це особливий спосіб поводження з І-паралелізмом, який дає засіб чіткої координації спільних змінних на двосторонньому рівні.

Зазначимо, що за твердженням Тіка і Уоррена за умови уніфікації процедур послідовного Прологу потенціальна продуктивність зростає як мінімум в 20 разів порівняно з ефективністю реалізації Прологу для машин DEC-10. Для реалізації цього достатньо розкласти логічні програми у вигляді інструкцій Прологу високого рівня на суміщене в часі виконання мікрокоманд, які опрацьовуються засобами конвейєрного паралелелізму.

2.10.4. Проект створення комп'ютерів п'ятого покоління

В 1979 році міністерство зовнішньої торгівлі і уряд Японії санкціонували унікальний захід в галузі обчислювальної техніки, який називають проектом створення комп'ютерів п'ятого покоління. Мета проекту, яка розрахована на десятиріччя вперед полягає в поєднанні високопродуктивної паралельної архітектури обчислювальних машин, відмінної від архітектури фон Неймана з алгоритмами і процедурами опрацювання знань, розробленими в галузі штучного інтелекту. Роботи з цього проекту розпочалися з весни 1981 під керівництвом Кацухіро Фучі в Інституті нового покоління обчислювальної техніки (ICOT) в Токіо. Комп'ютери п'ятого покоління повинні володіти здатністю до

широкомасштабного набуття знань, самонавчання, розмірковування і розв'язання задач, взаємодії з людиною з орієнтацією на неї.

Попередні розробки цього проекту були подані в матеріалах конференції JIPDEC в 1981 і в подальших публікаціях у доволі абстрактному концептуальному вигляді. Комп'ютер цього покоління має моделювальну систему програмного забезпечення (МСПЗ), яка розміщується між цілим набором людських прикладних систем (ЛПС) та апаратною частиною машини (АЧМ). Для формування ЛПС передбачається використання широкого спектра мов, які орієнтовані на користувача: штучні мови, природні мови, графічні зображення. ЛПС зв'язані з МСПЗ за допомогою інтерфейсів, які здатні розуміти знання, поміщені в ЛПС та перетворювати їх у форму, придатну для інтелектуального синтезу та опрацювання. Ядром МСПЗ є інтелектуальна система програмування та набір систем баз знань для побудови моделей подання знань у формі, що відповідає поставленим цілям. Основна особливість інтерфейсу МСПЗ-ЛПС – наявність функції синтезу. Сама АЧМ містить машину для розв'язання задач і машину баз знань, які пристосовані до роботи з базами даних, символами та числових розрахунків. Продуктивність такої машини повинна досягати 10^7 клвс. Така швидкодія забезпечується наявністю 10^4 паралельних процесорів, які звертаються до пам'яті обсягом 10^4 Мгбайт.

Окрім претензійної програми, яка стосується апаратних засобів, велике здивування після опублікування планів створення комп'ютера п'ятого покоління викликав вибір мови логічного програмування як основного формалізму. Вибір цієї мови обґрунтований в статті Фурукави в матеріалах конференції JIPDEC в 1981. Це зумовлено тим, що логіка покриває як функціональне програмування, так і формальні системи реляційних баз даних і, разом з тим, дає однотипне подання даних, програм, специфікацій і понять метарівня, забезпечуючи зв'язки між цілями, що стоять перед технікою програмного забезпечення та штучним інтелектом.

Японський проект комп'ютерів п'ятого покоління стимулював прийняття урядових фінансових програм створення нового покоління комп'ютерів в Великобританії (Звіт комітету АЛЬВІ, 1984), Європі, США. Якщо роботи із створення нового комп'ютера наблизяться до поставлених цілей, то виклик, кинутий японцями спеціалістам в галузі комп'ютерних наук усього світу, буде надзвичайно серйозним. Питання стоятиме так: "топтати на місці чи йти вперед, оскільки іншого вибору немає", – заявив Кацухіро Фучі на конференції JIPDEC.

2.10.5. Контрольні питання та вправи

1. Дати порівняльну характеристику процедурних та декларативних мов програмування.

2. Дати взначення інтелектуальної системи, що ґрунтується на знаннях.

3. Основні вимоги до побудови експертних систем.

4. Особливості застосування мов логічного програмування до реалізації процедури спілкування з комп'ютером мовою, близькою до природної.

5.Спільні та відмінні властивості реляційних баз даних та баз даних мовою логіки.

6. Способи вдосконалення логіки для створення баз даних у вигляді фреймів.

7. Структурні елементи комп'ютера п'ятого покоління.

8. Технічні характеристики проекту комп'ютера п'ятого покоління.

9. Дати порівняльну характеристику декларативних та процедурних мов програмування. Описати їх принципові відмінності.

Розділ 3. Лабораторні роботи мовою логіки

3.1. Лабораторна робота № 1. Робота в середовищі Турбо-Прологу

Мета: Набути практичних навиків роботи в середовищі Турбо-Прологу.

3.1.1. Теоретичні відомості

Турбо-Пролог – це здійснена компанією Borland International реалізація мови програмування високого рівня Пролог компіляторного типу. Його особливість – це висока швидкість компіляції і розрахунку. Турбо-Пролог призначений для давання відповідей, які він логічно виводить за допомогою своїх потужних внутрішніх вбудованих в систему процедур.

Пакет компілятора Турбо-Пролог складений із двох дистрибутивних дисків – вказівок користувача, що нараховують більш ніж 200 сторінок тексту. Турбо-Пролог може бути встановлений і запущений на персональному комп'ютері, який оснащений або вінчестерським диском, або двома дисководами для гнучких дисків. Необхідна наявність оперативної пам'яті не менше ніж 384К. На комп'ютері повинна бути встановлена версія операційної системи PC DOS з індексом 2.0 і вище.

Виклик Турбо-Прологу здійснюється наказом PROLOG. Комп'ютер запитує всі системні файли Турбо-Прологу і завантажує їх в оперативну пам'ять. Після активізації системи на екрані з'явиться початкова заставка Турбо-Прологу, яка складається з двох вікон. Текст верхнього вікна інформує про рік випуску і номер використовуваної версії Турбо-Прологу. Друге вікно містить інформацію про задану за замовчуванням конфігурацію Турбо-Прологу і містить вказівку на натиснення клавіші пробіл:

```
Press the SPACE bar.
```

Якщо це виконати, то система висвічує головне меню Турбо-Прологу, яке містить сім доступних користувачу опцій (команд). Ці команди належать до семи функцій Турбо-Прологу, якими є:

1. Запуск програми на виконання **(Run)**.
2. Трансляція програми **(Compile).**
3. Редагування тексту програми **(Edit).**
4. Задання опцій компілятора **(Options).**
5. Робота з файлами **(Files).**

6. Відлагодження системи відповідно до індивідуальних потреб **(Setup).**

7. Вихід з системи **(Quit).**

Перехід від команди до команди простий і зручний. Існує два способи задання команд. Перший вимагає натискання клавіші, що відповідає першій букві вибраної команди. Для закінчення роботи з командою використовується клавіша **Esc**. Другий спосіб полягає у пересуванні по меню за допомогою стрілок. Перехід до роботи з вибраною командою здійснюється за допомогою клавіші **Enter**.

Головне меню містить чотири вікна. У лівому верхньому куті мітиться вікно редактора Турбо-Прологу **(Editor),** в правому верхньому куті – вікно діалогу **(Dialog),** в лівому нижньому куті – вікно повідомлень **(Message),** в правому нижньому – вікно трасування **(Trace).** У разі використання кольорового монітора за замовчуванням для вікна редактора встановлюється блакитний колір, для вікна діалогу – червоний і чорний для вікон повідомлень і трасування. Робочим файлом за замовчуванням є файл з іменем **WORK.PRO**. **PRO** – це задане за замовчуванням розширення для файлів, що містить програми мовою Турбо-Пролог.

Щоб набути основних навиків у користуванні системою Турбо-Пролог розглянемо введення найпростішої програми, яку назвемо **welcome.pro**. Після виклику Прологу необхідно за допомогою стрілки перейти до команди головного меню **Edit** і натиснути клавішу **Enter.** В лівому верхньому куті вікна **Editor** з'являється миготлива риска – курсор редактора. Редактор готовий прийняти текст програми, який вводиться з клавіатури. Можна ввести такий текст програми **WELCOME.PRO**:

```
predicates
hello
goal
hello.
clauses
hello:- write("Welcome to TurboProlog!"), nl.
```

Коли рядок закінчується, то для переходу на наступний рядок необхідно натиснути клавішу **Enter**.

Для запуску введеної програми на виконання спочатку треба вийти з редактора; для цього необхідно натиснути клавішу **Esc**. Курсор редактора зникає, а курсор головного меню вказує на команду **Edit**. Якщо задати

команду **Run**, то результати трансляції з'являються у вікні **Message** і результати виконання програми у вікні **Dialog**. Перший рядок у вікні повідомлень вказує на те, що розпочалася трансляція програми **WELCOME.PRO**. Отже, трансляція задається автоматично, без задання спеціальної команди **Compile**. У разі задання команди **Run** програма транслюється в оперативну пам'ять. Другий рядок у вікні сигналізує про трансляцію предиката **hello**.

Для того, щоб зберегти програму, тобто записати її на диск, необхідно вибрати команду **Files** і підкоманду **Save** (або натиснувши комбінацію клавіш **Alt** і **S**, або використовуючи стрілки). В результаті цих дій на екрані з'явиться вікно, в якому буде ім'я, що задане раніше. Ім'я файла можна відкоригувати або залишити без змін, натиснути клавішу **Enter**.

Для того, щоб завантажити у вікно редактора вже інший файл, необхідно вибрати команду **Files** головного меню і підкоманду **Load**. Якщо у відповідь на запит імені файла натиснути клавішу **Enter**, то в спеціальному вікні буде висвітлений весь перелік файлів поточної директорії з розширенням .**PRO**. Тепер, використовуючи стрілки, можна добратися до будь-якого файла. Якщо є потреба ввести ім'я файла з клавіатури, то розширення .**PRO** задавати немає необхідності, оскільки за замовчуванням вважають, що файл має розширення .**PRO**.

Працюючи з редактором Турбо-Прологу, можна отримати інформацію про будь-яку його команду; для цього досить натиснути функціональну клавішу **F1**. На екран· з'являється невелике вікно підказки, яке містить короткий перелік команд редактора та іншу корисну інформацію. Натиснувши комбінацію **Shift-F10**, це вікно можна розширити до розмірів повного вікна; повторне натискання цієї комбінації повертає вікно у попередній стан.

Якщо необхідно роздрукувати текст файла, що міститься у вікні, використовується підкоманда **Print** команди **Files**, яку можна викликати згідно з вищенаведеним описанням.

3.1.2. Послідовність виконання роботи

1. За курсом лекцій повторити матеріал про роботу в середовищі Турбо-Прологу.

2. Ввести в Пролог-систему програму, яка б у першому рядку видала на друк прізвище, ім'я та по батькові студента, а в другій – його дату

народження.

3. Відтранслювати програму і запустити на виконання.

4. Проекспериментувати з програмою, використовуючи команди головного меню Прологу, щоб засвоїти навики роботи з Пролог-системою.

5. Здійснити запис програми в файл та роздрукувати її.

6. Оформити звіт по роботі.

3.1.3. Контрольні питання

1. Основне призначення Турбо-Прологу.
2. Суть команд головного меню та призначення його вікон.
3. Способи задання команд головного меню.
4. Послідовність запису програми в файл та її роздрук.
5. Як змінити ім'я програми, не виходячи з середовища Турбо-Пролог?
6. Як вибрати задану програму з бібліотеки програм з розширенням **.PRO** та очистити вікно екрана від тексту?

3.2. Лабораторна робота № 2
Введення фактів і правил на Пролозі

Мета: Освоєння техніки запису та введення фактів і правил мовою Пролог.

3.2.1. Теоретичні відомості

Пролог – це мова програмування, яка зводиться до розв'язання задач, зв'язаних з об'єктами і відношеннями між об'єктами. Наприклад, коли ми говоримо: "Петро має книгу", тим самим ми сповіщаємо, що між одним об'єктом "Петро" і другим "книга" існує відношення володіння. Причому це відношення має певний порядок: Петро має книгу, але не книга має Петра. Коли ми формуємо запит: "Чи має Петро книгу?", то нас цікавить правильність саме цього відношення.

Під час констатації деяких відношень нас цікавить лише певна частина об'єктів, яка необхідна для виконання заданої програми. Тобто ступінь деталізації програми залежить від того, що ми хочемо змусити робити комп'ютер.

Для описання відношень між об'єктами в Пролозі використовують

правила. Наприклад, правило "Дві особи є сестрами, якщо вони обидві жіночого роду і мають спільних батьків", пояснює, що означає бути сестрами. Варто відзначити, що хоча правила можуть бути спрощеними, тим не менше їх можна використовувати як визначення, якщо вони достатні для досягнення поставленої мети.

Написання найпростішої програми мовою Пролог можна задати у вигляді таких етапів:

- оголошення деяких фактів про об'єкти і відношення між ними;
- визначення деяких правил про об'єкти і відношення між ними;
- формулювання запитів про об'єкти і відношення між ними.

Оголошуючи факти, треба дотримуватися таких правил:

1) Імена всіх об'єктів і відношень записують з малої букви.

2) Першим записується ім'я відношення. Далі через кому записують імена об'єктів, а весь список імен об'єктів беруть в круглі дужки.

3) Кожен факт беруть в дужки.

Приклади фактів з можливою їх інтерпретацією природною мовою:

```
цінний (золото).                Золото є цінним.
жінка (марія).                  Марія є жінкою.
батько (петро, марія).          Петро є батьком Марії.
дає (петро, книга, марія).      Петро дає книгу Марії.
```

Формулюючи запити до фактів, необхідно дотримуватися тих самих правил, що і оголошуючи факти, за винятком необхідності ставити перед запитом спеціальний знак. Цей знак складається із знака запитання і риски, яка ставиться після нього.

Якщо б вищенаведені факти були введені в Пролог-систему, то на сформульовані нижче запити відповіді були би такими:

```
? - цінний (золото).           Чи золото є цінним?
так
? - жінка (марія).              Чи Марія є жінкою?
так
? - батько (петро, марія).      Чи Петро є батьком Марії?
так
```

```
? - дає (петро, книга, марія). Чи Петро дає книгу Марії?
Так
```

Для позначення об'єктів, які повинна визначити Пролог-система, використовують поняття змінної. Кожне ім'я, яке починається з великої букви, розглядається як змінна. Це дає змогу формулювати запити у вигляді:

```
? - подобається (петро, X). /* Чи подобається хто-небудь Петру? */
```

Зазначимо, що між послідовністю знаків **/***та ***/** можна поміщати будь-які коментарі.

Ім'я змінної може бути як завгодно довгим. Отримавши запит, Пролог-система порівнює його з базою даних і здійснює конкретизацію відповідної змінної першим знайденим фактом. Для продовження пошуку в базі даних необхідно натиснути клавішу ';'. Після натискання клавіші **RETURN** Пролог припиняє подальший пошук в базі. Для формулювання складніших відношень в Пролозі використовується поняття кон'юнкції цілей, коли в одному твердженні необхідно погодити кілька цільових рішень. Наприклад, якщо б ми хотіли для заданої бази даних в Пролозі сформулювати запит: **"Чи подобається Петро і Марія один одному?"**, то це можна зробити так:

```
? - подобається (петро, марія), подобається (марія, петро).
```

В кон'юнкції через кому можна записувати скільки завгодно цільових рішень. Пролог здійснюватиме перегляд бази даних, погоджуючи цілі зліва направо.

Правила в Пролозі використовують тоді, коли необхідно сказати, що деякий факт залежить від групи інших фактів. Правило – це деяке загальне твердження про об'єкти і відношення між ними. В Пролозі правило складається із заголовка і тіла правила. Заголовок і тіло з'єднуються за допомогою символу ":-", який читається "якщо". Для прикладу запишемо правило **"бути сестрою"** з двома аргументами – змінними X і Y. Тоді

природною мовою X є сестрою Y, якщо:

1) X є жінкою;

2) X має матір M і батька F;

3) Y має тих самих батьків, що і X.

Мовою Пролог це правило матиме вигляд:

```
бути_сестрою (X,Y):- жінка (X), батьки (X,M,F), батьки (Y,M,F).
```

3.2.2. Послідовність виконання роботи

1. Повторити за курсом лекцій теоретичний матеріал, пов'язаний з записом фактів, правил та запитів мовою Пролог.

2. Створити базу даних мовою Пролог про свою родину: батьків, сестер, братів, дядьків, тіток.

3. Записати мовою Пролог правила визначення предикатів: бути_сестрою, бути_братом, бути_дядьком, бути_тіткою.

4. Сформулювати запити про кількість ваших сестер і братів та наявність родичів.

5. Ввести програму в комп'ютер та перевірити її працездатність.

6. Оформити звіт про роботу.

3.2.3. Контрольні питання

1. Правила оголошення фактів про об'єкти.
2. Формулювання запитів стосовно фактів.
3. Роль змінних та способи їх конкретизація.
4. Поняття про кон'юнкцію цілей.
5. Подання відношень за допомогою правил.
6. Загальна схема пошуку з механізмом повернення.
7. Способи пошуку всіх відповідей на запит та виходу з програми.

3.3. Лабораторна робота № 3. Програма підтримки діалогу на англійській мові з використанням рекурсивно означуваних предикатів

Мета: Опанувати спискову структуру подання даних мовою логічного програмування та техніку побудови рекурсивно означуваних предикатів.

3.3.1. Теоретичні відомості

Списки – широко вживана структура даних в галузі нечислового програмування.

Список – це впорядкована послідовність елементів, яка може мати довільну довжину. Послідовність елементів є істотною. Елементами списку можуть бути будь-які терми: константи, змінні, структури, які містять і інші списки. Більше того, списки дають змогу подавати будь-який вид структури, який може знадобитися під час символьних обчислень. Ці властивості особливо корисні, коли заздалегідь невідома і довжина списку, і інформація, яка в ньому буде розміщена. Списки широко використовують для зображення дерев синтаксичного розбору, граматик, карт міст, математичних об'єктів, таких, як графи, формули чи функції.

Список – це будь-який порожній список, що не містить жодного елемента або структура, що має два елементи – голову і хвіст. Кінець списку, як правило, є хвостом, котрий є порожнім списком. Порожній список зображають як [] − квадратна дужка, яка відкриває, за якою йде дужка, яка закриває. Голова і хвіст списку є компонентами функтора, що позначається крапкою '.'.

Список з одного елемента це `.(a,[])`. Аналогічно, список, складений з атомів `a,b,c,` це

```
.(a,.(b,.(c,[])))
```

В Пролозі передбачена і інша, так звана дужкова форма подання списку. Так, вищезгадані списки можна подати у вигляді:

```
[a] ,  [a, b, c].
```

Списки можуть містити як елементи інші списки або змінні. Так, в Пролозі є наступні списки:

[ ]

[конкретна, людина, [любить, ловити, рибу]]

[a, v, d, [Z, Y]]

Робота над списками ґрунтується на розщепленні їх на голову та хвіст. Голова списку – це перший аргумент функтора '.', а хвіст − це другий аргумент функтора '.'. Оскільки процедуру розщеплення списку на голову та хвіст -широко використовують в Пролозі, то введена спеціальна форма для представлення списку з головою **X** та хвостом **Y**. Це записується як **[X | Y]**, де для розділення голови і хвоста використовується вертикальна риска. При конкретизації структури подібного виду **X** зіставляється з головою списку, **Y** з хвостом, як позначено в наступному приклад·:

```
p([1, 2, 3]).
p [ця, кішка, сиділа ,[на, цьому, вікні]]
?- [ X | Y ].
X = 1   Y = [2,  3] ;
X = ця Y = [кішка, сиділа, [на, цьому, вікні]]
?- p([ _ , _ , _ , [ _ | X ]])
X = [цьому, вікні]
```

Існує ще одна галузь використання списків. Це представлення рядків літер для введення чи виведення тексту. Якщо рядок взято в подвійні лапки, то він буде представлений як список літер, що відповідають літерам рядка.

Як приклад використання спискового подання даних означимо предикат належності елемент **X** до списку **Y**. Виникає дві умови, які необхідно перевірити для встановлення істинності предиката. Перша умова говорить про те, що **X** є елементом списку **Y**, якщо **X** співпадає з головою списку **Y**. Мовою Пролог цей факт можна записати так:

```
належить(X,[X|_]).
```

Цю умову можна було б записати у вигляді правила:

```
належить(X,[Y|_]) :- X=Y.
```

Друга умова говорить про те, що **X** належить до списку за умові, що він входить у хвіст цього списку.

```
належить(X,[_|Y]) :- належить(X,Y).
```

Два цих правила в сукупності і визначають предикат для відношення належності і вказують Прологу, як переглядати список від початку до кінця, шукаючи деякий елемент списку. У такому разі маємо справу з рекурсивно заданим предикатом, оскільки він визначається сам через себе. Найважливіший момент, про який необхідно пам'ятати, будуючи рекурсивний предикат – це визначення граничних умов як способу використання рекурсії.

Для предиката **"належить"** є два типи граничних умов. Або об'єкт, який ми шукаємо, є у списку, або його там немає. Перша гранична умова розпізнається першим твердженням, яке приводить до припинення пошуку елемента в списку, якщо перший аргумент предиката **"належить"** збігається з головою списку, що відповідає другому аргументові. Друга гранична умова зустрічається, коли другий аргумент предикату **"належить"** є порожнім списком.

Як можна переконатися, що граничні умови будуть коли-небудь виконані? Для цього варто звернути увагу на те, як використовується рекурсія в другому правилі предикату **"належить"**. Зауважимо, щоразу, коли ми шукаємо відповідність для цільового предиката **"належить"**, відбувається звертання до того самого предикату, нова ціль формується для коротшого списку. Хвіст списку завжди є коротшим, ніж вихідний список. Очевидно, що рано чи пізно відбудеться одна з двох подій: або відбудеться порівняння з першим правилом для **"належить"**, або як другий аргумент **"належить"** буде заданий список, довжина якого 0, тобто порожній список. Як тільки виникає одна з цих ситуацій, припиняється рекурентне породження цілей для предиката **"належить"**. Перша гранична умова розпізнається фактом, який не викликає породження нових підцілей. Друга гранична умова не розпізнається ні одним з тверджень для **"належить"**, так що пошуку зіставлюваного елемента списку для цільового твердження **"належить"** закінчується невдачею. Це демонструє такий приклад мовою Пролог:

```
належить (X,[X | _]).
належить (X,[_ | Y]):- належить (X,Y).
? - належить (d,[a, b, c, d, e, f]).
так
? - належить (2,[3, a, 4, f]).
ні
```

Використання рекурсивних означень спискових структур даних широко використовується у Пролозі, однак треба бути обережним, щоб уникнути "зациклення" означень, як у такому прикладі:

```
батько (X,Y):- дитина (Y,X).
дитина (A,B):-  батько  (B,A).
```

Запит, який містить як цілі батько або дитина, призводить до циклу, який ніколи не завершиться, оскільки перебуваючи у ньому, Пролог не зможе знайти якісь нові факти. Пошук розв'язку виявляється безмежно довгим, і

немає ніякої можливості його закінчити. Звідси можна зробити такий висновок: не треба вважати, що коли задати Прологу всі факти і правила, що стосуються поставленої задачі, то Пролог їх знайде. Створюючи програму мовою Полог, увесь час треба уявляти, як здійснюється пошук в базі даних, і які змінні будуть конкретизовані, коли буде використане одне з означених правил чи фактів.

3.3.2. Послідовність виконання роботи

1. Повторити за курсом лекцій теоретичний матеріал про спискову структуру подання даних у Пролозі та побудову рекурсивно означуваних предикатів.

2. Написати алгоритм логічної програми для підтримки діалогу англійською мовою для заданого набору речень, вибравши для подання речень спискову структуру даних.

3. Перевірити працездатність програми, ввівши її в комп'ютер.

4. Сформулювати всі можливі запити до програми та отримати всі можливі варіанти відповідей.

5. Оформити звіт про роботу.

3.3.3. Контрольні питання

1. Дати визначення рекурсії.

2. Синтаксис спискової структури даних у Пролозі.

3. Побудувати предикат належності елемента до списку.

4. Описати предикат вибору першого елемента зі списку.

5. Особливості використання рекурсивно означуваних предикатів.

6. Основні напрямки використання спискових структур.

3.4. Лабораторна робота № 4. Розробка програми користування бібліотечним каталогом

Мета: Засвоїти суть механізму відсікання та набути навиків його практичного використання для написання програм.

3.4.1. Теоретичні відомості

Механізм відсікання, який широко використовується у Пролозі, дає змогу вказати, які з раніше зроблених вибірок не варто переглядати, повертаючись по ланцюгу погодження цільових тверджень. Включення такого механізму в програму забезпечує:

1) швидше виконання програми, оскільки не буде відбуватися зіставлення для цілей, про які відомо, що вони нічого нового в розв'язок не внесуть;

2) програма займає менше місця у пам'яті машини, оскільки немає необхідності запам'ятовувати точки повернення для подальшого аналізу;

3) часто використання цього механізму може означати перехід від програми, яка не працює, до програми, яка буде виконуватися.

Синтаксично використання в правила відсікання виглядає як входження цільового твердження з предикатом `'!'`, який не має ніяких аргументів. Як цільове твердження цей предикат завжди погоджується з базою даних і не може бути перепогодженим. Разом з тим, він має побічний ефект, що міняє повернення у разі перепогодження цілей. Ефект полягає в тому, що маркери деяких цілей стають недоступними, оскільки для них неможливо знайти нове порівняння. Щоб зрозуміти суть дії цього механізму, розглянемо алгоритм побудови програми користування бібліотечним каталогом.

Нехай у базі даних Прологу міститься інформація про наявність книг, про те, хто і які книги взяв у бібліотеці і коли зобов'язаний їх повернути. Один із запитів, які можна сформулювати, – це види послуг, які можна надати кожному читачеві. Деякі послуги, які названі основними, доступні кожному читачеві. Вони передбачають користування каталогом і довідкою. З іншого боку, додаткові послуги, такі, як користування абонентом чи отримання книг з іншої бібліотеки, можна надавати читачам вибірково. Одне з таких правил могло б полягати в тому, що якщо читач не повернув книгу у вказаний термін, то додаткові послуги йому не надаються, поки книга не буде

повернута в бібліотеку. Під час написання фрагмента цієї програми доцільне використання механізму відсікання. Текст правила, яке реалізує цей процес, може виглядати так:

```
послуги (Читач, Вид_послуг):-
книга_не_повернута (Читач, Книга), !,
основні_послуги (Вид_послуг).
послуги (Читач, Вид_послуг):-
загальні_послуги (Вид_послуг).
основні_послуги (користування_каталогом).
основні_послуги (отримання_довідок).
додаткові_послуги (абонемент).
додаткові_послуги (міжбібліотечний_абонемент).
загальні_послуги (Х):- основні_послуги (Х).
загальні_послуги (Х):- додаткові_послуги (Х).
книга_не_повернута ('Джек_Лінден', книга1905).
книга_не_повернута ('Тарас_Шевчук', книга89).
читач ('Джек_Лінден').
читач ('Тарас_Шевчук').
```

Який ефект від використання механізму відсікання у цьому фрагменті програми? Припустимо, що є потреба переглянути список всіх читачів і визначити послуги, які доступні кожному з них. Для цього потрібно звернутися до Прологу із запитом

```
?- читач (Х), послуги (Х,Y).
```

Пролог вибере першого читача Джека Ліндена і виявить, що в нього є неповернута книга. Для визначення послуг, які йому доступно, Пролог - використає перше твердження для предиката послуги. Це приводить до появи нового цільового твердження – **книга_не_повернута**. Після пошуку серед фактів **книга_не_повернута** виявляється, що першою серед неповернутих Джеком Ліндоном книг є книга під номером 1905. Наступне цільове твердження – відсікання. Це твердження автоматично погоджується з базою даних, в результаті в системі закріплюються всі рішення, прийняті з моменту вибору першого твердження послуги. Результат дії відсікання в правилі для предиката послуги полягає в тому, що всі цілі, вибрані з моменту, коли було взяте це правило, запам'ятовуються в системі як незмінні під час зворотного перегляду. Тому у разі спроби повторного погодження цілей альтернативне рішення для цільового твердження **"книга_не_повернута"** розглядатися не буде і цілком розумно, оскільки нас цікавить хоча б одна книга, яка не

повернута читачем у вказаний термін. Друге твердження предиката послуги (оскільки зустрілось відсікання) теж розглядатися не буде під час перепогодження цілей. Ця поведінка системи в такому разі також є розумною, оскільки ми не хочемо породжувати рішення, які б вказували, що Джеку Ліндену доступні всі види послуг. Отже, підсумовуючи дію механізму відсікання у такому разі, можна сказати, що читач, який не повернув книгу, може користуватись лише основними послугами. Немає необхідності у виявленні всіх книг, не повернутих читачем, як і в тому, щоб розглядати інші правила щодо послуг цьому читачеві. У такому разі використання відсікання привело до скорочення рішень, які прийняті після вибору цільового твердження послуги. Це твердження називають батьківським. Формально ефект механізму відсікання можна сформулювати так.

Якщо відсікання трапляється як цільове твердження, то після цього система вже не має можливості змінити рішення, прийняті з моменту появи батьківського цільового твердження. Всі альтернативи до прийнятих рішень відкидаються. Отже, спроба знову довести погодженість з базою даних будь-якого цільового твердження між батьківським і твердженням ! (відсікання) зазнає невдачі. Можна сказати, що всі ці рішення відсікаються або заморожуються, і система не має можливості їх змінити, тобто всі альтернативи відкидаються. Символ відсікання можна розглядати як деякий роздільник цільових тверджень. Так, під час опрацювання комбінації цілей

for:- a,b,c,!,d,e.f.

Пролог без обмежень може виконувати повернення серед цілей **a, b, c** доти, доки не перейде через ! (відсікання) і розпочне погодження **d**. Подальші повернення можуть здійснюватися між цілями **d, e, f,** причому може бути досягнуте неодноразове погодження кон'юнкції цілей. Але якщо буде невдача під час доведення·цільового твердження **d**, що викличе повернення через відсікання справа наліво, то ніяких спроб нового доведення цього твердження не буде, а отже доведення погодження кон'юнкція цілей **for** загалом зазнає невдачу.

Використовуючи механізм відсікання, необхідно детально з’ясувати, як саме будуть засовуватися правила програми. Оскільки для одного використання правил відсікання може бути нешкідливим або навіть корисним, то для іншого вона може призвести до непередбаченого результату. У разі введення відсікання, яка забезпечує функціонування програми для одних цільових тверджень, немає ніякої гарантії, що за умови

виникнення нових цільових тверджень буде відбуватись що-небудь розумне. Для надійності роботи програми у разі появи нових правил необхідно переглядати всі можливості використання відсікання.

3.4.2. Послідовність виконання

1. За курсом лекцій повторити теоретичний матеріал, пов'язаний з використанням механізму відсікання під час написання Пролог-програм.

2. Написати та ввести в комп'ютер програму надання послуг читачам, передбачаючи додаткові послуги читачам, які своєчасно повертали всі книжки і навчаються на відмінно.

3. Власне ім'я ввести в базу даних програми відповідно до стану своєї успішності.

4. Відлагодити програму та запустити на виконання.

5. Отримати дані про всі власні неповернуті книги та можливі послуги для себе.

6. Оформити звіт про роботу.

3.4.3. Контрольні питання

1. Суть механізму відсікання.

2. Способи використання механізму відсікання.

3. Що відбудеться, якщо в першому правилі предиката **"приєднати"** використати відсічку?

4. Який предикат чи комбінація вбудованих предикатів дає змогу в ряді випадків замінити використання предиката '!' ?

5. Суть використання відсікання з вбудованим предикатом fail.

6. Записати предикат **кількість_родичів (X,Y)** з використанням механізму відсікання і без нього. Порівняти їх ефективність.

3.5. Лабораторна робота № 5. Програма побудови елементарних геометричних зображень

Мета: Набути навикі−в практичного використання графічних можливостей середовища Турбо-Пролог.

3.5.1. Теоретині відомості

Графічні засоби середовища Турбо-Пролог володіють потужними можливостями, що дають змогу створювати на екрані зображення високої точності.

Графічні предикати Турбо-Прологу підтримують кольоровий графічний адаптер (CGA) фірми IBM − розширений графічний адаптер (EGA) фірми IBM, а також сумісні з ним.

Керування графічними режимами і кольорами в Турбо-Пролозі здійснюється за допомогою параметрів, які задають у предикатах **graphics** і **makewindow**. Предикати **dot** і **line** використовуються для зображення об'єктів в графічному режимі.

Монітор комп'ютера в графічному режим· можна розглядати як поверхню, що складена з дрібних лампочок, як· світяться різними кольорами. Ці елементи називаються пікселями (pixels). Кількість пікселів екрана визначається режимом роботи графічного адаптера. Як CGA так і EGA мають режими роботи з середньою та високою роздільною здатністю. У разі середнього розділення екран ділиться на 320 пікселів по горизонталі і 200 пікселів по вертикалі; при високому розділенні – на 640 пікселів по горизонталі та 200 по вертикалі. Для монітора EGA існує режим з покращеним розділенням, що забезпечує розділення екрану на 640 пікселів по горизонталі і 350 пікселів по вертикалі.

Створення графічних зображень у Турбо-Пролозі розпочинається з переведення екрана у графічний режим роботи. Для цього використовують предикат **graphics** такої форми:

```
graphics (Mode, Palette, Color).
```

Параметр **Mode** − ціле число від 1 до 5 для вибору графічного режиму.

Монітор CGA підтримує режим роботи 1, 2, EGA - 3, 4, 5.

Параметр **Palette** є цілим числом з можливим значенням 0 і 1 та використовується для задання однієї з двох кольорових палітр.

Параметр **Color** є цілим числом для задання кольорів на графічному дисплеї. Режим роботи 3, 4, 5 дасть змогу вибору 16 коьорів; режим 1 – 4 кольори; режим 2 – 1 колір.

Для створення вікна в графічному режимі використовується предикат **makewindow** такої синтаксичної форми:

```
makewindow (Window_number, Screen_attribute,
 Frame_attribute, Frame_string,
Starting_row, Starting_column,
Window_height, Window_width).
```

Параметр **Window_number** – ціле число, ідентифікує вікно в програм·. Параметр **String_attribute** обчислюється підсумуванням значень, що відповідають необхідним кольорам фону і тексту. Але колір фону у графічному режимі такий самий, як і колір всього тексту. Отже, можна управляти лише кольором тексту разом із параметром **Palette** предикату **graphics**.

Параметр **Frame_attribute** визначає колір рамки разом з параметром **Palette** предиката **graphics**.

Параметр **Starting_row** – ціле число, яке визначає верхній рядок створюваного вікна. Його максимальне значення 24.

Параметр **Starting_column** вказує крайній лівий стовпчик вікна. Значення цього аргументу може змінюватись від 0 до 79.

Параметр **Window_height** – ціле число, що вказує на кількість рядків, які займає вікно. Максимально можливе число їх 25.

Параметр **Window_width** – ціле число, що вказує на кількість стовпців, як· замалює вікно. Максимальне значення аргументу 80.

Для переходу від одного вікна до іншого використовують предикат **gotowindow** (**Window_number**), де **Window_number** – номер вікна, який йому присвоюється під час його створення. Для очищення вікна від текстової та графічної інформації використовується предикат **clearwindow**, який не має ніяких параметрів. Для видалення вікна з екрана використовують предикат **removewindow**. Якщо за цим вікном є інше вікно , то воно з'являється на екран·. Якщо видалене вікно є останнім, то на екрані з'являється зображення, яке було на моніторі до початку створення вікон.

Для створення зображень на екрані використовують предикати **line** і **dot.** Предикат **line** має форму

```
line (Row_1, Column_1, Row_2, Column_2, Color),
```

а предикат **dot** описується так

```
dot (Row, Column, Palette_color).
```

Усі параметри цих предикатів є цілими числами.

Параметри, що вказують рядок **(row)** і стовпець **(column)** задають розташування кінцевих точок лінії для предиката **line** і положення крапки для предиката **dot**. Параметр **Palette_color**, як правило, задає колір лінії або крапки.

У графічному режимі координати точки на екрані змінюються від 0 до 31999 по горизонталі·та вертикалі.

Використання графічних предикатів **line** і **dot** дає змогу створювати графічні об'єкти простої і складної форми. До цих простих належать прямокутник, трикутник, коло та еліпс. На їх основі можна будувати зображення дерев, будинків, пейзажів.

Як приклад побудови зображення прямокутника у верхній лівій частині екрана наведено такий предикат, в якому використовують кілька предикатів **line**:

```
    draw_a_rectange:- line (2000, 2000, 2000, 1200,
1),
    line (2000, 1200, 800, 1200, 1),
    line (800, 1200, 800, 2000, 1),
    line (800, 2000, 2000, 2000, 1).
```

Для зображення вершин трикутника можуть бути використані такі три предикати:

```
    dot (2000, 2000, 1).
    dot (2000, 3000, 1).
    dot (1000, 3000, 1).
```

Наступний приклад – це зображення кола, яке теж належить до простих фігур. Всі точки кола рівновіддалені від центра. Тому координати **Column** і **Row** для будь-якої точки можуть бути розраховані за допомогою радіуса і координати центра **Center_column i Center_row**. Виконуються такі співвідношення:

```
    Row = Center_row - R * cos A,
    Column = Center_column + R * sin A.
```

Для кола значення параметра A зманюється від 0 до 6.283 рад. Якщо збільшувати параметр A на 0.02 рад, то отримаємо близько 314 точок кола.

Після одержання значень **Row** і **Column** за допомогою предиката **dot** можна вибрати точку бажаного кольору.

Під час програмування комп'ютерної графіки необхідно враховувати деякі технічні аспекти, що ускладнюють завдання програміста. Розробники графічних моніторів і адаптерів використовують дещо іншу шкалу горизонтальних і вертикальних розмірів. Ці розміри масштабуються за допомогою параметра, який називається видовим відношенням (aspect ratio), і визначається відношенням висоти екрана до його ширини. Для більшості екранів це відношення становить 5/7. Якщо зобразити коло за допомогою наведених рівнянь, то отримаємо еліпс. З урахуванням видового відношення перше рівняння можна подати у вигляді

```
Row = Center_row - 1.4 * R * cos A.
```

Треба відзначити, що коефіцієнт в цьому рівнянні може змінюватися залежно від типу монітора.

3.5.2. Послідовність виконання роботи

1. За курсом лекцій повторити матеріал, пов'язаний з використанням графічних можливостей Турбо-Прологу.

2. Написати та ввести в комп'ютер програму, яка забезпечує побудову трикутника у верхній лівій частині екрана, квадрата у верхній правій частині екрана, ромба в нижній лівий частині екрана, кола у правій нижній частині екрана та еліпса в центрі екрана, забезпечивши не перекриття цих фігур.

3. Відлагодити програму та запустити її на виконання.

4. Проекспериментувати з вибором кольору фігур, працюючи в режимах з різною роздільною здатністю.

5. Оформити звіт про роботу.

3.5.3. Контрольні питання

1. Які можливі режими роботи комп'ютера при створенні графічних зображень?

2. Призначення предикатів makewindow та graphics і характеристика їх параметрів.

3. Які основні предикати використовуються для зображення простих фігур?

4. Що таке коефіцієнт видового відношення?

5. Записати модуль побудови еліпса в графічному режимі.

6. Як зміниться зображення, якщо в програмі побудови еліпса врахувати коефіцієнт видового відношення?

3.6. Лабораторна робота № 6. Описання підходів до побудови експертних систем

Мета: Освоїти основні підходи до побудови експертних систем та набути практичних навиків їх реалізації в середовищі· Турбо Пролог.

3.6.1. Теоретичні відомості

Експертна система – це комп'ютерна програма, яка в деякій галузі знань проявляє рівень пізнань, який рівнозначний рівню пізнання людини-експерта Як правило, ця галузь знань є доволі вузькою, але кількість використань надзвичайно велика. Це і розуміння мови, і аналіз зображень, прогноз погоди і медична діагностика, розробка інтегральних схем і моделювання складних процесів та багато інших.

Для здійснення експертизи комп'ютерна програма повинна бути здатна розв'язувати задачі за допомогою логічного висновку і отримувати надійні результати. Крім цього, система повинна мати доступ до бази фактів, які називаються базою знань. Під час консультації експертна система повинна вміти робити висновки з інформації, яка міститься в базі знань, а крім того використовувати і ту інформацію, яка виникає в ході консультації. Отже, експертну систему можна подати у вигляді трьох складових частин:

1. База знань (БЗ).
2. Механізм висновків(МВ).
3. Система інтерфейсу користувача(СІК).

Центральна частина експертної системи – база знань. Вона містить правила, які описують явища і взаємозв'язки між ними, методи і знання для розв'язання задач з галузі використання системи. База знань може бути складена з фактичних знань і знань, що використовують для виведення інших знань. База знань розміщується на диску або іншому носії інформації.

Механізм висновків містить принципи і правила роботи. Він "знає", як розумно отримати потрібні висновки на основі тієї інформації, що міститься в базі знань. Цей механізм визначає які правила бази знань потрібно викликати і організовує до них доступ, виконує ці правила, визначає найсприйнятливіший результат і передає його програмі інтерфейсу з користувачем. Інтерфейс – це частина експертної системи, яка взаємодіє з користувачем.

Система інтерфейсу з користувачем приймає від нього інформацію і передає йому цю інформацію. Як правило, користувачі мають поверхове уявлення про організацію бази знань, тому інтерфейс полегшує роботу користувача з експертною системою.

Механізм висновків і система інтерфейсу з користувачем розглядаються як додатки до бази знань і утворюють оболонку експертної системи. Для однієї бази знань може бути реалізовано кілька оболонок. Добре розроблені оболонки в своєму складі містять механізм для додавання і оновлення інформації в базі знань.

Отже, найвідповідальніший момент у розробці експертної системи – це створення бази знань. Будуючи бази знань, необхідно їх подавати у найпростішій і найзручнішій формі для забезпечення до них доступу за допомогою простих і природних механізмів. Найширше застосовуються для подання баз знань два способи. Перший – це розміщення фактів і чисел в правилах Турбо-Прологу. Це представлення використовується в експертних системах, що ґрунтуються на правилах. Другий спосіб – це організація фактів і чисел у твердженнях, які утворюють базу знань на твердженнях. Таке представлення знань використовується в експертних системах, що ґрунтуються на логіці.

Існують інші системи представлення знань. До них належать системи на фреймах, що ґрунтуються на логічних групах атрибутів об'єкта досліджень, і системи на модулях, що ґрунтуються на знанні структури і поведінки пристрою, що є предметом дослідження.

Сьогодні найпоширеніші експертні системи, які ґрунтуються на правилах.

Прикладом побудови найпростішої бази знань може бути таблиця, складена з двох стовпчиків. Один містить назву країни, другий – відповідну їй столицю

Країна	**Столиця**
США	**Вашингтон**
Англія	**Лондон**
Франція	**Париж**

Під час побудови реальної бази знань представлення знань відповідає мові програмування, яка вибрана для розробки експертної системи. Твердження Турбо Прологу, що містять наведені знання, можна записати як:

```
capital ("Washngton","USA").
```

```
capital ("London","England").
capital ("Parish","France").
```

Це представлення утворює базис експертної системи, що ґрунтується на логіці.

Ці самі знання можна записати і у вигляді правил "якщо, то". Правила для попередніх трьох тверджень матимуть вигляд:

```
capital_is ("Washington"):-country (is,"USA"),!.
capital_is ("London"):-country (is,"England"),!.
capital_is ("Parish"):-country (is,"France"),!.
```

Наведені правила прості, але вони демонструють принципи побудови експертних систем Турбо Прологу.

Механізм виведення в експертній системі реалізується через пошук і порівняння розрахунку. У Турбо-Пролозі використовують внутрішні програми уніфікації, для яких необхідно лише записати відповідну специфікацію. Якщо механізм виведення виявляє кілька можливих відповідей, то здійснюється певний вибір. Пріоритет надається або більш конкретним правилам, або правилам, що враховують більше поточних даних. Цей процес називають розв'язанням конфлікту.

Для демонстрації механізму виведення допустимо, що необхідно визначити, чи Мадрид є столицею Іспанії. Для формування запиту механізм висліду, що ґрунтується на логіці, утворює ціль:

capital ("Madrid","Spain").

Якщо порівняний факт не знайдено в базі, то система видає відповідь (F), тобто "невірно".

Якщо експертна система ґрунтується на правилах, то запит формується у вигляді:

capital_is ("Madrid"):-country (is,"Spain"),!.

Система інтерфейсу користувача забезпечує взаємодію між експертною системою і користувачем; вона включає кілька функцій:

1. Обробка даних, що вводяться з клавіатури і виведення їх на екран.
2. Підтримка діалогу між користувачем і системою.
3. Розпізнавання ситуації нерозуміння між користувачем і системою.
4. Забезпечення "дружності·" стосовно користувача.

У всіх експертних системах існує зв'язок між вхідним потоком даних і даними в базі знань. Під час консультацій вхідні дані зіставляються з даними в базі знань. Як результат видається позитивна або негативна відповідь.

Роботу інтерпретатора експертної системи можна описати послідовністю таких кроків:

1. Інтерпретатор зіставляє зразок правила з елементами даних в базі знань.

2. Якщо можна викликати більше від одного правила, то інтерпретатор використовує механізм розв’язання конфлікту.

3. Інтерпретатор застосовує вибране правило, щоб знайти відповідь на поставлений запит.

Цей три кроковий процес має назву циклу розпізнавання дій. Кількість таких продукційних правил визначає розмір бази знань.

Дуже важливим є вибір структури самих правил. Будуючи їх, необхідно дотримуватись таких рекомендацій:

1. Використовувати максимально достатню множину умов для визначення продукційного правила.

2. Уникати суперечливих продукційних правил.

3. Конструювати правила, дотримуючись структури, яка характерна для певної предметної галузі.

Розробка експертної системи вимагає великої організованості та уваги до дрібниць. Проектуючи громіздкі експертні системи, перевагу віддають системам, що ґрунтуються на логіці, оскільки тут база знань міститься в файлі на диску, і, власне, обмежень на пам'ять не накладається. Водночас, коли відомо, що експертна система міститиме не кілька сотень продукційних правил, необхідно віддати перевагу системі, що ґрунтується на правилах. Продукційні правила майже не залежать одне від одного, тому створення і відлагодження такої системи простіше. Якщо необхідно, нескладно ввести зміни в продукційні правила.

В системах, що ґрунтуються на логіці, зміна параметрів всередині бази знань потребує обережності, оскільки такі зміни менш помітні, що утруднює їх експлуатацію.

3.6.2. Послідовність виконання роботи

1. Повторити за курсом лекцій теоретичний матеріал, пов'язаний з розробкою експертних систем в середовищі Турбо Пролог.

2. Створити базу знань, в яку увійшли б всі країни Європи з їх столицями. Базу створити двома способами: основану на правилах та логічну базу знань.

3. Відлагодити написану програму та протестувати, формуючи відповідні запити.

4. Оформити звіт про роботу.

3.6.3. Контрольні питання

1. Структура експертної системи.
2. Означення експертної системи та типи експертних систем.
3. Що таке система інтерфейсу користувача?
4. Основні функції інтерпретатора експертної системи.

ЧАСТИНА 2. ОСНОВИ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ

Розділ 4. Основні методи та підходи у функціональному програмуванні

4.1. Суть функціонального програмування

4.1.1. Історія розвитку дисципліни та її основні завдання

Метою дисципліни є вивчення і практичне засвоєння засобів функціонального програмування для розв'язування як наукових, так і прикладних задач, зокрема, використання програмних засобів, які створені у функціональному програмуванні для розв’язання задач штучного інтелекту.

Під функціональним програмуванням слід розуміти спосіб складання програм, в якому єдиною дією є виклик функції, єдиним способом розчленування програми на частини – виклик імені цієї функції і задання для цього імені виразу для вирахування функції. У функціональному програмуванні існує одне правило створення нових функцій, так зване правило композиції функції, яка полягає в використанні оператора суперпозиції функцій. У такому програмуванні не існує ніяких операторів присвоєння, ніяких операторів циклу, операторів передачі управління. Тим самим мова Лісп як мова функціонального програмування істотно відрізняється від таких традиційних мов програмування, як Алгол, Фортран, Асемблер, Паскаль, СІ. Ці особливості функціонального програмування зумовлюють певні труднощі при його вивченні і це природно.

Якщо згадати історію розвитку функціонального програмування, то автор λ-обчислення, яке є теоретичною моделлю сучасного функціонального програмування, – американський вчений Альфонсо Чьорч немало витратив часу на те, щоб запрограмувати у вигляді функції процес віднімання 1 з натурального ряду. Він так і не впорався з цією задачею і лише в 1932 р. молодий в той час аспірант Стефан Кліні запропонував дещо штучну на перший погляд, а насправді повністю відповідну суті справі конструкцію:

F(x,y,z) = якщо x=0 то 0 інакше якщо y+1=x то z
інакше F(x,y+1,z+1);

n-1 = F(n,0,0).

Далі на зміну λ-численню прийшли теорія частково-рекурсивних функцій та машин Тюрінга. Але, тим не менше, тепер знову на озброєння функціонального програмування береться ця теорія λ-числення, яка все ширше застосовується при розв'язанні складних прикладних задач.

Якщо говорити про становлення функціонального програмування як науки в нашій країні, то існувала методологічна школа побудови пакетів прикладних програм, започаткована у 1954 р. Саме в ній були реалізовані основні поняття функціонального програмування. Тут мається на увазі багатблочне програмування, обробка складних об'єктів у функціональних позначеннях, що було запропоновано Канторовичем і реалізовано в серії конкретних програмних систем СТРІЛА, БЕСМ, М-20 в 50-і – 60-і роки.

Наступний крок у розвитку функціонального програмування пов'язаний з іменем американського вченого Дж. Маккарті, який в 1961 р. розробив мову Лісп. Хоча й Маккарті усвідомлював фундаментальний характер мов функціонального програмування, тим не менше вони ще довгий час використовувались вузьким колом спеціалістів.

Сьогодні мови функціонального програмування досить широко застосовуються. Використання цих мов при дослідженнях зумовлене двома причинами:

1) ці мови ідеально пристосовані для запису інтерпретатора тієї мови, що вивчається;

2) для кожної програми на цій мові завжди можна записати еквівалентну функцію чи функціональну програму.

Надзвичайна простота та потужність функціонального програмування робить його зручним для таких семантичних специфікацій.

У задачах штучного інтелекту доводиться оперувати складними символьними структурами даних. Маккарті, створивши мову Лісп, подолав деякі з цих труднощів.

Якщо в перших ЕОМ розміри пам'яті і швидкодії були невеликі, то сьогодні, завдяки сучасним ЕОМ, є можливість більшу увагу приділяти програмному забезпеченню, а не вартості самої машини. Тому все більший акцент в комп'ютерних науках буде робитися на якість, зрозумілість і легкість програм, а не на оптимальне використання обладнання.

У мовах функціонального програмування характерний компроміс між орієнтацією на машину і користувача з поступовим наближенням до потреб

користувача. Але глобального прориву в застосуванні функціональних мов слід очікувати з появою ЕОМ нової архітектури, відмінної від нейманівської.

4.1.2. Основні поняття функціонального програмування

Базовим поняттям в строго функціональній мові є поняття функції, саме воно має первинне значення. За допомогою цього поняття програми, які виглядають довгими і незрозумілими традиційними мовами, стають короткими, зрозумілими і доступними на мовами функціонального програмування.

Під функцією будемо розуміти правило, згідно з яким кожному елементу з деякого класу ставиться у відповідність єдиний елемент з іншого класу. Іншими словами, для заданих двох класів елементів, які називаються відповідно областю визначення і множиною значень цієї функції, кожному елементу із області визначення ставиться у відповідність один елемент із області значень. Важливою властивістю такої відповідності є те, що елемент із області значень єдиним способом визначиться за елементами із області визначення.

Існує багато способів описання простих функцій. Цю відповідність можна записати у вигляді послідовності пар

(a, p) (b, g) (c, g) (d, n),

або у вигляді таблиці

X	a	b	c	d
f(x)	p	q	q	r

або за допомогою рівностей

f(a)=p, f(b)=g, f(c)=g, f(d)=n.

Але кожен з цих способів має той недолік, що тут ми не знайдемо однозначно відповідності між областю визначень і значень функції. Коли область визначень велика, то це не є цілком очевидним. Більше того, коли область визначення нескінченна, то такі позначення функції не можуть використовуватися.

Розглянемо реальний приклад. Функція **квадрат** має областю визначення клас цілих чисел (додатні, від'ємні і число нуль), а область значень цієї функції – всі невід'ємні числа. Для прикладу можна записати:

квадрат (3) = 9; квадрат (5) = 25.

Але у такий спосіб ми не можемо повністю описати функцію, оскільки область визначення функції є нескінченною.

Але її цілком можна описати, задавши правило або визначення цієї функції

квадрат (у) = у · у .

У цьому правилі точно вказано як для будь-яких **х** з області визначення функції **квадрат** вирахувати його образ. Тут **х** використано для позначення будь-якого елемента з області визначення. Таку змінну називають параметром визначення. Визначальне правило, яке розглянуто для функції **квадрат,** є тим стандартним способом, за допомогою якого ми будемо представляти функції. При визначенні функцій за допомогою правил області визначень і області значень можуть задаватися за допомогою додаткових умов. Адже точно так само ми б могли дати визначення функції **квадрат** і для дійсних чисел, задавши інші обмеження (умови) на область визначення функції. Часто будемо називати областю значень і областю визначення ширші множини, в яких деякі елементи із області визначення не мають відповідника в області значень. Так функція **обернена (х)=1/х** відображає область дійсних чисел в область дійсних чисел, хоча й число 0 не входить в область визначення, оскільки елемент **обернена (0)** не визначений згаданим правилом.

Інший приклад. Розглянемо функцію **max**, яка видає як результат більше з двох чисел, до яких застосована:

max(1, 3)=3; max(1.3, -1,7)=1.3.

Правило для визначення цієї функції найзручніше записати за допомогою двох параметрів

$$\mathbf{max}(x, y) = \begin{cases} x, & \text{якщо } x > y \\ y & \text{в іншому випадку} \end{cases}.$$

Якщо скористатися відомою з традиційних мов програмування формою

якщо <умова> то <вираз> інакше <вираз> ,

маємо простіше означення:

max(x, y)= якщо x > y то x інакше y.

Розглянемо тепер, як можна здійснювати програмування за допомогою функцій. Насамперед ми створюємо базовий набір потрібних нам функцій і, беручи їх з відповідними аргументами, знаходимо потрібний результат згідно заданого правила при означенні функцій. Функція, по суті, є програмою, яка як вхідний сигнал сприймає її аргумент, а як вихідний – результат вирахування функції. Щоб мати можливість конструювати потужніші програми, треба вміти визначати нові функції через старі. Але, очевидно, ці нові повинні посилатися лише на базові функції.

Розглянемо визначення

найб (x, y, z)= max (max (x, y), z).

За іменем функції неважко здогадатися, що вона здійснює обчислення найбільшого з трьох чисел.

Визначення цієї функції ілюструє один із фундаментальних способів побудови нових функцій із старих. Це метод композиції функцій. Вкладенням одного використання функції в інше ми отримуємо композицію функції з самою собою. Загалом функції можуть бути вкладені на довільну глибину і утворювати як завгодно глибокі композиції. Так, найбільше з 6 чисел може бути визначено однією з наступних композицій:

max (найб (a, b, c), найб (d, e, f));
найб (max (a, b), max (c, d), max (e, f));
max (max (max (a, b), max (c, d)), max (e, f)).

Щоб програмувати за допомогою функцій, треба мати множину основних функцій і використовувати композицію для означення нових функцій в термінах старих. Отже, можна побудувати цілу бібліотеку функцій. Деякі з них можуть бути настільки потужними, що називаються

функціональними програмами. Виникає запитання, чи потрібно ще що-небудь, крім набору основних функцій і можливості скласти з них композиції, щоб визначити бібліотеку функціональних програм. Виявляється, що більше нічого не потрібно.

Вибір сукупності функцій, безсумнівно, дуже важливий. Для побудови ефективних функціональних програм на сучасних ЕОМ цей набір повинен бути досить потужним, і очевидно, як базові можуть бути вибрані різні функції залежно від потреб користувача.

Очевидно, до основних функцій необхідно включити і арифметичні операції. В математиці є звичними алгебраїчні вирази, що включають арифметичні операції:

a+b*c;
(2*x+3*y)/(x-y).

Очевидно, операції +, -, *, / є функціями, кожна з яких має область визначення і область значень. Можна явно визначити їх як функції:

плюс (x, y) = x+y мнж (x, y) = x*y
мінус (x, y) = x-y діл (x, y) = x/y

Використовуючи ці імена функцій замість операцій, можна переписати алгебраїчні вирази, наведені вище, у вигляді

плюс (a, мнж(b, c));
діл (плюс (мнж (2, x), мнж (3, y)), мінус (x, y)).

Цей запис демонструє структуру, яку Лендін назвав аплікативною. У такій структурі кожен вираз може бути розділений на складові, кожна з яких є або операцією, або операндом. Операнд означає значення, операція – функцію. Структура аплікативного виразу проста, він складається із операцій, які застосовуються до операндів. Важлива риса аплікативної структури, – що її значення знаходять за значеннями складових частин. Мова, в якій зберігається ця риса аплікативної структури, називається аплікативною або строго функційною мовою.

4.1.3. Спільні та відмінні риси функціональних та процедурних мов

Розглянемо структуру традиційних мов і обговоримо ті риси, які не є строго функціональними.

В усіх сучасних мовах програмування є конструкції, які називаються процедурою або підпрограмою, поняття яких базується на ідеї функції.

Запишемо означення функції **max** на Фортрані і на Паскалі

Фортран	Паскаль
REAL FUNCTION MAX(X,Y) REAL X,Y IF (X.GE.Y) GOTO 10 MAX=Y RETURN 10 MAX=X RETURN END	FUNCTION MAX(X,Y:REAL):REAL; BEGIN IF X>=Y THEN MAX:=X ELSE MAX:=Y END.

Ці визначення реалізують строгі визначення функції. Однак цими мовами може бути визначена обмежена кількість функцій і в практичних задачах використовуються загальніші процедури, які не є строго функціональними, дають побічний ефект, мало наочні, важкі для розуміння.

Ми почнемо відходити від строгих функцій, коли визначимо процедури, які не видають результати, а присвоюють його значення одному із параметрів. Можна дати означення **max** так, що виклик **max(a,b,n)** визначає найбільше з двох чисел **A** і **B** і присвоює це значення змінній **n**.

Запишемо означення функції на Фортрані і на Паскалі.

Фортран	Паскаль
SUBROUTINE MAX(X,Y,Z) REAL X,Y,Z IF (X.GE.Y) GOTO 10 Z=Y RETURN 10 Z=X RETURN END	PROCEDURE MAX(X,Y:REAL;VAR Z:REAL); BEGIN IF X>=Y THEN Z:=X ELSE Z:=Y END.

Щоб тепер вирахувати найбільше з трьох значень **A**, **B**, **C,** необхідні два послідовні виклики процедури **max (a,b,m)**, **max (m,c,m)**. Означення функції не порушується, але ми втратили деякі зручності при обчисленні найбільшого з трьох чисел. Ми відходимо трохи далі від строгого означення функції, якщо визначимо **max** як процедуру, яка визначає найбільше з двох чисел **A** і **B** і присвоює це значення параметру **A**:

Запишемо означення функції на Фортрані і на Паскалі

Фортран	Паскаль
SUBROUTINE MAX(X,Y) REAL X,Y IF (X.GE. Y) GO TO 10 X=Y 10 RETURN END	PROCEDURE MAX(X,Y:REAL); BEGIN IF X<Y THEN X:=Y END.

Можна констатувати, що тут означення функції не порушено, хоча зроблено неявним. Програми, складені з строгих функцій, просто визначають як нові значення, визначені із вхідних даних, в той час, як в останніх двох процедурах є побічний ефект присвоювання відносно одного із своїх параметрів.

Ще сильніший побічний ефект спостерігається, коли означити процедуру, яка присвоює результат змінній, що не є параметром процедури.

Запишемо означення функції на Фортрані і на Паскалі

Фортран	Паскаль
SUBROUTINE MAX(X,Y) REAL X,Y COMMON M REAL M IF (X.GE.Y) GOTO 10 M=Y RETURN 10 M=X RETURN END	PROCEDURE MAX(X,Y:REAL); BEGIN IF X>=Y THEN M:=X ELSE M:=Y END.

Тепер найбільше з трьох значень визначається двома викликами

max (A, B) і max (M, C)

Процедури з такими визначеннями є важкими для розуміння, тим не менше вони включаються в традиційні мови програмування, оскільки без них програми не можуть ефективно використовувати машину.

Строго функційна мова не допускає побічних ефектів. Програміст будує тільки ті функції, які за значенням аргументів вираховують значення функції. Тому в функціональному програмуванні відсутні такі поняття, які є звичні в традиційних мовах, зокрема відсутній оператор присвоєння. Те саме відноситься і до поняття змінної. У строго функціональній мові змінна подібна швидше до математичної змінної і використовується для того, щоб дати імена аргументам функції, її постійне значення зберігається протягом всього процесу обчислення функції.

Строго функційна мова має дві важливі переваги над іншими. Перша – це можливість структурувати дані. Друга – можливість визначати функції вищих порядків. Цілі структури даних можуть виступати як єдине ціле. Вони можуть вводитися в функцію як аргументи і видаватися як результат. Важливо те, що один раз у такий спосіб побудована структура зберігає своє значення протягом всього процесу обчислень. Те саме можна сказати і про функції вищих порядків. Вони мають інші функції або своїми аргументами або результатами. Це дає змогу робити програми на функціональній мові коротшими, наочнішими. Очевидно їх легше написати, а тому і доцільніше експлуатувати.

4.1.4. Контрольні питання та вправи

1. Суть функціонального програмування та напрямки його застосування.
2. Основні відмінності мов функціонального програмування від традиційних мов програмування високого рівня.
3. Суть принципу композиції в функціональному програмуванні.
4. Характеристика способів представлення функцій та особливості цього представлення в строго функціональній мові.
5. Характеристика аплікативної структури та її переваги над іншими способами представлення даних.
6. В чому суть побічного ефекту при визначенні функцій?

7. Описати функцію **min (x, y)**, яка вибирає найменше з двох аргументів функційною мовою, мовою Паскаль та Сі і порівняти ефективність цих означень.

8. На основі функції **min (x, y)** побудувати функцію **min (x, y, u, v)**, яка вибирає найменше з чотирьох значень.

4.2. Строго функціональна мова. Примітивні функції мови Ліспі

4.2.1. Способи подання даних

Коли мова йде про побудову строго функціональної мови, про створення строго функціональних програм, то це означає що нові функціональні програми створюють, використовуючи деякі базові або як їх ще називають примітивні функції. Для побудови строго функціональної мови введемо ряд абстрактних понять для представлення даних в функціональних програмах.

Для представлення даних, які характерні для нечислового програмування, зокрема, при розв'язанні задач штучного інтелекту, побудові експертних систем, доведенні теорем, розпізнаванні образів, теорії ігор необхідні багатші області визначення функцій, ніж звичайне представлення числа в машині. Разом з тим, таке представлення повинно бути як універсальним, так і простим, зручним в користуванні. Такою властивістю характеризується форма даних, запропонована Джоном Маккарті при розробці функціональної мови Ліспі (від англійського List Processing – обробка списків). Ця форма даних, яку називають S-виразом, є одночасно доволі простою і достатньо універсальною.

Розглянемо найпростіші об'єкти Ліспу, з яких будуть будуватися інші структури. Очевидно, мова складається з алфавіту, який може бути вибраний достатньою мірою довільно. Побудуємо алфавіт із таких літер:

а) букви: великі букви англійського та українського алфавіту;

б) цифри від 0 до 9;

в) обмежувачі Ліспу: ( ) . _ nl ;

г) інші знаки : + - x / , : : = .

Як бачимо, кількість літер в алфавіті є невеликою. Водночас Ліспі призначений для обробки великих обсягів даних, коли доводиться мати справу з колосальною кількістю об'єктів різної природи. Для їх позначення

використовується поняття атома. Атомом називається довільна послідовність букв, цифр, інших літер, розміщених між двома обмежувачами мови Лісп, не включаючи їх самих.

Слід відзначити, що в різних реалізаціях функціональної мови на атоми можуть бути накладені певні обмеження. В послідовності літер

(A (ATOM X24) - ABC 012 A+B(X) (GO TO))

містяться такі атоми :

A
ATOM
X24
-
ABC
012
A+B(X)
GO
TO

У програмі на Ліспі атоми можуть означати функції і константи, які мають певний зміст в Ліспі, функції, змінні і константи, які визначені в самій програмі; і, нарешті, самих себе, тобто ті послідовності літер чи окремі літери, якими вони зображаються.

Інший тип даних, який використовується в функціональному програмуванні – це списки. Списком називається скінченна послідовність елементів, яка взята в круглі дужки. Елементом списку може бути атом або список. Приклади списків :

(A B C D E);
(U1 U2 U3 U4);
(CAR (QUOTE (P Q 10));
(1 (2 3) 6 (78 80));
(((1000)));
().

Останній приклад – порожній список, який не містить жодного елемента. Для нього прийняте спеціальне позначення у вигляді атома **NIL**. Елементи списку відділяються один від одного пропусками. Кілька пропусків, які йдуть підряд, еквівалентні одному пропуску. Пропуск є істотним, коли підряд йдуть два атоми, в інших випадках він не несе ніякого функціонального навантаження і може використовуватися для наочності і читабельності функціональної програми.

Дужки в Ліспі є надзвичайно важливими синтаксичними знаками. Необхідно уважно стежити за правильністю їх розставлення. Кількість дужок, що відкривається, повинна збігатися з кількістю тих, що закриваються. Відсутність однієї дужки робить програму беззмістовною. На практиці списки можуть вкладатися один в інший на довільну глибину і утворювати доволі складну дужкову структуру. Труднощі при читанні довгих списків компенсуються легкістю їх опрацювання.

Надалі під S-виразами ми будемо розуміти атоми (числові, символьні або змішані) або S-вирази, взяті в круглі дужки (списки).

Як приклад використання S-виразів розглянемо представлення для простих алгебраїчних формул. Очевидно, необхідно розрізняти константи, змінні і бінарні операції. Будемо константи і змінні зображати за допомогою атомів, а бінарну операцію у вигляді списку, де на першому місці записується ім'я операції, а за нею слідують імена операндів. Наведемо для кожної з можливих арифметичних формул її подання у вигляді S-виразу:

Формула	S-вираз
константа	атом числовий
змінна	атом символьний
p + q	(ПЛЮС P Q)
p - q	(МІНУС P Q)
p · q	(МНОЖ P Q)
p / q	(ДІЛ P Q)

Отже, якщо би мали задану формулу **Y+5·X-6**, то вона може бути записана так у вигляді S-виразу:

(ПЛЮС Y (МІНУС (МНОЖ 5 X) 6))

або у вигляді

(МІНУС (ПЛЮС Y (МНОЖ X)) 6)

і багатьма іншими можливими способами.

Тобто існує багато різних способів представлення одних і тих самих даних. Який із можливих способів ми виберемо – цілком визначається тим, що ми хочемо робити з цими даними. Конкретний вибір даних цілком може бути придатним для одних цілей і беззмістовним для інших.

4.2.2. Примітивні функції мови Ліспа

Для того, щоб оперативно працювати з S-виразами, необхідно вміти розчленити їх на складові частини. Для цих цілей введемо деякі базові, або, як їх називають, примітивні функції і покажемо, як можна визначити нові функції в термінах цих старих.

Цим функціям дано традиційні імена мови LISP, хоча, на перший погляд, вони здаються незвичними. Перша функція, яка дає змогу вибрати перший елемент із списку – це функція **CAR**, яка є функцією одного параметра. Якщо значення аргумента є **NIL**, то значення функції **CAR** не визначене. Поряд з функцією **CAR** є зв'язана з нею функція **CDR**, яка також є функцією одного аргумента, значення якого повинно бути непорожнім списком. Виключаючи з цього списку перший елемент, отримаємо значення функції **CDR**. Для атома функція **CDR** також невизначена. Приклади роботи цих функцій

X	(CAR X)	(CDR X)
(A B)	A	(B)
(A B C)	A	(BC)
((A B) (C D))	(AB)	((CD))
(A)	(A)	NIL

За допомогою цих двох функцій можна вибирати із списку будь-який елемент, знаючи що він є в списку. Так, третій елемент списку **X** може бути вибраний за допомогою S-виразу

(CAR (CDR (CDR X))),

якщо список **X** заданий раніше.

Надалі ми будемо допускати введення нових функцій за допомогою функціональних визначень, використовуючи малі букви для позначення імен функцій та їх аргументів. Так, співвідношення

третій (x) = car (cdr (cdr (x)))

визначає нову функцію **третій**, яка розглядалася раніше.

Ці дві функції **CAR** і **CDR** називаються примітивними селекторами, оскільки вони здійснюють вибірку елементів із заданого S-виразу і формують новий S-вираз.

Перейдемо до розгляду примітивного конструктора – функції **CONS**. Ця функція бере як аргументи два S-вирази і з'єднує їх в єдиний S-вираз так, щоб вихідні компоненти отримувалися застосуванням до результату функцій **CAR** і **CDR**, як показано нижче

X	Y	(CONS X Y)
A	(B C)	(A B C)
(A B)	((C D))	((A B) (C D))
(A B)	(C D)	((A B) (C D))
A	NIL	(A)

Взагалі кажучи, другий аргумент функції **CONS** повинен бути списком або спеціальним атомом **NIL**, тобто порожнім списком. Результатом **CONS** тоді буде список, який отримується включенням першого аргумента як першого елемента цього списку **Y**. Тому, якщо список **Y** має довжину **n**, то **(CONS X Y)** буде мати довжину **n+1** незалежно від величини **X**. Як приклад використання функції **CONS** розглянемо означення функції, яка для заданих атомів **A, B, C, D** будує спискову структуру з двох елементів, кожен з яких, своєю чергою, є списком: **((A B) (C D))**. Така функція може бути означена так:

двочлен (X, Y) = cons (X, cons (Y, NIL))
квадрат (A, B, C, D) = двочлен (двочлен (A, B), двочлен (C, D))

Остання функція, використовуючи функцію **двочлен (x,z)**, будує вищенаведену спискову структуру.

4.2.3. Елементарні предикати

Для успішного опрацювання списків необхідно вміти розрізняти, чи є значення S-виразу списком або атомом, а також встановлювати рівність атомів. Цим цілям служать елементарні предикати **АТОМ** і **РІВНО**. Будемо позначати істину атомом **І**, а фальш – **Ф**. Функція **(АТОМ Х)** набуває значення істина лише якщо **Х** є символьним або числовим атомом. Наприклад:

X	(ATOM X)
A	I
(A)	Ф
(A B C)	Ф
ABC	I
NIL	I
127	I
(127)	Ф

Відзначимо, що функція **ATOM** не розрізняє числових і символьних атомів. Використовуючи цю функцію, можна довизначити функцію **CAR** так, щоб вона видавала символ **NIL**, якщо її аргументом є атом і перший елемент S-виразу в іншому випадку:

car1 (x) = якщо атом (x) то NIL інакше car (x)

Для встановлення рівності атомів використовується предикат **РІВНО (X Y)**. Але необхідна обережність при його використанні: принаймні один із аргументів його повинен бути атомом, в противному разі результат невизначений. Причому предикат є істинним лише тоді, коли **X** і **Y** є одним і тим самим атомом і фальшивим, коли атоми є різні або один із них є списком. Очевидний спосіб використання цього предиката дає наступне визначення:

розмір (x) = якщо рівно (x, NIL) то 0 інакше
якщо рівно (car (x), NIL) то 1 інакше 2

Ця функція видає значення 0, 1 або 2 відповідно тому, чи її аргумент є списком з 0, 1 або більшою кількістю атомів.

Вище розглянуті примітивні функції **CAR**, **CDR**, **CONS**, **ATOM**, **РІВНО** є тією базою, на основі якої можна будувати нові складні функції.

4.2.4. Контрольні питання та вправи

1. Дати означення атома у функціональному програмуванні.
2. Поняття порожнього простого та складного списків.
3. Що таке S-вираз? Подати алгебричну формулу **(ax+bx)/(dy+cy)** у вигляді S-виразу.

4. Дати функціональне означення примітивних селекторів мови Лісп.

5. Пояснити роботу елементарного конструктора мови Лісп та обмеження на його використання.

6. Дати визначення 4 функцій, кожна з яких видає один із елементів структури **((AB)(CD))**.

7. Характеристика елементарних предикатів мови Лісп.

8. Визначити функцію, яка видає значення **NIL**, якщо її аргумент атом, а в противному випадку – початковий список без першого елемента.

9. Описати функцію, яка визначає віддаль між двома точками в просторі. Вважати, що функція **корінь(x)**, яка видає квадратний корінь з **x**, є задана.

10. Записати у вигляді S-виразів арифметичні операції: **p + q; p - q; p * q; p / q; p в степ q;**

11. Запропонувати опис позицій на шахівниці у вигляді S-виразів.

12. Вибрати подання для опису схем, складених з резистивних елементів у вигляді S-виразів.

4.3. Нестрого функційні елементи та вбудовані функції мови Лісп

4.3.1. Арифметичні функції

Коли ми говоримо про побудову програм на строго функціональній мові, то мається на увазі написання функціональних програм, виходячи з деякого базису простих функцій. Як такі функції в мові Лісп вибрані примітивні селектори **CAR** і **CDR**, конструктор **CONS** та примітивні предикати **ATOM** і **EQ**. На основі цих функцій та використовуючи фундаментальний принцип композиції, можна будувати нові функції, найскладніші з яких називають функціональними програмами.

Але написання програм лише таким способом було б надто примітивним і громіздким. Тому в мові Лісп є ряд вбудованих функцій, на які можна посилатися безпосередньо по імені при побудові власних функціональних програм. Це дає змогу створювати ефективніші програмні конструкції. Ознайомимося з цими функціями. Слід мати на увазі, що в різних версіях функціональної мови ці функції можуть мати різні імена.

В діалекті S-Ліспу при написанні програм використовують ряд функцій, які називають арифметичними. Їх аргументи – цілі десяткові числа. Перерахуємо ці функції, записуючи їх у вигляді S-виразу:

1) **(PLUS X1 X2 ... XN)** – кількість аргументів довільна. Результат функції – сума значень аргументів.
2) **(TIMES X1 X2 ... XN)** – кількість аргументів довільна. Результат функції – добуток значень аргументів.
3) **(DIFFERENCE X Y)** – двомісна функція. Результат – різниця між аргументами.
4) **(MINUS X)** – одномісна функція, яка змінює знак перед аргументом.
5) **(QUOTIENT X Y)** – двомісна функція. Результат – ціле число від ділення **X** на **Y**.
6) **(REMAINDER X Y)** – двомісна функція. Результат – залишок від ділення **X** на **Y**.
7) **(EXPT X Y)** – двомісна функція. Результат – **x** в степені **y**, якщо **y >= 0**. Для від'ємних **y** функція невизначена.
8) **(ADD1 X)** – одномісна функція. Результат – аргумент, збільшений на одиницю.
9) **(SUB1 X)** – одномісна функція. Результат – аргумент, зменшений на одиницю.
10) **(MAX X1 X2 ... XN)** – кількість аргументів довільна. Результат – найбільше із значень аргументів.
11) **(MIN X1 X2 ... XN)** – кількість аргументів довільна. Результат – найменше із значень аргументів.

4.3.2. Арифметичні предикати та логічні умови

В цій мові можна використовувати також ряд арифметичних предикатів, результат яких – атом **T**, якщо предикат істинний, а якщо ні – атом **NIL**. До них належать такі:

1) **(LESSP X Y)** – атом **T**, якщо **X<Y** і **NIL** в іншому разі,
2) **(GRATERP X Y)** – атом **T**, якщо **X>Y** і **NIL** в іншому разі;
3) **(MINUSP X)** – атом **T**, якщо **X<0** і **NIL** в іншому разі;
4) **(ZEROP X)** – атом **T**, якщо **X=0** і **NIL** в іншому разі;
5) **(ONEP X)** – атом **T**, якщо **X=1** і **NIL** в іншому разі.

При означенні функцій використовується стандартна логічна умовна форма:

якщо <логічна умова> то <вираз1> інакше <вираз2>.

Для запису цієї форми в Ліспі використовується спеціальна функція **COND**. Але ця функція відрізняється від тих, про які ми говорили. Аргументи її не враховуються, а беруться в тому вигляді, в якому записані. Кількість цих аргументів може бути довільною. Кожен аргумент – це список з двох елементів. Перший аргумент використовується як логічний вираз, а другий як вираз, значення якого може стати значенням функції. Форма звертання до цієї функції така:

(COND (P1 E1) (P2 E2) ... (PN EN)),

де **P1, ..., PN** – логічні вирази; **E1,..., EN** – довільні правильні S-вирази.

Суть функції **COND** полягає в тому, щоб вибрати один із виразів **EI** і взяти його як результат функції. Вибирають це значення у такій послідовності.

Знаходять значення логічного виразу **P1**; якщо воно є істинним, то як результат функції береться вираз **E1**. В протилежному випадку йде звертання до другого аргументу і з ним відбувається така сама процедура. Результатом функції буде деякий вираз **EI**, для якого буде встановлена істинність предиката **PI**. Далі виконання функції припиняється. Якщо ж всі **P1, ..., PN** є хибними, то результат функції буде невизначеним, про що засвідчить відповідне повідомлення. Для того, щоб ця функція в будь-якому випадку видавала результат на практиці як останній аргумент логічного виразу використовують атом **T** (істина), а як вираз також цей атом.

Як приклад застосування цієї функції запишемо вираз, який видає значення **NIL**, якщо **Y** є атом і значення **T**, якщо **Y** є списком. Тобто побудуємо функцію, яка є протилежною до примітивного предикату **(ATOM Y)**. Ця функція матиме вигляд:

(COND ((ATOM Y) NIL) (T T))

Після того, як побудована функція, необхідно вміти викликати її в термінах S-виразів. Прийняте в математиці звертання до функцій **f(x,y)** доцільно записати у вигляді S-виразу, як **(F X Y)**. Тобто при виклику функції на першому місці ставиться ім'я функції, а далі йдуть її аргументи. При виклику функції її аргументи можуть бути фактичними значеннями, а можуть

бути формальними і підлягати вирахуванню. Необхідно вміти розрізняти ці випадки. Для цих цілей служить функція **QUOTE**.

Саме ця функція вказує, що аргумент зображає сам себе і не має потреби його вираховувати. Значенням цієї функції є сам аргумент у тому вигляді, як він записаний. Так, значення виразу **(QUOTE A)** – атом **A**, а значення виразу **(QUOTE (A B C))** – атом **(A B C)**. Значення виразу **(QUOTE A B C)** є невизначеним, оскільки тут задано три аргументи, в той час як **QUOTE** є функцією одного аргументу. Не слід вважати, що значення аргументу є значенням функції **QUOTE**. Аргумент може не мати ніякого значення, а якщо і має, то результат функції **QUOTE** – не значення, а сама зовнішня форма цього аргументу. Так, значенням **(QUOTE (CAR X))** буде список **(CAR X)**, а не значення тієї функції, до якої можна звернутися в такому вигляді. Для спрощення запису програми замість функції **QUOTE** використовується символ **'**. Тобто такі S-вирази є еквівалентними:

(CAR ' (A B C D)) = (CAR (QUOTE (A B C D))
(CDR ' (A B C D)) = (CDR (QUOTE (A B C D))

В останніх виразах значення аргументу функції не вираховується, а береться в такому вигляді, як записується.

4.3.3. Поняття контексту та процедури зв’язування змінних

Часто виникає така ситуація, що аргумент вираховується. Для цього з ним необхідно зв'язати певне значення. Зв'язування змінних з конкретним значенням відбувається в контексті. Розглянемо конкретну реалізацію цього процесу в комп'ютері. Для визначення контексту в Ліспі використовується функція **SET**. Ця функція не тільки здійснює зв'язування змінної (роль якої виконує перший аргумент функції **SET**) з її значенням (другий аргумент), але і обчислює як змінну, так і її значення. Для блокування обчислення змінної застосовується модифікована функція **SETQ**. Для блокування вирахування як змінної, так і її значення є ще одна функція **SETQQ**. Для блокування обчислення одного із аргументів функції **SET** можна використовувати функцію **QUOTE** або **'**.

Для прикладу результатом виклику функції

(SET ' X ' (A B C D))

буде зв'язування змінної **X** як списку **(A B C D)**. Якщо після цього набрати функцію **(CDR X)**, то її результатом буде список **(B C D)**. Значення змінної **X**

як списку **(A B C D)** зберігається і його можна використовувати в будь-якому місці програми. Означення функцій в Ліспі ґрунтується на λ-численні Чьорча, для якого існує і інший тип аргументів у такому вигляді:

(LAMBDA (X1 X2 X3 ... Xn) Pn)

Але цей вираз інтерпретатором Ліспу не буде сприйнятий, оскільки для формальних аргументів **X1 ... Xn** тіла функції **PN** не вказано її фактичних аргументів. Коли ми хочемо в цьому місці програми отримати і використати функціональне значення, не посилаючись на ім'я функції, необхідно здійснити таке:

((LAMBDA (X1 X2 ... Xn) Pn) E1 ... En)

У цьому записі **E1 … En** є конкретні значення для формальних аргументів **X1 ... Xn**.

Будуючи нові функції, ми повинні вміти задавати ім'я функції і описати її тіло у вигляді λ-виразу. Це робиться за допомогою функції **DE**, яка з'єднує ім'я функції з λ-виразом. Оскільки всі функції описуються за допомогою λ-виразів, то при їх завданні службове слово **LAMBDA** можна опустити, записавши на першому місці **DE**, далі ім'я даної функції, список її аргументів і в кінці тіло функції. Як значення такий список виглядає так:

*** (DE LIST1 (X...Y) (CONS X (CONS Y NIL)))**

LIST1

Коли ми тепер хочемо побудувати функцію, яка формує єдиний список з підсписків **(A B C D)** і **(D K)**, то можна звернутися до функції **LIST1**, задавши вказані списки як фактичні аргументи:

(LIST1 ' (A B C D) ' (D K))

((A B C D)(D K))

Маючи ці знання про вбудовані функції, можна розпочати написання власних функціональних програм в середовищі S-Ліспу.

4.3.4. Опис середовища S-LISP

Функційна мова S-LISP записана на диску в бібліотеці LISP, в якій є 4 піддиректорії FASL, SFASL, SLFILES, RUTILS. В RUTILS англійською мовою наведено 7 уроків для освоєння мови, 6 з них описують мову аналітичних перетворень REDUCE і один – лише версії S-Ліспу. Піддиректорія SLFILES використовується для формування середовища LISP або REDUCE чи їх версій. У кореневому каталозі бібліотеки LISP в файлі

a.bat міститься інформація щодо входження в систему. Для входження в середовище Ліспу необхідно набрати системну команду REDUCE I, внаслідок чого буде створено файл UOINIT.FRZ. Після цього задається команда REDUCE.EXEC UOINIT.FRZ і система входить в середовище S-Ліспу, про що сповіщає поява символу * . Для виходу із системи необхідно набрати S-вираз (QUOTE). Система працює так, що її не потрібно підтримувати в робочому стані: чистити, стискати чи робити інші операції. Цією мовою написаний так званий збірник сміття, який починає працювати після входження в систему, постійно переглядаючи і у міру нагромадження непотрібних речей, до яких втрачений доступ, викидаючи їх. Тому після вимкнення комп'ютера необхідно заново створювати робочий файл UOINIT.FRZ, оскільки він буде знищений.

4.3.5. Контрольні питання та вправи

1. Які характерні вбудовані функції використовуються в S-Ліспі?
2. Характеристики вбудованих предикатів мови Ліспі.
3. Яка функція використовується для запису форми **ЯКЩО ... ТО … ІНАКШЕ ?**
4. Використовуючи функцію **COND,** довизначити елементарні селектори **CAR** і **CDR** та конструктор **CONS** для довільних значень аргументів.
5. Пояснити дію функцій **SET**, **SETQ**, **SETQQ** та обґрунтувати доцільність їх застосування.
6. Яка функція використовується для блокування вирахування значення будь-якого S-виразу?
7. Яке основне призначення λ–виразів?
8. Чи можлива ситуація, за якої функція **CON** буде невизначеною?
9. Описати функцію, яка обчислює середнє геометричне заданої множини цілих чисел.
10. Дати означення і записати S-вираз функції, яка видаляє заданий елемент зі списку, всі входження заданого елементу у список.
11. Означити функцію яка формує з довільного списку елементів множину елементів.

4.4. Побудова рекурсивних функцій

4.4.1. Підхід до побудови рекурсивних функцій

Для побудови функціональних програм в Ліспі використовується потужний апарат – апарат рекурсії. В перекладі слово **рекурсія** означає "повторення", "повернення назад". Рекурсивними називаються такі функції, які для свого визначення потребують повторного звертання до самих себе.

Як перший приклад розглянемо побудову функції **довжина (x)**, аргументом якої є список, а результатом – кількість елементів цього списку на найвищому рівні. Тобто результат дії функції такий:

X	довжина(X)
(A)	1
NIL	0
(A B C)	3
((A B) (C D))	2
(A (B C) K)	3
(((NIL)))	1

Очевидно, ця функція не є елементарною. Для її побудови доцільно використати такий алгоритм: послідовно беремо **cdr** від **X** і рахуємо, скільки разів це може бути зроблено, поки не дійдемо до порожнього списку **NIL**. Але, тим не менше, відразу записати функціональне означення не просто. Взагалі кажучи, при побудові функцій доцільно виділяти два випадки: **X** є **NIL** і **X** не є **NIL**. В другому випадку доцільно застосовувати **cdr(x)** і, оскільки її результат є списком, то до нього можна знову звернутися рекурсивно за іменем функції, яка підлягає визначенню. Рекурсія обов'язково закінчиться, оскільки з кожним викликом довжина аргументу зменшується на одиницю, і рано чи пізно ми прийдемо до порожнього списку.

Для побудови нашої функції **довжина(x)** перший випадок **X** є **NIL** тривіальний. Якщо **X** не є **NIL**, то можна допустити наявність результату рекурсивного виклику для аргументу **cdr(x)**. Вважаючи, що довжина **cdr(x) = n**, можна зробити висновок, що **довжина(x) = n+1**. Наведені міркування дають змогу сформулювати такий алгоритм побудови функції:

випадок 1) **X = NIL**
довжина(x) = 0
випадок 2) **X ≠ NIL**
нехай довжина (cdr (x)) = z
тоді довжина (x) = z+1

Записаний алгоритм приводить до функціонального означення у вигляді:

довжина (x) ≡ якщо рівно (x, NIL), то 0
інакше плюс (довжина (cdr (x)), 1)

У символьній формі маємо таке подання функції:

(DE DOV(X) (COND((EQ X NIL) 0) (T (PLUS(DOV(CDR X)) 1)))

Цей метод до побудови рекурсивних функцій можна застосувати для означення широкого класу функцій. Так може бути визначена аналогічна функція **сума(x)**, яка як аргумент бере список цілих чисел, а як результат видає суму цих чисел. Оскільки при побудові алгоритму визначення цієї функції не з'являється ніяких нових елементів, відразу запишемо означення функції:

сума (x) ≡ якщо рівно (x, NIL), то 0
інакше плюс (car (x), (сума (cdr (x))

Розглянемо побудову рекурсивної функції **з'єднати(x,y)**, аргументами якої є два списки **X** і **Y**, а результатом єдиний список, в якому перераховані всі елементи списків **X** і **Y**:

X	Y	(CONS X Y)
(A B C)	(D E)	(A B C D E)
(A B)	(C D E)	(A B C D E)
NIL	(A B C)	(A B)
(A)	NIL	(A NIL)
NIL	NIL	( )

Оскільки ця функція є функцією двох аргументів, кожен з яких може бути списком, то, очевидно, необхідно передбачити аналіз всіх можливих варіантів, який приводить до такого алгоритму побудови функції:

випадок 1)
X = NIL
підвипадок 1.1)
Y = NIL
з'єднати (X, Y) = NIL
підвипадок 1.2)
Y ≠ NIL
з'єднати (X, Y) = Y
випадок 2)

X ≠ NIL

підвипадок 2.1)

Y=NIL

з'єднати (X, Y) = X

підвипадок 2.2)

Y ≠ NIL

нехай з'єднати (CDR (X), Y) = Z

тоді з'єднати (X, Y) = CONS (CAR (X), Z)

Перші три випадки алгоритму є тривіальними і одразу можна записати відповідь. У підвипадку (2.2) використовується рекурсія. Тут можна брати **CDR** як від **X** так і від **Y**. По суті, є три кандидати на обчислення значення функції за допомогою рекурсивних викликів

з'єднати (CDR (X), Y)

з'єднати (CDR (X), CDR (Y))

з'єднати (X, CDR (Y))

Ми обгрунтували справедливість використання першого кандидата. Функціональне означення згідно з наведеним алгоритмом можна записати так:

з'єднати (X,Y) ≡ якщо рівно (X, NIL) то
якщо рівно (Y, NIL) то NIL інакше Y
інакше
якщо рівно (Y, NIL) то X інакше
CONS (CAR(X), з'єднати (CDR(X), Y))

Оскільки при **X=NIL** **з'єднати (X, Y) = Y** незалежно від значення **Y**, означення можна переписати:

з'єднати (X, Y) ≡ якщо рівно (X, NIL) то Y інакше
якщо рівно (Y, NIL) то X інакше
CONS (CAR (Y), з'єднати (CDR (X), Y))

Якщо тепер врахувати, що при **X ≠ NIL** незалежно від **Y** результатом функції є **CONS(CAR(X)**, **з'єднати (CDR (X), Y)**, то перевірку по **Y** теж можна опустити і записати визначення функції у вигляді:

з'єднати (X,Y) ≡ якщо рівно (X, NIL) то Y інакше
CONS (CAR (X), з'єднати (CDR (X), Y))

Всі ці три версії означають еквівалентні функції і дають один і той самий результат. Першій версії можна віддати перевагу за те, що вона явно перераховує всі випадки. Третій – за її короткість. Але друга і третя версії цієї функції **з'єднати** мають різну ефективність. Друга версія уникає вирахувань, якщо **Y=NIL,** за рахунок зайвих перевірок, коли **Y<>NIL**. Третя версія уникає повторних перевірок по **Y**, але вимагає перебудовувати **X** навіть, якщо **Y=NIL**.

4.4.2. Вибір підфункцій при рекурсивних визначеннях

Часто для побудови оптимальної функції доцільно вводити додаткові функції. Так, для визначення функції **з'єднати (X, Y)** можна було б використати проміжну функцію **з'єд (X, Y)**, яка з'єднує **X** і **Y** у припущенні, що **Y<>NIL**. Тоді приходимо до визначення:

з'єднати (X, Y) = якщо рівно (Y, NIL) то X інакше з'єд (X, Y)
з'єднати (X, Y) = якщо рівно (X, NIL) то Y
інакше CONS (CAR (X), З'ЄД (CDR (X), Y))

Відзначимо, що тут об'єднані переваги другої і третьої версій функції **з'єднати**.

Це загальноприйнятий прийом у функціональному програмуванні, коли на шляху побудови головної програми визначаються нові функції в термінах старих. Отже, функційна програма складається з множин функцій, які визначені одна через другу. Ця функція, яку програміст планує визначити, є головною програмою, першопричиною інших, а всі інші функції служать для неї підпрограмами.

Проблема вибору підфункцій при розробці головної функції є центральною проблемою структурування програми. Іноді стандартні підфункції виявляються самі по собі, але частіші випадки, коли вдалий підбір підфункцій спеціального призначення дає змогу спростити структуру множин функцій загалом. Для того, щоб побудувати добре структуровану програму, можна дати одну пораду: намагатися постійно покращувати те, що вже зроблено.

Цікавою з погляду вибору підфункцій є побудова функції **обернути(X)**, результатом якої є список з елементами, перерахованими у зворотній послідовності.

Спосіб дії цієї функції ілюструє такий приклад:

X	**обернути(X)**
(A B C) ((A B) (C D)) NIL	(C B A) ((C D) (A B)) NIL

Відзначимо, що якщо елементи списку, своєю чергою, є списками, то елементи останніх не обертаються.

Для побудови цієї функції необхідно проаналізувати такі випадки: якщо **X=NIL,** то **обернути (X)** теж **NIL**. Коли **X <> NIL** можна використати **обернути (CDR (X))** і щоб закінчити побудову, необхідно помістити елемент **CAR (X)** в кінець списку. З такого погляду корисна функція **добавити (X, Y)**, яка отримує список **X** і елемент **Y**, і робить **Y** новим останнім елементом списку **X**:

X	Y	**добавити(X, Y)**
(A B) ((A B)(C D)) NIL	C (E F) A	(A B C) ((A B) (C D) (E F) (A)

Вважаючи, що ми маємо функцію **добавити**, можна дати алгоритм побудови функції **обернути (X)**:

випадок 1) **X=NIL обернути (X) = NIL**
випадок 2) **X<>NIL**
нехай обернути (CDR (X)) = Z
тоді обернути (X) = добавити (Z, CAR (X))
Тоді функціональне означення **обернути (X)** має вигляд:

обернути (X) = якщо рівно (X, NIL) то NIL інакше
добавити (обернути (CDR (X)), CAR(X))

Залишилося побудувати функцію **добавити (X, Y)**. Її побудова нагадує попередню розробку. Тому відразу дамо функціональне означення:

добавити (X, Y) = якщо рівно (X, NIL) то CONS (Y, NIL)
інакше CONS(добавити(CDR(X),Y), CONS(Y, NIL))

Тепер визначення функції **обернути (X)** використовує **добавити(X, Y)**, як підфункцію, і ці дві функції разом утворюють програму обертання списку. Цю функцію **добавити** ми могли б визначити через **з'єднати** так:

добавити (X, Y) = з'єднати (X, CONS (Y, NIL)

Тоді б програма обертання складалася із трьох підфункцій. Однак доцільніше виклик **з'єднати** безпосередньо включити в означення функції **обернути** і позбутися функції **добавити**. Отримаємо визначення :

обернути (X) ≡ якщо рівно (X, NIL) то NIL інакше
з'єднати(обернути(CDR(X)), CONS (CAR (X), NIL))
з'єднати (X, Y) ≡ якщо рівно (X, NIL) то NIL інакше
CONS (CAR (X), з'єднати (CDR (X), Y))

Цікаво підрахувати кількості викликів функції **cons** в **обернути**. Функція **з'єднати (X, Y)** викликає себе рекурсивно **n** разів, а значить і **n** разів викликатиметься **cons**. Функція **обернути** також викликає себе **n** разів, значить і **n** разів буде викликаний **cons**. З іншого боку вона **n** разів звертається до функції **з'єднати** і для кожного з цих викликів довжина її першого аргумента дорівнює відповідно **0, 1, 2 ,..., n-1**. Тому кількість викликів **cons** функцією **з'єднати** від імені **обернути** буде

$$1+2+3+...+n-1 = n*(n-1)/2$$

А загальна кількість викликів **cons** в функції **обернути** буде

$$n+n*(n-1)=n*(n+1)/2$$

Очевидно, слід було чекати, що кількість цих викликів буде **n**. Наведена кількість викликів видається надмірним, якщо врахувати, що кожний виклик функції **cons** є дорогим. Надалі, використовуючи метод параметрів нагромадження при побудові функціональних програм, покажемо, як можна оптимізувати побудову функції **обернути (X)**, щоб зробити її ефективною з мінімальною кількістю викликів функції **cons**.

4.4.3. Контрольні питання та вправи

1. Дати визначення поняття рекурсії.
2. Побудувати функцію, яка видає останній елемент списку.
3. Розробити рекурсивний предикат, який перевіряє чи введене у вигляді списку слово є паліндромним.

4. Які є критерії перевірки правильності роботи рекурсії?

5. Характерні помилки при побудові рекурсивних функцій?

6. Дати означення функції, яка на основі заданого списку формує список з елементів, що стоять на парних позиціях.

4.5. Метод параметрів нагромадження та локальні означення у функціональному програмуванні

4.5.1. Суть методу з параметрами нагромадження

Ідея методу з параметрами нагромадження полягає в тому, щоб визначити допоміжну функцію з додатковим параметром, який використовується для нагромадження бажаного результату.

Проілюструємо суть цього методу, заново програмуючи функцію **обернути(x)**. Для цього введемо додаткову функцію **обр (x,y)**, де **x** – список, який підлягає обертанню, а **y** – додатковий параметр, який накопичує обернений список. Дамо таке означення:

обр (x, y) ≡ якщо рівно (x, NIL) то y
інакше обр (cdr (x), cons (car (x), y))

Через цю функцію можна визначити функцію **обернути (x)**:

обернути (x) = обр (x, NIL)

Легше зрозуміти як діє ця функція, ніж описати алгоритм її побудови. Коли викликана функція **обр (x,y)**, то список **y** нагромадив в собі всі розглянуті елементи списку, які підлягають обертанню. Отже, якщо **x** є **NIL**, то в **y** міститься весь обернутий список, а якщо **x** не є **NIL**, то ми можемо в **y** нагромадити **car (x)** і знову рекурсивно викликати **обр** для обробки **cdr (x)**. Наведемо таблицю послідовних викликів функції **обр (x,y)**, якщо перший раз функція **обернути (x)** звертається до списку **(A B C D)**:

X	Y
(A B C D)	NIL
(B C D)	(A)
(C D)	(B A)
(D)	(C B A)
NIL	(D C B A)

По суті, ми навгад записали означення цієї функції, а потім описали, як вона працює. Для того, щоб застосувати загальноприйняту методику до побудови функцій, необхідно обґрунтувати вигляд результату **обр (x, y)** при довільному **y**. Нехай результатом функції **обр (x,y)**, коли **x** і **y** є списками, можливо порожніми, є список всіх елементів **x**, які взяті в зворотній послідовності і які доповнені всіма елементами **y** в їх початковій послідовності. Тобто формально ми можемо записати:

обр (x, y) = з'єднати (обернути (x), y)

Хоча це визначення і не зовсім підходить, оскільки невідомою є сама функція **обернути**. Тим не менше, тепер можна записати алгоритм побудови функції:

випадок (1): **x = NIL обр (x, y) = y**
випадок (2): **x ≠ NIL**
нехай обр (cdr (x),z) = з'єднати (обернути (cdr(x),z) тоді
обр (x, y) = з'єднати (обернути (x), y) =
= з'єднати (обернути (cdr (x)), cons (car (x),y)) =
= обр (cdr (x), cons (car (x),y))

Наведений алгоритм швидше є доведенням справедливості вищеозначеної функції, ніж способом її побудови, оскільки в другому випадку перетворення є доволі складними.

Проблема побудови ефективних функціональних програм може бути успішно вирішена вдалим підбору підфункцій з додатковим параметром. Дійсно, якщо підрахувати кількість викликів конструктора в наведеній версії функції **обернути**, то їх буде рівно **n**, якщо довжина списку **x** є **n**. Якщо порівняти цю кількість викликів конструктора з кількістю викликів **n*(n-1)/2** у разі визначення функції **обернути (x)** без параметрів нагромадження, то маємо значну економію машинних ресурсів.

Розглянемо типовіше використання методу параметрів нагромадження, якщо є потреба використати два параметри для нагромадження результату. Побудуємо функцію **сумдобуток (x)**, яка як результат видає двочлен, в якого перший елемент є сумою, а другий добутком елементів списку **x**. Тобто хочемо визначити функцію

сумдобуток (x) = двочлен (сума (x), добуток (x))

Для цього, щоб це зробити, введемо допоміжну функцію **сд (x, c, g)** з двома параметрами **c** і **g**, які відповідно нагромаджують суму і добуток елементів списку **x**. Іншими словами, хочемо визначити функцію **сд** у вигляді

сд (x, c, g) ≡ двочлен (s + сума (x), p*добуток (x))

і тоді можна визначити

сумдобуток (x) ≡ сд (x, 0, 1)

Функцію **сд (x, s, p)** можна побудувати, аналізуючи випадки по **x**:

випадок 1) **x = NIL**
сд (x, s, p) = двочлен (s, p)
випадок 2) **x ≠ NIL**
нехай сд (cdr (x),c',g') = двочлен (s' + сума
(cdr(x)), p' · добуток (cdr (x))
тоді сд (x,c,g) = двочлен (c + сума (x), g ·
добуток (x)) = двочлен (c + car (x) +
сума (cdr (x)), g* car (x)* добуток
(cdr (x)) = сд (cdr(x), c + car(x), g* car(x))

Цей алгоритм дає змогу відразу записати визначення функції **сд**:

сд (x, c, g) ≡ якщо рівно (x, NIL) то двочлен (s, p)
інакше сд (cdr (x), c + car (x), g* car (x))
двочлен (x, y) ≡ cons (x, cons (y, NIL))

Як бачимо, ми отримали винятково просту і цікаву функцію. Якщо б ми хотіли визначити цю функцію без параметрів нагромадження, то, аналізуючи можливі випадки по **x,** прийшли б до алгоритму

випадок 1) **x = NIL**
сумдобуток (x) = двочлен (0, 1)
випадок 2) **x ≠ NIL**

нехай сумдобуток (cdr (x)) = z
тоді сумдобуток (x) = двочлен (car (x) +
car (z), car (x) · car (cdr (z)))

Приходимо до такого означення функції **сумдобуток (x)** без параметрів нагромадження

сумдобуток (x) ≡ якщо рівно (x, NIL) то двочлен (0,1) інакше
двочлен (car (сумдобуток (cdr (x))) + car (x),
car (cdr (сумдобуток (cdr (x)))) · car (x))

Як бачимо, при такому визначенні необхідно виконати масу розчленувань списку і будувати **2*n + 1** двочлен, де **n** – розмірність списку **x**. Водночас використання підфункції з параметрами нагромадження потребує побудови лише одного **двочлена**, в якому буде записаний остаточний результат.

Ми вже говорили про те, що означаючи рекурсивні функції, доцільно вводити додаткові підфункції, що забезпечує побудову більш ефективної програми. Якщо ввести функцію **нагромадити**, то останнє означення **сумдобуток (x)** виглядало би так:

сумдобуток (x) ≡ якщо рівно (x, NIL) то двочлен (0, 1)
інакше нагромадити (car (x), сумдобуток (cdr (x))
нагромадити (n, z) = двочлен (car (z) + n, n* (car (cdr (z)))

Вже в цій версії необхідно будувати **n+1** двочлен, тобто кількість викликів **cons** зменшується.

4.5.2. Прості та рекурсивні локальні форми

Окрім введення додаткових підфункцій, для покращення ефективності функціональних програм є ще один спосіб: введення локальних означень за допомогою форм **нехай** і **де**. У визначенні функції **сумдобуток (x)** без додаткових функцій і параметрів нагромадження кількість рекурсивних викликів функції **сумдобуток (x)** зростає за експоненціальним законом до $\mathbf{2^n}$, де **n** елементів списку **x**. Використовуючи форми **нехай** і **де**, це означення перепишеться:

сумдобуток (x) = якщо рівно (x, NIL) то двочлен (0, 1) інакше
{нехай z = сумдобуток (cdr (x))
двочлен (car (x) + car (z), car (cdr (z)) · car (x))}

або

сумдобуток (x) = якщо рівно (x, NIL) то двочлен (0, 1) інакше
{двочлен (car (x) + car (z), car (cdr (z)) · car (x))
де z = сумдобуток (cdr (x))}

Обидва означення є еквівалентними і вибір будь-якого з них є справою смаку. Областю дії цього локального визначення змінної **z** є область, взята в фігурні дужки. Форма **нехай** дає визначення раніше ніж воно використовується, в той час як форма **де** дає означення після його використання. При такому визначенні значення **z** обчислюється один раз і потім двічі застосується у визначальному виразі. Ось ці локальні форми **нехай** і **де** дають форми, аналогічні блочній структурі в Алголі. Використовуючи їх, ми можемо зробити визначення функції **сумдобуток (X)** подібнішим на версію з двома параметрами:

сумдобуток (X) ≡ якщо рівно (X, NIL) то двочлен(0, 1) інакше
{ нехай Z = сумдобуток (cdr (x))
і n=car (x)
двочлен (n + car (z), n · car (cdr (z)))}

Локально визначені функції в області їх дії відділені розділювачем **і** .

Іноді ці локальні означення використовуються для того, щоб функціональну програму зробити не тільки ефективнішою, але й наочною і

читабельною. Так, вводячи локальні змінні **c** і **g** для часткової суми і часткового добутку від **z,** отримуємо означення

сумдобуток (X) ≡ якщо рівно (X, NIL) то двочлен (0, 1) інакше
{ нехай Z = сумдобуток (cdr (x))
і n = car (x)
{нехай c = car (x)
g = car (cdr (z))
двочлен (n + c, n · g)}}

Тут необхідно взяти в дужки визначення **c** і **g**, оскільки вони зсилаються на значення **z**. Пояснення використання подвійних дужок полягає в тому в тому, що якщо в області **нехай** або **де** є ряд визначень, то вони доступні лише у визначальному виразі, але не в самих визначеннях. В зв'язку з цим, щоб зробити доступним **z** для змінних **c** і **g**, ми зробили їх більш локальними. Загалом форма **нехай** може бути подана так:

{нехай x1 = e1
і x2 = y2
.....
і xi = ei
e}

В останньому представлені **x1, x2, ..., xi** використовуються для позначення локально означуваних змінних; **e1, e2, ..., ei** – для виразів, що їх визначають, **e** – для означуваного виразу.

Усі змінні **x1, ..., xi** доступні лише у виразі **e**, але ніяк не в самих локальних означеннях.

4.5.3. Фрагмент програми символьного диференціювання

Локальні форми широко використовуються в функціональному програмуванні для покращення як ефективності, так і наочності програми.

Зокрема ці засоби дають змогу побудувати доволі просту функцію для символьного диференціювання. Будемо оперувати з формулами, в які входять операції “+” і “*”, константи і змінні. Позначивши похідну за змінною **X** у вигляді **DX**, запишемо основні правила диференціювання:

DX(X) = 1;
DX(Y) = 0 Y<>X (Y-константа або змінна);
$DX(e_1 + e_2) = DX(e_1) + DX(e_2)$;
$DX(e_1 \cdot e_2) = DX(e_1) \cdot e_2 + DX(e_2) \cdot e_1$

Перш ніж записати функціональне означення цих правил, домовимося про представлення даних такими S-виразами.

константа -> числовий атом
змінна -> символьний атом
$e_1 + e_2$ -> (PLUS E1 E2)
$e_1 \cdot e_2$ -> (TIMES E1 E2)

і введемо дві функції для формування суми і добутку

сума (x, y) = cons (PLUS, cons (X, cons (Y, NIL)));
добуток (x, y) = cons (TIMES, cons (X, cons (Y, NIL)))

Тепер можна безпосередньо записати функціональне означення функції **диф (e)**, в якій безпосередньо будуть реалізовані вищенаведені правила диференціювання:

диф (e) ≡ якщо атом (e) то якщо рівно (e, x) то 1 інакше 0
інакше якщо рівно (car (e), PLUS) то
{сума (диф (e_1), диф (e_2))
де e_1 = car (e)
і e_2 = car (cdr (e))}
інакше якщо рівно (car (e), TIMES) то
{сума (добуток (e_1, диф (e_2)),
добуток (e_2, диф (e_1)))
де e_1 = car (e)
і e_2 = car (cdr (e))} інакше помилка

Оскільки аналіз випадків за **e** є очевидним, то детальний опис цієї функції не наводиться. Функція побудована так, що видає атом **помилка** при отриманні неправильного аргументу. Правила диференціювання представлені коротко і можуть бути при потребі розширені.

4.5.4. Контрольні питання та вправи

1. Суть методу параметрів нагромадження у функціональному програмуванні.
2. Використовуючи параметри нагромадження, побудувати функцію **обернути (x)**, яка здійснює обертання елементів підсписків, що входять в **x**.
3. Яка причина використання локальних означень в функціональному програмуванні та область їх дії?
4. Яка різниця між підфункціями і локальними означеннями?
5. Чи є принципова відмінність форм **нехай** і **де** ?
6. Побудувати програму диференціювання, в яку б входили арифметичні операції ділення, додавання, піднесення до степеня.
7. Побудувати програму диференціювання виразів з тригонометричними функціями cos, sin, tg, ctg, попередньо визначивши ці функції.
8. Визначити функції exp, ln, log та побудувати програму диференціювання виразів, в які входять ці функції.

4.6. Типове використання S-виразів

4.6.1. Алгоритм, функціональне означення S вираз функції набір

Розглянемо побудову програми в середовищі S-Ліспу, починаючи з запису алгоритму програми, її функціонального означення та представлення у вигляді S-виразу.

Як перший приклад побудуємо функцію **набір (t)**, яка як аргумент бере список **t** атомів, а як результат – видає список, в якому кожен атом з **t** має одне входження. Таке використання списку, який формує ця функція, є типовим використанням S-виразів. Результат функції **набір (t)** є множиною, яка представлена списком атомів без повторення. Для задання алгоритму побудови функції **набір (t)**, можливі випадки по **t**:

випадок 1) **t = NIL : набір (t) = NIL**
випадок 2) **t ≠ NIL :**
нехай набір (cdr (t)) = z
тоді набір (t) = включити (car (t), z)

У наведеному алгоритмі випадок **t = NIL** є тривіальним і відразу дає змогу записати результат. В другому випадку, допустивши наявність множини атомів з **cdr (t)** для отримання всього результату необхідно добавити атом **car (t)** до **z**, якщо він там ще не зустрічався. З цією метою використовуємо функцію **включити (e, S)**, де **e** – атом, **S** – список, яку ще необхідно побудувати. Тоді функціональне означення набуває вигляду:

набір ≡ якщо рівно (t, NIL) то NIL інакше
включити (car (t), набір (cdr (t)))

Побудова функцій **включити** є очевидною, якщо допустити наявність функції **елемент (e, S)**, яка перевіряє наявність атома **e** в списку **S**. Обмежимося функціональним означенням функції **включити (e, S)**:

включити (e, S) ≡ якщо елемент (e, S) то S інакше cons (e, S)

Залишилося побудувати функцію **елемент (e, S)**. Як звичайно, запишемо алгоритм, виходячи з аналізу випадків по **S**:

випадок 1) **S = NIL : елемент (e, S) = NIL**
випадок 2) **S ≠ NIL :**
нехай елемент (x, cdr (S)) = b
тоді елемент (e, S) = якщо b то T інакше
якщо e = cons (S) то T
інакше NIl

В приведеному алгоритмі **T** – означає істинність.
Цей опис приводить до означення

елемент (e, S) ≡ якщо рівно (S, NIL) то NIL інакше
якщо елемент (e, car (S)) то T інакше
якщо рівно (e, car (S)) то T
інакше NIL

Це означення є дещо неефективним і, звичайно, перевпорядкування двох останніх рядків визначення функції **елемент (e, S)** дасть змогу уникнути пошуку по всьому списку, якщо **x = car (S)**. Якщо врахувати, що вираз

якщо b то T інакше NIL

завжди має те саме значення, що і **b**, якщо **b** набуває логічне значення, то функція **елемент (e, S)** приводиться до вигляду

елемент (e, S) ≡ якщо рівно (S, NIL) то NIL інакше
якщо рівно (e, car (S)) то T інакше
елемент (e, cdr (S))

Цим закінчується визначення функції **набір (t)**, яка включає в себе три функції: **набір**, **включити**, **елемент**. Її представлення у вигляді S-виразів є такими

(DE NABIR (S) (COND ((E Q S NIL) NIL) (T
(VKL (NABIR (CDR S)) (CAR S)))))
(DE VKL (E S) (COND ((ELEMENT E S) S)
(T (CONS E S))))
(DE ELEMENT (E S) (COND ((E Q S NIL) NIL)
((E Q E (CAR S)) T))) (T (ELEMENT E (CDR S)))))

В наведеному визначенні S-виразу функціональним іменам **набір**, **включити**, **елемент** відповідають атоми S-виразів **NABIR**, **VKL**, **ELEMENT**.

Розглянемо деякі узагальнення функції **набір (t)**. Розширимо функцію **набір (t)** так, щоб вона була працюючою і якщо елементи списку **t** можуть бути списки з підсписками, вкладеними на довільну глибину. Для запису алгоритму побудови такої функції маємо такий аналіз випадків:

випадок 1) **t = NIL : набір (t) = NIL**
випадок 2) **t ≠ NIL :**
нехай набір (cdr (t)) = z
підвипадок 2.1) **car (t) є атом**

тоді набір (t) = включити (car (t), S)

підвипадок 2.2) **car (t) є список**

тоді набір (car (t)) = S'

тоді набір (t) = об'єднати (S, S')

Як і раніше, випадок **t=NIL** є тривіальним. У другому випадку **t ≠ NIL** маємо два підвипадки. Коли **car (t)** – атом, то маємо той самий результат, що і раніше. Якщо **car (t) – о**писок, то слід викликати **набір (cdr (t))**, щоб визначити множину атомів, які входять в **car (t)**. На основі множин **S** і **S'** слід сформувати єдину множину, куди входили всі елементи **S** і ті з **S'**, яких немає в **S**. Цю функцію назвемо **об'єднати (u, v)** і вона підлягає визначенню. Виконавши типовий аналіз випадків, приходимо до алгоритму

випадок 1) **u = NIL : об'єднати (u, v) = v**

випадок 2) **u ≠ NIL :**

нехай об'єднати (cdr (u), v) = z

тоді об'єднати (u, v) = включити (car(u), z)

Отже, функція **набір (S)** для свого визначення потребує функцій **об'єднати (u, v)**, **елемент (e, S)** та **включити (e, S),** які були наведені раніше.

Отже, маємо таке функціональне визначення:

набір (t) ≡ якщо рівно (t, NIL) то NIL
інакше якщо атом (car (t)) то
включити (car (t), набір (cdr (t)))
інакше об'єднати (набір (car (t)), набір
об'єднати (u, v) = якщо рівно (u, NIL) то v
інакше включити (car (u), об'єднати (cdr (u), v))
включити (e, S) ≡ якщо елемент (e, S) то S
інакше cons(e, S)
елемент (e, S) ≡ якщо рівно (S, NIL) то NIL інакше
якщо рівно (e, car (S)) то I інакше
елемент (e, cdr (S))

Відповідне S-представлення програми має вигляд

(DE NABIR (S) (COND ((E Q S NIL) NIL)

((ATOM (CAR S)) (V K L (CAR S))
(NABIR (CDR S))) (T (O B D (NABIR (CAR S))
(NABIR (CDR S))))))
(DE OBD (U V) (COND ((E Q NIL) V)
(T (VKL (CAR U) (O B D (CDR U) V)

S-вирази функцій **елемент** і **включити**, наведені в попередній реалізації програми **набір (S)**. Для імені функції **об'єднати** використано атом **OBD.**

4.6.2. Приклад побудови функції індив

Як другий приклад типового використання S-виразів розглянемо побудову функції **індив (t)**, яка, будучи застосованою до простого списку атомів з **t**, формує список атомів, які в **t** зустрічаються лише один раз. Аналіз можливих випадків по **t** є традиційним і приводить до такого алгоритму:

випадок 1) **t = NIL : індив (t) = NIL**
випадок 2) **t ≠ NIL :**
нехай індив (cdr (t)) = z
і елемент (car (t), cdr (t)) = b
тоді індив (t) = якщо b то вилучити (car (t), z)
інакше z

Перший випадок є тривіальним. В другому випадку, допустивши побудову функції **індив** для коротшого списку без першого елементу, необхідно з результату **z** вилучити елемент **car(t)**, якщо він міститься в списку **cdr (t)**. В противному разі **z** залишиться без зміни. Для цієї мети необхідно побудувати функцію **вилучити (e, S)**, яка вилучає елемент **e** із списку **S**. Алгоритм побудови цієї функції аналогічний до попередніх записів і має вигляд:

випадок 1) S = **NIL : вилучити (e, S) = NIL**
випадок 2) **S ≠ NIL :**
підвипадок 2.1) **e = car (S): є атом вилучити (e, S) = cdr (S)**
підвипадок 2.2) **e ≠ car (S):**
вилучити (e, S) = cons (car (S), вилучити (e, cdr (S))

Перший випадок є очевидним. Для не порожнього списку **S** і збігу першого елемента **S** з **e** як результат функції **вилучити** видаємо **cdr (S)**. Якщо **e** не збігається з першим елементом з **S**, то рекурсивно викликаємо функцію **вилучити** до списку **cdr (S)**, зберігаючи перший елемент з **S** в результуючому списку застосовуючи операцію **cons**. Гарантією закінчення рекурсії є те, що якщо **e** не міститься в списку **S**, то рано чи пізно ми прийдемо до порожнього списку і як результат після поелементного перегляду буде виданий весь список **t**.

Отже, приходимо до такого функціонального визначення:

індив (t) ≡ якщо рівно (S, NIL) то NIL
інакше якщо елемент (car (S), cdr (S))
то вилучити (car (S), індив (cdr (S))) інакше
індив (cdr (S))
вилучити (e, S) ≡ якщо рівно (S, NIL) то NIL інакше
якщо рівно (car (S), e) то cdr (S) інакше
cons (car (S), вилучити (e, cdr (S))
елемент (e, S) ≡ ...

Функція **елемент** розроблена раніше, тому її функціональне означення тут не наводиться. Запис наведеної функції в середовищі S-Ліспу матиме такий вигляд:

(DE IND (S) (COND ((E Q S NIL) NIL) ((ELEMENT
(CAR S) (CDR S)) (VID (CAR S) (IND (CDR S))))
(T (IND (CDR S)))))
(DE VID (E S) (COND ((E Q S NIL) ((E Q (CAR S) E)
(CDR S)) (T (CONS (CAR S) (VID E (CAR S)))))

Як функційні імена **індив** та **вилучити** в середовищі S-Ліспу вибрані атоми **IND** та **VID**. **S**-вираз для функції **елемент** тут не наводиться, оскільки він відповідає **S**-виразу для **ELEMENT**, який записаний при розробці функції **набір (S)**. Але слід мати на увазі, що при записі тієї чи іншої функціональної програми в S-Ліспі для її експлуатації необхідно задати S-вирази всіх підфункцій, які вона використовує. Можна узагальнити побудовану функцію **індив (t)** для довільних списків **S**, виходячи з традиційного запису функцій в

термінах **cdr**. Але такий підхід приводить до побудови малоефективної програми. Використовуючи метод з параметрами нагромадження, можна значно покращити ефективність цієї функції. Більше того, використовуючи можливість представлення множин значень за допомогою функцій, можна побудувати ефективну і елегантну функцію **індив (S)**, що й далі буде продемонстровано після введення крапкової форми представлення S-виразів.

4.6.3. Контрольні питання та вправи

1. Навести алгоритм, функціональне визначення та S-вираз функції, яка вилучає всі входження елемента **e** з простого списку **S**.
2. Узагальнити функцію п.1 для довільних списків **S**.
3. Побудувати функцію **часто (S)**, яка дає кількість входжень простого елемента з **S** в описок **S**, якщо він є простий.
4. Дати функціональне означення **кількість (S)**, яка дає числа входжень кожного елемента з **S** в цей список.
5. Побудувати функцію **частота (S)**, яка формує список пар, де на першому місці стоїть елемент, а на другому – кількість його входжень в список **S**.
6. Показати, як можна використати функцію, побудовану в п. 5 для визначення функції п. 4.
7. Вважаючи заданою умовну форму **cond (e)**, дати визначення простішої форми **якщо (a1, e2, e3),** де **e → ((A1 E2) (T E3).**
8. Забезпечити визначення форми **cond (e)**, маючи задану форму **якщо (a1, e2, e3)**, де кількість пар в **e** наперед не відома.

Розділ 5. Класифікація функцій вищих порядків та їх застосування до побудови функціональних програм

5.1. Функції вищих порядків та лямбда-вирази і способи їх використання

5.1.1. Побудова функцій першого роду

Функції вищих порядків – це такі функції, які мають інші функції як свої аргументи або як результати. Для ряду програм побудова цих функцій є доволі складним завданням. Але той факт, що побудована у такий спосіб програма є значно коротшою і зрозумілішою, виправдовує витрачені зусилля.

Якщо б ми хотіли визначити функцію, яка, будучи застосованою до списку цілих чисел, повертає цей список, але з елементами, збільшеними на одиницю щодо вхідного списку, ми б отримали таке функціональне означення:

збіль (X) = якщо рівно (X, NIL) то NIL інакше
cons (car (X) + 1, збіль (cdr (X)))

Цілком аналогічно функція:

зал (X) = якщо рівно (X, NIL) то NIL інакше
cons (car (X) зал 2, зал (cdr (X)))

вираховує по вихідному списку **X** список залишків від ділення кожного його елемента на 2.

Подібність цих двох означень і здатність уявити собі можливість побудови багатьох інших аналогічних функцій дає змогу дати означення деякої узагальненої функції **відобразити**, де та конкретна операція, яка виконується над кожним елементом списку, є параметром цієї функції. Приходимо до такого означення:

відобразити (X, f) = якщо рівно (X, NIL) то NIL інакше
cons(f (car(X)),відобразити(cdr (X),f))

Це означення є точною копією попередніх означень з тією відмінністю, що операція, яка застосовується до кожного елемента списку **X** є

неконкретна, а деякий параметр **f**. Тепер ми можемо дати означення функції **збіль (X)** і **зал (X)** через **відобразити (X, f)**:

зб (Z) = Z + 1
збіль (X) = відобразити (X, зб)
зал1 (Z) = Z зал2
зал (X) = відобразити (X, зал1)

Ця функція **відобразити** є функцією вищого порядку, оскільки вона використовує іншу функцію як аргумент.

Як другий приклад функції вищого порядку розглянемо побудову функції **редукція (X, G, A)**, де **X** – список елементів, **G** – бінарна функція (функція двох аргументів), **A** – константа, яка приводить (редукує) список **X** до значення

G (X1, G (X2, G (X3, ..., G (Xk, A) ...)))

Алгоритм побудови цієї функції є типовим шляхом аналізу можливих випадків по **X**:

випадок 1) **X=NIL редукція (X, G, A) = A**
випадок 2) **нехай редукція (car (X), G, A) = Z**
тоді редукція (X, G, A) = G (car (X), Z)

Цей алгоритм дає змогу записати таке означення:

редукція (X, G, A) = якщо рівно (X, NIL) то NIL інакше
G (car (X), редукція (cdr (X), G, A).

Маючи цю функцію, можна тривіально визначити функції **сума(X)** і **добуток (X)**, які визначать відповідно суму і добуток елементів списку **X**:

пюс (X, Y) = X + Y
редукція (X, плюс, 0) = сума (X)
множ (X, Y) = X · Y
редукція (X, множ, 1) = добуток (X)

Цікавим є те, що, використовуючи цю функцію, досить просто можна визначити функцію **сумдобуток (x)**, результатом якої є двочлен, перший елемент якого є сумою елементів списку **X**, а другий елемент – добутком елементів списку **X**. Якщо означити функцію:

нагромадити (n, z) ≡ двочлен (n + car (z), n · car (car (z))) то
сумдобуток (x) ≡ редукція (x, нагромадити, двочлен 0, 1))

Деяким недоліком при побудові функцій вищих порядків є необхідність задавати імена функціям, які використовуються як фактичні параметри. Досі, будуючи нові функції, ми задавали їх так

f (X1, ..., XN) ≡ e,

де **f** – ім'я функції, яка визначається; **X1 ... XN** – список її аргументів; **e** – визначальний вираз або тіло функції. **f** є глобальне і його можна використовувати всюди, де виникає потреба, в той час коли змінні **X1 ... XN** є локальними. Якщо б ми захотіли зробити визначення функції **f** локальним, то виявляється, для цього ми не маємо засобів. Цей недолік можна усунути, використовуючи так звані λ-вирази.

5.1.2. Поняття про λ-вирази та їх застосування

λ-вираз є таким виразом, значенням якого є функція. В вищенаведеному означенні функції з ім'ям **f** зв'язується (а не присвоюється) значення, яке є функціональним. Це функціональне значення може бути задано наступним λ-виразом

λ (X1, X2, ..., XN) e.

Функція є правилом для визначення її значення за заданими аргументами. λ-вираз містить собі це правило, виділяючи аргументи і вираз в термінах цих аргументів. Так λ-вираз

λ (z) z + 1

вираховує функцію, яка, будучи викликана при конкретному аргументі, видає

значення, збільшене на одиницю.

Тобто це функція, яку раніше ми назвали **зб (x)**. Звідси знаходимо негайне застосування λ-виразів – використовувати їх як фактичні параметри функцій вищих порядків. Тепер ми можемо перевизначити функцію **збіль (x)**:

збіль (x) ≡ відобразити (x, λ (z) z+1)

Аналогічно можна перевизначити такі функції:

зал (x) ≡ відобразити (x, λ (z) z зал 2)
сума (x) ≡ редукція (x, λ (y, z) y+z, 0)
добуток (x) ≡ редукція x, λ (y, z) y · z, 1

λ-вирази мають одне і те саме значення незалежно від імен, які вибрані для їх параметрів, лише щоб вони були різні для різних параметрів. Так вирази

λ (x) x+1
λ (z) z+1

мають одне і те саме значення і є взаємозамінними незалежно від того, де вони зустрічаються.

Дозволяється використання λ-виразів всюди, де потрібне функціональне значення. Зокрема, їх можна використовувати з формами **нехай** і **де** в локально означуваних функціях. Розглянемо такий варіант визначення функції **збіль (x)**:

збіль (x) ≡ {відобразити (x, зб), де зб = λ (z) z+1}

Тут ми не уникнули використання імені **зб**, але звузили область її дії, зробивши цю функцію локальною.

При потребі локального визначення рекурсивно означуваних функцій використовуються модифіковані форми **нехай** і **де**. Будемо називати їх **нехайрек** і **дерек**. Якщо б ми хотіли перевизначити функцію **сумдобуток (x)**, зробивши функцію **сд (x, s, p)** локальною, то це можна реалізувати так:

сумдобуток (X) ≡ {сд (X, 0, 1)
дерек сд = λ (X, S, P)

якщо рівно (X, NIL) то двочлен (0, 1)
інакше сд (cdr (X), S + car (X), P · car (X))}

Різниця між формами **нехай** і **нехайрек** (відповідно між (**де**) і (**дерек**)) в області дії цих форм. Якщо навести загальні вирази для цих форм

{нехай	**{нехайрек**
$\mathbf{x_1 = e_1}$ **i**	$\mathbf{x_1 = e_1}$ **i**
$\mathbf{x_2 = e_2}$ **i**	$\mathbf{x_2 = e_2}$ **i**
.....	**.....**
$\mathbf{x_n = e_n}$	$\mathbf{x_n = e_n}$
e }	**e }**

то імена x_1, ..., x_n в формі **нехай** є доступним лише в виразі **e**, в той же час як у формі **нехайрек** вони є доступними і у виразах $\mathbf{e_1}$, ..., $\mathbf{e_n}$ і у **e**. Ось це скорочення **рек** якраз підкреслює, що всі визначення є взаєморекурсивними. Його слід використовувати, якщо локально визначаються рекурсивні функції і є гарантія того, що всі паралельні обчислення є визначенням функцій.

Все сказане є справедливим і для форми **дерек**.

Розглянемо визначення:

{{X дерек X= X+1} де X = 0}

Оскільки тут використовується форма **дерек**, то **X** у виразі **X=X+1** повинен бути одним і тим самим, як праворуч так і ліворуч. Очевидно, що цей вираз не має змісту. Водночас вираз

{{X де X = X + 1 } де X = 0}

цілком правильний і його значенням буде 1.

Як правило функційна програма складається з множини взаєморекурсивних функцій, одна з яких є головною. Використовуючи ці форми **нехайрек** і **дерек**, можна сформувати вираз, щоб виділити головну програму, а всі додаткові заховати в середині:

{f дерек $\mathbf{f_1}$ **= ...**
$\mathbf{f_2}$ **= ...**
.....
$\mathbf{f_n}$ **= ... }**

5.1.3. Функції вищого порядку другого роду

Другий тип функцій вищих порядків – це функції, які як результат теж видають функцію:

збільш1 (N) ≡ λ (Z) Z+n.

В цьому означенні безумовно **збільш1** є функцією, оскільки потребує задання аргументу **n**. Однак її результат – теж функція. Так, **збільш1 (3)** є функція, яка при виклику видає її аргумент, збільшений на 3. Так, значення виразу:

{f (2) де f = збільш1 (3)}

є число 5. Аналогічно можна перевизначити функцію **збільш (X)**:

збільш1 (X) ≡ відобразити (X, збільш1 (1))

Якщо б ми хотіли зробити визначення функції **збільш1 (X)** локальним, то слід записати таке означення:

збільш1 (X) ≡ {відобразити (X, збільш1 (1)) де збільш1= λ (n)
{λ (Z) Z+n } }

В цьому означенні використовується змінна **N**, яка не є параметром виразу **λ (Z) Z+n**. Такі змінні є глобальними. Виникає проблема зв'язування цих змінних. Цю проблему наочно ілюструє такий приклад:

{нехай n=1 {нехай f =λ (Z) Z+n {нехай n=2 f (3) } } }

Значення цього виразу є 5 або 4 залежно від того, яке значення **N** буде використано при виклику функції **f(3)**.

Спочатку **n=1**, потім вираховується λ-вираз, присвоюючи функції **f** значення **λ (Z) Z+1**, далі змінюється значення **n** і здійснюється виклик функції **f** з фактичним параметром 3. Оскільки **f** присвоюється значення **λ (Z) Z+1**, то результат буде 4.

Така форма зв'язування глобальних змінних, коли вони визначаються в момент вирахування λ-виразу, а не в момент виклику функції, носить назву статичного або простого.

Третім типом функції вищого порядку є функції, які мають функції і в якості аргументу і в якості результату. Визначимо

комп (f, g) ≡ λ (x) f (g (x))

Це так звана композиція або добуток функції. Як аргументів є дві функції і результат – знову функція, яка послідовно застосовує обидві функції. Так:

збобер (x) ≡ комп (збіль, обернути)

є функцією, яка спочатку обертає список, а потім збільшує кожен його елемент на одиницю.

Функції вищих порядків є достатньо складними при їх реалізації строго функційною мовою, однак дають змогу підвищити ефективність функціональних програм.

5.1.4. Контрольні питання та вправи

1. Дати означення функцій вищих порядків.
2. Які є типи функцій вищих порядків?
3. Порівняльні характеристики форм **нехай** і **де** та **нехайрек** і **дерек**.
4. Що таке статичне зв'язування даних?
5. Для чого в функціональному програмуванні використовуються **λ**-вирази?
6. Описати функцію **частота (t)**, яка в якості аргументу бере список атомів **t** і видає список всіх атомів, які зустрічаються в **t** разом з частотою їх появи.
7. Побудувати функцію **відобразити (x, t)**, аргументом якої є список **x**, який розглядається як множина, а результат її використання є знову множина у вигляді списку, кожен елемент якого можна отримати застосовуючи функцію **f** до кожного елемента **x**.
8. Описати функцію **редукція (x, g, a)**, при використанні якої до списку $\mathbf{x = (x_1 \ldots x_n)}$ отримується значення $\mathbf{g (\ldots g (g (a, x_1), x_2) \ldots x_n)}$.

5.2. Крапкове подання S-виразів та використання функцій для зображення структур даних у вигляді множин

5.2.1. Правила спрощення крапкових S-виразів

Необхідність крапкового подання S-виразів виникає через те, що не існує способу подання значення **cons (x, y)**, коли **y** є атом (окрім того випадку, коли **y** є **NIL**). А тому будемо подавати значення **cons (A,B)** за допомогою S-виразу

(A.B)

з крапкою між атомами. Слід відзначити, що записаний S-вираз відрізняється від значення

(A B),

яке є результатом операції **cons (A, cons (B, NIL))**. Отже, тепер ми можемо одне і те ж значення записати у вигляді різного подання. Так значення виразу **cons (A, NIL)** може бути подано або як **(A)** або згідно з крапковою формою подання як **(A.NIL)**. Аналогічна операція має місце при поданні чисел, коли ми вважаємо, що 127 і 0127 представляють одне і те ж значення, ігноруючи 0 перед числом. Насправді це розширення є більш загальним. Наприклад, значення

cons (A, cons (B, cons (C, NIL)))

може бути подано у вигляді **(A B C.NIL)** або **(A B C)**. У загальному випадку вираз **cons (e_1, e_2)** подається крапковою парою **(e_1 . e_2)**, де **e_1, e_2** – правильні S-вирази:

X	Y	CONS (X, Y)
A	B	(A . B)
B	NIL	(B . NIL)
B	(A, NIL)	(B. (A . NIL))
(A, B)	(B,C)	((A, B) . (B, C))
(A, B)	C	((A, B).C)

При крапковому поданні S-виразів використовуються такі скорочення: S-вираз

(V1 .(V2 .(V3 ...(VK . VK1)...)))

записується у вигляді

(V1 V2 ... Vk . Vk1)

Тобто залишаємо лише одну крапку. Далі S-вираз

(V1 V2 V3 ...Vk . Vk1)

записується у вигляді

(V1 V2 V3 ... Vk)

Приведені правила можна узагальнити так:

– крапка, за якою іде дужка, що відкриває S-вираз, може бути опущена разом з відповідно їй дужкою, закриває S-вираз.

– крапка, за якою слідує атом **NIL,** також може бути опущена разом з атомом **NIL**.

Крапкове подання може бути використане у тих випадках, наприклад, коли є спроба зіслатися на перші два елементи списку **X** довільної довжини. Запишемо

(X1 X2 . Y) замість **(X1 X2 X3 ... Xk)**

В цьому записі **X1** і **X2** перші два елементи списку **X**, а **Y**-залишок списку **(X3 ... Xk)**.

Іноді зручніше при побудові пар елементів використовувати крапкове подання за допомогою операції **CONS**, а не будувати двочлени. Вибір подання – це справа смаку. Але слід відзначити, що точкові пари потребують менше пам'яті при машинній реалізації, хоча версія з двочленами може бути більш наглядною і чіткою, коли пари утворюють з загальних S-виразів, а не з атомів.

Порівняємо

((X . A) (Y . (B . A))(Z . NIL))

з виразом

((X . A) (Y B . C) (Z)),

який є скороченою формою списку у крапковому поданні. При видруку S-виразу вибирається найкоротша форма S-виразу.

Тепер під S-виразом ми будемо розуміти або атом, або крапкову пару двох S-виразів – результат операції **cons**. Тому при побудові функцій від S -виразів, на противагу спискам, маємо інший аналіз випадків.

Побудуємо функцію **розмір(S)**, яка вираховує кількість атомів в S-виразі. Маємо такий алгоритм:

випадок 1) S-атом
розмір (S) = 1
випадок 2) S не є атом
нехай розмір (car (S)) = m
і розмір (cdr (S)) = n
тоді розмір(S) = m + n

Отже, ми побудували функцію

розмір (S) = якщо атом (S) то 1 інакше
плюс (розмір (car (S), розмір (cdr (S))

Слід відзначити, що при такому означенні при застосуванні функції **розмір** до складного списку враховуватимуть атоми NIL, якими закінчується кожний список.

Маючи аналіз випадків від загальних S-виразів, ми можемо побудувати предикат **тотожність (x, y)**, який порівнює довільні S-вирази і видає значення **T**, якщо **x** і **y** є тотожними (тобто в них на одних і тих же місцях стоять однакові атоми), і значення **NIL** в противному разі. Не зупиняючись на алгоритмі побудови цієї функції, наведемо її функціональне означення:

тотожність (S) = якщо атом (x) то рівно (x, y) інакше
якщо атом (y) то рівно (x, y) інакше
якщо тотожність (car (x), car (y)) то
тотожність (cdr (x), cdr(y))
інакше NIL

Ця функція є узагальненням предиката **рівно (x, y)**, який використовується, якщо один із аргументів є атомом. Слід відзначити, що в наведеному визначенні друге використання предиката **рівно (x, y)** завжди має значення **NIL**, оскільки використовується у випадку, якщо **x** не є атомом.

5.2.2. Застосування крапкового подання виразів до побудови функції індивід

Продемонструємо необхідність використання крапкового подання для покращення ефективності на прикладі побудови функції **індивід (X)**, яка із загального S-виразу формує множину атомів, які в **X** входять лише один раз. Використовуючи звичайний аналіз можливих випадків, приходимо до алгоритму:

випадок 1) X є атом
індивід (X) = cons (S, NIL)
випадок 2) X не є атом
нехай індивід (car (X)) = A
і індивід (cdr (X)) = B
і множина (car (X)) = C
і множина (cdr (X)) = D
тоді індивід (X) = об'єднати (різниця (A, D), різниця (B, C))

Наведений алгоритм потребує деякого пояснення. Випадок 1, якщо S – атом є тривіальним. В другому випадку допускаємо, що множина індивідуумів обчислена для **CAR (X)** і **CDR (X)**. Остаточний результат не може бути простим об'єднанням **A** і **B**, оскільки в неї попадуть елементи, які є спільними для **A** і **B**, бо якщо якийсь елемент зустрівся один раз в **CAR (X)** і двічі в **CDR (X)**, то він буде в множині **A** і його не буде в множині **B**. отже, елемент, який тричі зустрівся в списку **X**, може потрапити в останній список. Фактично, для обчислення значення функції **індивід(X)**, нам треба взяти усі елементи з **A**, яких не має в списку **CDR (X)** і ті з B, яких немає в списку **CAR (X)**.

Для закінчення опису програми допускаємо існування трьох функцій: **множина (S)**, яка формує множину атомів із загального S-виразу; **об'єднати (U, V)**, яка формує множину атомів з **X**, які входять або в множину **U** або в множину **V**; **різниця (U, V)**, яка формує множину атомів з **X**, які не входять в **U**. Відзначимо, що кожен атом з **A** входить і в **C**, аналогічно, кожен із **B** входить і в **D**. Тоді множини **різниця (A, D)**, та **різниця (B, C)** не мають спільних елементів.

Тепер дамо функціональне означення **індивід(X)** і всіх підфункцій, що до неї входять :

індивід (X) ≡ якщо атом (X) то cons (X, NIL) інакше
об'єднати (різниця (індивід (car (X)), множина (cdr (X))),
різниця (індивід (cdr (X)), множина (car (X)))).
множина (X) ≡ якщо атом (X) то cons (X, NIL) інакше
об'єднати (множина (car (X)), множина (cdr (X))).
об'єднати (U, V) ≡ якщо рівно (U, NIL) то V інакше
{нехай Z = об'єднати (cdr (A), V)
якщо елемент (car (U), Z) то Z
інакше cons (car (U), Z) }.
різниця (U, V) ≡ якщо рівно (U, NIL) то NIL інакше
{нехай Z = різниця (cdr (U), V)
якщо елемент (car (U), V)) то Z
інакше cons (car (U), Z) }

Ця програма ілюструє кілька видів неефективності. Зокрема, це стосується того, як функція **множина** викликається функцією **індивід (X)**. Досліджуючи виклик **індивід (car (X))** бачимо, що слід вирахувати **множина (cdr (car (X)))** і **множина (car (car (X)))**. Аналогічно таких самих обчислень потребує виклик **множина (car (X))**. Отже одні й ті самі досить довгі обчислення виконуються двічі.

Цю неефективність можна усунути, використовуючи параметри нагромадження. Визначимо підфункцію **інд (X, a, b)**, застосовуючи параметр **a** для нагромадження атомів, які зустрічаються в **X** більше ніж один раз, а в **b** нагромаджуючи атоми, які зустрічаються в **X** лише один раз.

Аналізуючи можливі випадки, приходимо до такого алгоритму:

випадок 1) **X** - атом
підвипадок 1.1) **X** входить в **a**
інд (X) = cons (a, b)
підвипадок 1.2) **X** не входить в **a**
підвипадок 1.2.1) **X** входить в **b**
інд (X) = cons (cons (X, a), викрес (X, b))
підвипадок 1.2.2) **X** не входить в **b**
інд (X) = cons (a, cons (X, b))
випадок 2) **X** не атом
нехай (c.d)= інд (car (X), a, b)
тоді інд (X, a, b) = інд (cdr (X), c, d)

Випадок 1) є очевидним і не потребує коментаріїв. У другому випадку маємо справу з не препарованою структурою. Виклик **інд (car(X), a, b)** використовується для нагромадження однократно і багатократно повторюваних атомів в **car (s)**, які додають до списку **b** та **a**, формуючи множини **d** і **c**. А далі очевидним способом викликається **інд (cdr (X), c, d)**. Тут також використана функція **викрес (x, y)**, яка знищує елемент **X** із списку **Y**.

Отже, ми побудували таку функцію:

індивід (X) = cdr(інд (X, NIL, NIL))
інд (X, a, b) = якщо атом (X), то
якщо елемент (X, a) то cons (a, b), інакше
якщо елемент (X, b) то cons (cons (X, a), викрес (X, b))
інакше cons (a, cons (X, b)) інакше
{нехай p = інд (car (X), a, b)
інд (cdr (X), car (p), cdr (p)) }
викрес (X, Y) = якщо рівно (Y, NIL) то NIL інакше
якщо рівно (car (Y), X) то cdr (Y) інакше
cons (car (Y), викрес (X, cdr (Y))).

5.2.3. Застосування функцій для подання множин даних

Розглянемо використання множин **a** і **b** у визначенні функції **інд (s, a, b)**. Множина **a** досліджується функцією **елемент** і поповнюється використанням функції **cons**. У множині **b** аналогічно проводиться пошук та поповнення, а іноді вона і зменшується застосуванням функції **викреслити**. В кінцевому підсумку нас цікавить тільки множина **b**, а тому є певна свобода у виборі подання для множини **a**. Будемо подавати множину за допомогою функції.

Під множиною ми розуміємо список атомів без повторень. Припустимо, що **s** – список, який є множиною. Можна визначити функцію **f** з такою властивістю

f (x) = елемент (x, s)

і використати **f** для подання множини замість **s**. Очевидно, що не всі операції над множинами можна перенести на **f**. Але деякі з них можна. Так, якщо ми хочемо множину **s** поповнити елементом **x**, про який відомо, що він не

входить в **s**, то записуємо **cons (x, s)**. Маючи функцію **f**, можна дати функціональне означення, яке веде себе як функція елемент для розширеної множини

λ (y) якщо рівно (y, x) то T інакше f(y).

Цей **λ**-вираз означає функцію, яка, будучи застосованою до свого аргументу **y**, видає значення T, якщо **y** є або **x** або елемент множини **s**. Тому, останній **λ**-вираз є множиною, яка є розширенням множини, створеної функцією **f (x)**, в яку включено елемент **x**. Порожню множину можна подати як **λ(y) NIL**, оскільки ніякий атом не може бути елементом порожнього списку. Тепер ми можемо перевизначити функцію

```
індивід (x): інд (x, a, b) = якщо атом (x) то якщо
                a (x) то cons (a, b) інакше якщо
                елемент (x, b) то cons ({λ (y) якщо рівно (x, y) то T
                інакше a(y)}, викреслити (x, b) інакше
                cons (a, cons (x, b)) інакше
                {нехай p = інд (car (x), a, b)
                        інд (cdr (x), car (p), cdr (p))}
```


По суті, ми маємо лише три зміни в означенні функції. Порожня множина **NIL** заміняється функцією **λ(y) NIL**, виклик **елемент (x, a)** замінюється викликом **a (x)** і виклик **cons(x,a)** замінюється на **{λ (y) якщо рівно (x, y) то T інакше a (y)}**. В цьому випадку в парі, яка видається як результат функції **інд (x, a, b)** перший елемент є функцією. Ця можливість включати функції в структури даних і використовувати їх замість структур даних є надзвичайно потужним засобом функціонального програмування.

5.2.4 Контрольні питання та вправи

1. Поняття про крапкове подання S-виразів.
2. Сформувати правила спрощення S-виразів в крапковому їх поданні.
3. Записати всі можливі еквівалентні подання виразу (A . (B . (C . NIL)))
4. Модифікувати функцію **розмір (x)** так, щоб при підрахунку кількості елементів в списках не враховувався атом **NIL**, яким закінчується кожен вкладений список.
5. Записати функції **тотожність (x, y)**, використовуючи локальну форму **нехай**. Порівняти її ефективність з вищенаведеним означенням.
6. Побудувати функцію **індивідум (x)**, яка формує множину тільки з тих елементів, що зустрічаються в **x** лише один раз, де **x** – простий список.
7. Навести алгоритм побудови та функціональне означення функції **симрізниця (u, v)**, яка формує множину з тих елементів, які входять в **u** або **v**, але не входять в обидві множини одночасно. Записати S-вираз цієї функції.
8. Означити функцію частота(x, к), яка з простого списку формує список пар, де на першому місці стоїть елемент вхідного списку, а на другому кількість його входжень у цей список. Означення дати двояким чином: у вигляді асоціативного списку та на основі крапкового подання. Порівняти ефективність приведених означень.

5.3. Програма аналізу розмірності арифметичних виразів – як приклад структурованої побудови та відлагоджування функціональної програми

5.3.1. Вибір способу подання даних

Арифметичні формули, які описують явища реального світу, виражаються в термінах величин, які мають відповідні розмірності. Ці розмірності виражаються в фундаментальних одиницях маси (M), довжини (L) і часу (T). Тоді площа має розмірність L^2, швидкість LT^{-1}. Поки що некратною величиною одиниць розмірності цікавитися не будемо. Для прикладу, рівняння

$$V = U + at$$

може бути проаналізовано так. Величини, що входять в вище приведену формулу, мають такі розмірності

$$V = LT^{-1};\ U = LT^{-1};\ a = LT^{-2};\ t = T.$$

Як бачимо, розмірність aT є LT^{-1}, а тому розмірності величин лівої та правої частин нашої формули збігаються. Приходимо до висновку, що формула є правильною за розмірністю.

Нехай нам потрібно написати функціональну програму, яка б здійснювала аналіз розмірності. По суті, необхідно дати означення функції, яка як аргумент сприймає арифметичний вираз, а як результат видає його розмірність. Перш ніж писати таку програму, необхідно вибрати подання арифметичних виразів і подання розмірності величин, що входять у вираз. Обмежимось аналізом виразів, в які входять змінні, константи, операції +, -, *, /.

Тепер розглянемо структуру даних для подання розмірності. Взагалі розмірність має форму $M^{n1}\ L^{n2}\ T^{n3}$, де n1, n2, n3 – цілі числа, які можуть бути додатними, від'ємними або дорівнювати нулю. Тобто розмірність може бути подана триелементним списком. Так, швидкість LT^{-1} може бути подана списком (0 1 -1), сила MLT^{-2} – списком (1 1 -2). Маючи таке просте подання, розпочнемо побудову нашої функції і покажемо, що вона не залежить від способу подання даних. Для спрощення аналізу формул всі змінні перенесемо в праву частину і будемо аналізувати лише праву частину. Очевидно, необхідно задати розмірності всіх змінних, що входять в формулу і для них також необхідно вибрати структуру подання.

Очевидно, ці розмірності повинні бути завданні, як аргументи функцій аналізу, так, що необхідно передбачити структуру цих аргументів. Оскільки кожній змінній відповідає єдина розмірність, то цей факт може бути відображеній списком пар, кожна з яких являє змінну і пов'язану з нею розмірність. Такий список, як правило, називають асоціативним. Структура для асоціативного списку з K елементів, де V1...VK – їх розмірності подається у такому вигляді:

((V1 d1)(V2 d2)...(VK dK)).

Основна функція, за допомогою якої можна маніпулювати з таким списком – це функція **асоц (V, a)**, де **V** – змінна, значення якої (в нашому випадку це розмірність) необхідно видати. Якщо **V** входить в список **a**, то маємо

асоц (V, a) = якщо рівно (v, car (a)) то car (cdr (car (a)))
інакше асоц (V, cdr (a)).

Припустивши, що **V** не обов'язково входить в **a**, можна видавати деяке стандартне значення, якщо ніякого асоціативного зв'язку не буде виявлено між змінною **a** і списком **V**. Для досягнення цієї мети використаємо атом неозначеності, який для опрацювання запису позначимо **!**. Тоді функція визначення розмірності конкретної змінної перепишеться в такому вигляді

асоц (V, a) = якщо рівно (a, NIL) то ! інакше
якщо рівно (V, car (car (a))) то
car (cdr (car (a))) інакше асоц (V, cdr (a)).

5.3.2. Побудова основної функції

Тепер можна розпочати побудову основної функції аналізу, відклавши на пізніше побудову всіх підфункцій, які в неї будуть входити. Назвемо нашу функцію аналізу розмірностей формул **розм (e, a)**, де **e** – це вираз, що підлягає аналізу, **a** – асоціативний список. Для запису алгоритму побудови цієї функції нам необхідно розглянути п'ять випадків, що відповідають п'ятьом можливим типам правильних арифметичним виразів **e**. Приходимо до такого алгоритму побудови цієї функції

випадок 1) **e** - змінна **розм (e, a) = асоц (e, a)**
випадок 2) **e** має форму (ПЛЮС E1 E2)
нехай розм (E1, a) = d1
і розм (E2, a) = d2
тоді якщо d1 і d2 співпадають то розм (e, a) = d1
інакше розм (e, a) = Ф
випадок 3) **e** має форму (МІНУС E1 E2)
аналогічно випадку 2)
випадок 4) **e** має форму (МНОЖ E1 E2)
нехай розм (E1, a) = d1
і розм (E2, a) = d2
тоді розм (e, a) = множ (d1, d2)
випадок 5) **e** має форму (ДІЛ E1 E2)
нехай розм (E1, a) = d1
і розм (E2, a) = d2
тоді розм (e, a) = діл (d1, d2)

Розглянуті випадки 1-5 є аналогічні в тому відношенні, що розмірність складного виразу є функцією розмірності його компонент. Функції **множ (d1, d2)** і **діл (d1, d2)** у випадках 4) і 5) ще підлягають визначенню. Для операцій ПЛЮС та МІНУС функція **діл** щодо збігу розмірностей також ще невизначена. Назвемо її **співпад (d1, d2)** і дамо означення функції **розм (e, a)**:

```
розм (e, a) = якщо  атом (e) то  асоц (e, a) інакше
                {нехай d1 =  розм (car (cdr (e)), a)
                      і d2 =  розм (car (cdr (cdr (e))), a)
                якщо car (e) = ПЛЮС то
                якщо  співпад (d1, d2) то d1 інакше Ф
                інакше якщо car (e) = МІНУС то
                      якщо  співпад (d1, d2) то d1 інакше Ф
                 інакше якщо car (e) = МНОЖ то  множ (d1, d2)
                 інакше якщо car (e) = ДІЛ то  діл (d1, d2)
                 інакше Ф}
```

Спосіб дії цієї функції досить прямолінійний. Спочатку присвоюються розмірності самим внутрішнім підвиразам, далі – більш зовнішнім і нарешті – визначається розмірність всього виразу.

5.3.3. Побудова додаткових функцій

Для закінчення побудови цієї функції необхідно побудувати функції **співпад**, **множ**, **діл**. Вважаючи заданим попереднє подання для розмірностей, приходимо до визначень

```
співпад (d1, d2) = якщо d1 = Ф то NIL інакше
                        якщо d2 = Ф то NIL інакше
                        якщо маса (d1) = маса (d2) то
                        якщо довжина (d1) = довжина (d2) то
                              час (d1) = час (d2)
                        інакше NIL
        маса (d) = car (d)
        довжина (d) = car (cdr (d))
        час (d) = car (cdr (cdr (d)))
        множ (d1, d2) = якщо d1 = Ф то Ф інакше
```

```
                якщо d2 = Ф то Ф інакше
                трійка (маса (d1) + маса (d2),
                довжина (d1) + довжина (d2),
                час (d1) + час (d2))
діл (d1, d2) = якщо d1 = Ф то Ф інакше
                якщо d2 = Ф то Ф інакше
                трійка (маса (d1) - маса (d2),
                довжина (d1) - довжина (d2),
                час (d1) - час (d2) )
трійка (x, y, z) = cons (x, cons (y, cons (z, NIL)))
```

Ці визначення говорять самі за себе. Розглянемо, як ці функції справляються з виразом

(ПЛЮС U (МІНУС (МНОЖ A T) V))

при асоціативному списку

((U (0 1 -1)) (V (0 1 -1)) (A (0 1 -2)) (T (0 0 1)))

Функція **розм (e, a)** будує розмірності, наведені в таблиці

E	розм (e, a)
U	(0 1 -1)
A	(0 1 -2)
T	(0 0 1)
(множ AT)	(0 1 -1)
V	(0 1 -1)
(МІНУС (множ (AT) V)	(0 1 -1)
(ПЛЮС U (МІНУС (множ AT V)))	(0 1 -1)

При відлагоджені всієї програми необхідно здійснити відлагодження кожної підпрограми незалежно. Очевидно функції **маса**, **довжина**, **час**, **трійка** є тривіальними і для них таке відлагодження не є необхідне. Однак, функції **співпад**, **множ** і **діл** дещо менш тривіальні і їх незалежна відлагодження має ту перевагу, що коли перейдемо до відладки всієї програми, а саме функції **розм (e, a)**, то можна вважати роботу підфункцій

правильною. Функцію **асоц** також можна відлагодити окремо, а вже потім розпочати відлагодження головної програми, спочатку з простими даними, а вже потім з усім виразом, який підлягає аналізу.

5.3.4. Універсальний спосіб подання даних

При потребі використання даних в абсолютній конкретній формі, очевидно, необхідно використати іншу форму подання. Насамперед необхідно переконатись в погодженості всіх одиниць. Так, якщо довжина вимірюється в метрах, час – в секундах, то швидкість – в метрах за секунду, а прискорення – в метрах за секунду за секунду.

Попереднє подання нас не влаштовує, бо можемо мати більше ніж три одиниці вимірювання, наприклад, вимірюючи довжину в сантиметрах. Розширення даних до n-членного списку не є зручним, оскільки тут допускається перераховування всіх одиниць, які будуть коли-небудь використовуватися, а також цей список повинен бути достатньо коротким. Тому будемо оперувати з одиницями більш явно.

Відзначимо, що розмірність може бути записана як відношення одиниць. Так

метри за секунду – м/сек
метри за секунду за секунду – м/сек^2
кілограм на квадратний метр – кг/м^2.

Отже, розмірність можна подати як пару (двочленний список) списків, відповідно чисельник і знаменник такої розмірності:

((м) (сек))
((м) (сек сек))
((кг) (м м)).

Таке подання є доцільним, оскільки воно дає змогу відобразити всі можливі екстремальні випадки. Наприклад,

((м) NIL) – базова одиниця, метри
(NIL (сек)) – одиниця за секунду
(NIL NIL) – безрозмірна величина.

Але є деякі недоліки такого подання, зв'язані з його надлишковістю. Наприклад, розмірність ((м) (м сек)) краще записати як ((NIL (сек)), "скорочуючи" на м. Будемо говорити, що подання, в якому неможливі такі скорочення, є приведене. Зразу ж побудуємо функцію **привести (d)**, яка приводить задану розмірність **d** до її нескоротного подання.

При її розробці вважаємо, що **d** є атомом ⊥, або має форму **(t b)**, де чисельник t і знаменник b нашої розмірності є списками. Функція **привести(d)** буде брати почергово список чисельника і дивиться, чи не можна його викреслити із знаменника. Спочатку розглянемо випадок, коли **d=⊥**, а потім записуємо алгоритм всіх подальших обчислень:

привести (d) = якщо d=⊥ то - інакше прив (d)

Розглянемо випадки, з якими буде мати справу функція **прив(d)**:

випадок 1) **d** має форму **(NIL b)** - форма є приведеною

випадок 2) **d** має форму **((u.t) b)**

нехай d" = прив(d')

де d' = (t b)

підвипадок 2.1) **u** входить в знаменник **d"**

тоді результат функція **d"** з викресленим **u** із знаменника **d"**

підвипадок 2.2) **u** не входить в знаменник **d"**

тоді результат функції **d"** з добавленням **u** в чисельник **d"**.

Випадок 1) відповідає приведеній формі. У другому випадку введемо побудовану приведену форму **d"**, яка не опрацювала ще лише елемента **u**. Якщо елемент **u** входить в знаменник **d"**, то викидаємо його, а якщо не входить, то включаємо в чисельник і отримуємо результат. Отже, приходимо до такого функціонального означення:

```
прив (d) = якщо car (d) = NIL то d інакше
          {нехай d" = прив (пара (cdr (чис (d))) знам (d))
           якщо елемент (car (чис (d)), знам (d")) то
          пара (чис (d"), викр (car (чис (d)), (знам (d")) інакше
          інакше пара (cons (car (чис (d)), чис (d')), знам (d"))}
чис (d) = car (d)
знам (d) = car (cdr (d))
пара (t,b) = cons (t, cons (b, NIL)
```

Приводити можна будь-яку розмірність незважаючи на те, чи вона приведена. Організуємо нашу програму так, щоб як тільки була отримана нова розмірність, то вона відразу приводилася.

Згадаємо, що функція **розмір (e, a)** для обробки розмірностей викликає функції **співпад(d1,d2)**, **множ(d1,d2)**, **діл(d1,d1)**. Конкретне подання розмірностей стосується саме цих функцій. Тому щоб працювати з новим поданням даних, необхідно перепрограмувати саме ці функції. Найпростіша з них – функція **множ(d1,d2)**. Ми просто з'єднуємо два чисельники та два знаменники щоб отримати відповідно новий чисельник і новий знаменник. Оскільки нова розмірність може бути неприведеною, то її слід привести. Нарешті, необхідно врахувати випадок, коли один із аргументів є невизначеним. Нова функція **множ(d1,d2)** має вигляд:

множ(d1,d2)=якщо d1=⊥ то ⊥ інакше
якщо d2=⊥ то ⊥ інакше
привести (пара(з'єднати(чис(d1),
чис(d2)), з'єднати(знам(d1),знам(d2))))

Функція ділення двох розмірностей отримується аналогічно, за винятком того, що чисельник результату отримується із з'єднання чисельника **d1** із знаменником **d2**, а знаменник результату – це з'єднання чисельника **d2** і знаменника **d1**:

діл(d1,d2)= якщо d1=⊥ то ⊥ інакше
якщо d2=⊥ то ⊥ інакше
привести (пара(з'єднати(чис(d1))
знам(d2)), з'єднати(чис(d2),знам(d1))))

При перепрограмуванні функції **співпад(d1,d2)** ми не можемо просто порівняти ці розмірності, оскільки порядок слідування одиниць нас не цікавить. Так, такі розмірності є еквівалентними:

((кг)(м м сек))
((кг)(м сек м))
((кг)(сек м м))

Тут необхідно порівняти списки атомів і подивитися, чи вони не збігаються безвідносно до їх впорядкування. Можна брати елементи першого списку і викреслювати їх з другого і подивитися, чи ми не прийдемо одночасно до порожнього списку. Це нагадує "скорочення" і приводить до думки використати для цієї мети функцію **привести(d)**. Дві розмірності рівні, якщо їх відношення безрозмірне. Тобто, якщо вирахувати **діл(d1,d2)** і результат буде **(NIL, NIL)** то **d1** збігається з **d2**. Звичайно, необхідно потурбуватися про випадок, коли одна із розмірностей є невизначеною. Маємо таке означення функції **співпад(d1,d2)**:

співпад(d1,d2)= якщо d1=⊥ то ⊥ інакше
якщо d2=⊥ то ⊥ інакше
{ нехай d1=діл(d1,d2)
якщо чис(d)=NIL то
знам(d)=NIL інакше NIL }

Для відлагодження програми доцільно побудувати схему викликів всіх функцій:

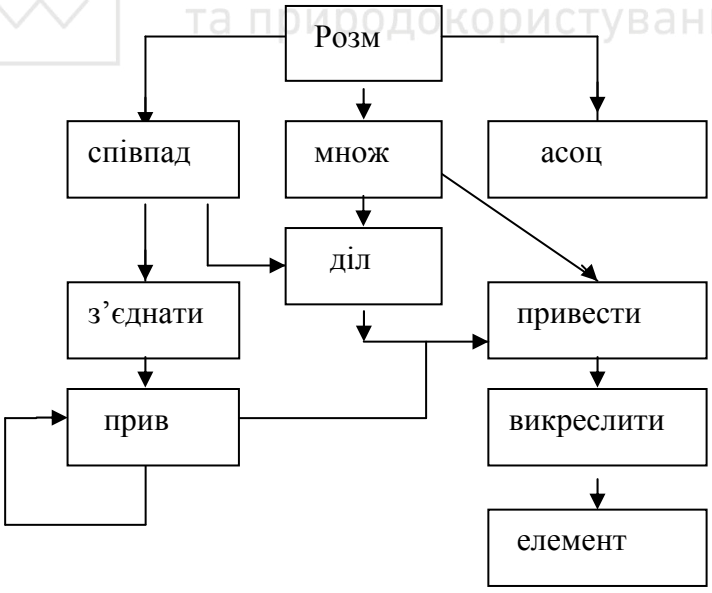


Рис. 2.1. Схема викликів всіх функцій

Існують два способи відлагодження поданої програми. Можна все відлагодження виконувати методом знизу вверх або можна включати елементи аналізу зверху вниз. При відлагоджені знизу вверх спочатку досліджуються ті функції, які не викликають інших, потім ті функції, які викликають лише відлагоджені функції і так далі. Текстові дані слід вибирати так, що якщо виникне невдача при відлагоджені, то слід сподіватися, що помилки є в функції вищого рівня, яка є головною для цього тексту. Головну функцію слід відлагоджувати лише тоді, коли відлагоджені всі підфункції, які вона викликає. При відлагоджені, де тільки це можливо, підфункції повинні оперувати лише з тими аргументами, які їм дає функція, що викликає. Дещо заплутує справу наявність рекурсії. Але якщо продумати структуру вхідних даних, то можна забезпечити виклик функції 0, 1 і більше разів, поступово її освоюючи. При відповідному підборі даних можна забезпечити і відлагодження взаємно рекурсивних функцій. Наприклад, якщо **f** викликає **g** і **g** викликає **f** так,

f(...)= якщо ... то ... g(...) ... інакше ...
g(...)= якщо ... то ... f(...) ... інакше ...,

то ми повинні відлагодити **f** і **g** разом. Але, очевидно, можна вибрати дані так, щоб f не викликала **g**, потім викликала її 1 раз, а **g** в свою чергу не викликала **f** і так далі.

У розглядуваному прикладі функції **з'єднати**, **асоц**, **елемент** і **викреслити** слід відлагодити окремо. Далі доцільно відлагодити **сумісно прив** і **привести**, які порівняно є прості. Далі при відлагоджені програми по методу знизу вверх слід відлагоджувати функції **множ**, **діл**, а потім **співпад**. Нарешті, можна відлагоджувати функцію **розм** шляхом послідовної перевірки її віток.

Побудова програми аналізу розмірностей засвідчує, що наведений спосіб подання даних дозволяє побудувати досить універсальну програму. Від вибору структури програми залежить значною мірою спосіб її відлагоджування та локалізації можливих помилок.

5.3.5. Контрольні питання та вправи

1. Дати означення асоціативного списку.
2. В чому суть структурного відлагодження програми?
3. Як реалізувати процес відлагодження взаємно рекурсивних функцій?

4. Побудувати схему викликів підфункцій при поданні розмірностей у вигляді триелементного списку і описати процес відлагодження зверху вниз.

5. Для рівняння F=m · a сформувати асоціативний список та задати аргументи та функції **розм** у виді S-виразів.

6. Перепрограмувати функцію **привести(d)** при поданні розмірності як пари символів, беручи кожен атом із знаменника і викидаючи його з чисельника.

7. Навести схему всіх викликів підфункцій при побудові функції **розм(e, a)** при поданні розмірності двоелементним списком, де перший елемент список імен змінних, що входять в чисельник розмірності, а другий – список імен змінних знаменника

8. Описати процедуру відлагодження взаєморекурсивних функцій на прикладі функцій **парні(x)** і **непарні(x)**. Результати функцій – це, відповідно, списки парних і непарних елементів списку **x**.

5.4. Конкретна та абстрактна форма подання функціональних програм та способи зв'язування змінних

5.4.1. Універсальний спосіб подання даних

Коли створений алгоритм побудови функціональної програми в термінах основних операцій Ліспу – **car, cdr, cons, атом** і **рівно**, які введені в розділі 1, то його можна записати у вигляді визначення деякої функції. Відзначимо, що будь-який алгоритм можна записати в функціональних позначеннях. Таку форму подання програми будемо називати абстрактною.

Для спрощення інтерпретації функціональних програм, для їх представлення в машині вибрана форма S-виразів. Подання програми у вигляді набору S-виразів будемо називати конкретною формою подання. Такий спосіб подання дещо відрізняється від реального Ліспу, але суттєво спрощує розгляд процесу подання та інтерпретації функціональних програм. Надалі будемо називати його інструментальним Ліспом або просто Ліспом.

Посилаючись на програму, написану на Ліспі, будемо мати на увазі її подання у вигляді S-виразу. Так, наступний запис є строгим функціональним означенням в абстрактній формі відомої нам функції **з'єднати (x,y):**

{ з'єднати
дерек з'єднати=L(x,y) якщо x=NIL то y інакше
cons(car(x), з'єднати(cdr(x),y))}

Записавши це на інструментальному Ліспі, тобто у вигляді S-виразу, отримаємо форму

(НЕХАЙРЕК З'ЄДНАТИ (З'ЄДНАТИ ЛЯМБДА(X,Y)
(ЯКЩО (РІВНО X NIL) Y (CONS (CAR X)
(З'ЄДНАТИ (CAR X) Y)))))

яка вже є конкретною формою подання функціональної програми.

Покажемо, що будь-яка програма, записана на строго функціональній мові, може бути подана у вигляді S-виразу. Для цього визначимо ті вирази, які будемо називати правильними. В інструментальному Ліспі правильними є ті підвирази програми, з якими зв'язуються певні значення. Прикладом правильних виразів можуть бути такі

y
y+10
cons(x,y)
якщо x=0 то 1 інакше y+1

Кожен з виразів є довершеним в тому розумінні, що якщо відомі значення змінних, що в нього входять, то можна вирахувати значення всього виразу. Наприклад, такі вирази не є правильними, оскільки вони недовершені:

y+
cons(x
якщо x=0 то 1.

Навіть, якщо відомі значення всіх змінних, що входять в ці вирази, то з ними неможливо зв'язати певне значення. Для прикладу, яке значення останнього виразу, коли $x \neq 0$?

Всі функціональні програми, які ми розглядали до цих пір, побудовані на вкладених один в одного правильних виразах. Тому функційна програма є єдиним правильним виразом, який можна розчленувати на головну функцію, що має певне значення і її аргументи, які своєю чергою є правильними виразами з конкретними значеннями. Посилаючись на правильний вираз,

можна мати його абстрактну або конкретну форму, оскільки S-вирази наслідують властивість правильності.

Розглянемо основний набір правильних виразів, які використовують при побудові функцій та приведемо їх як конкретну так і абстрактну форми.

Найбільш елементарними правильними виразами є змінні та константи. Змінні подають символьними атомами: a, в, c; відповідно їх конкретна форма у вигляді S-виразу: A, B, C. Для розрізнення константи від змінної використовується ключове слово **КОД**. Так константи 127, NIL у вигляді S-виразів будуть **(КОД 127)**, **(КОД NIL)**.

Будь-який правильний вираз в конкретній формі зображається списком, в якому на першому місці стоїть атом, що сприймається як ключове слово. З викликом функції поступаємо аналогічно. Виклик $\mathbf{f(e_1, e_2, \ldots, e_k)}$ перетворюється в конкретну форму (**F E1 E2 ... EK),** де на першому місці ім'я функції, а далі її аргументи.

Ні умовна форма, ні **λ-**вирази не викликають ніяких ускладнень при переході до конкретної форми. Абстрактній формі

$\mathbf{якщо\ e_1\ то\ e_2\ інакше\ e_3}$

відповідає конкретна S-форма виду

(ЯКЩО E1 E2 E3) .

Таким чином, маємо чотирьохелементний список з ключовим словом **ЯКЩО.**

λ-вираз

$\mathbf{\lambda\ (x_1, x_2, \ldots, x_k)\ e}$

перетворюється до вигляду

(LAMBDA(X1 ... XK)E).

тобто трьохелементному списку з ключовим словом **LAMBDA**, списком параметрів і виразом, який використовується для вирахування функції.

Так, функція

λ(x,y) cons(x, cons(y, NIL))

перетворюється в форму

(LAMBDA(X Y) (CONS X (CONS Y (КОД NIL))))

Перейдемо до розгляду представлення конструкцій **нехай** і **де** та їх рекурсивних аналогів. Відзначимо, що в конкретній формі допускається лише один спосіб локальних означень, який є гібридною формою конструкцій

нехай і **де**. Тобто допускається використання ключових слів **НЕХАЙ** і **НЕХАЙРЕК**, але визначальний вираз ставиться перед локальними визначеннями, як в звичайній формі **де**. Так кожен з абстрактних правильних виразів

{ e		{НЕХАЙ
де $x_1=e_1$		X1=e1
і $x_2=e_2$	або	і X2=e2
.....		.....
і $x_k=e_k$ }		і XK=eK
		e}

перетворюється в єдину конкретну форму

(НЕХАЙ E (X1 E1) (X2 E2) ... (XK EK))

Аналогічно дві абстрактні рекурсивні форми

{ e		{НЕХАЙРЕК
дерек		X1=e1
$x_1=e_1$	або	і X2=e2
і $x_2=e_2$		.....
.....		і XK=eK
і $x_k=e_k$ }		e}

забезпечуються єдиною формою у вигляді S-виразу

(НЕХАЙРЕК E (X1 E1) (X2 E2) ... (XK EK))

Таким чином, блок з k визначеннями зображається у вигляді списку з k+2 елементів, де на першому місці ключове слово, другий елемент – визначений вираз, а далі йдуть локальні визначення. Кожен локальний вираз зображається крапковою парою. Оскільки здебільшого значення локальної змінної є виразом, то ця крапка разом з відповідними дужками опускаються, згідно правил спрощення S-виразів.

Перерахуємо всі S-вирази, які є правильними для програм, написаних на інструментальному Ліспі.

X	змінна
(КОД E)	константа
(ПЛЮС E1 E2) (МІНУС E1 E2) (МНО E1 E2) (ДІЛ E1 E2) (ЗАЛ E1 E2)	арифметичні операції
(РІВНО E1 E2) (АТОМ E) (CAR E) (CDR E) (CONS E1 E2)	примітивні функції Ліспу
(ЯКЩО E1 E2 E3)	умовна форма
(ЛЯМБДА(X1 X2 ... XN) E)	L-вираз
(E E1 ... EN)	виклик функції
(НЕХАЙ E (X1 E1) ... (XN EN))	рекурсивний блок

Тут X1 X2 ... XN – атоми; E E1 ... EN – правильні вирази, включені один в одного.

Якраз при інтерпретації функціональних програм виділяються і враховуються всі правильні вирази, на які може бути розчленована основна функція. Дійсно, тільки правильний вираз є довершеною формою і для нього можна вирахувати значення, а тому структури інтерпретатора продиктовані множиною правильних виразів, які тут перераховані.

5.4.2. Проблеми зв'язування змінних у контексті

Очевидно, щоб отримати значення правильного виразу, необхідно всі змінні. які в нього входять, зв’язати з певними значеннями. Для пояснення правил зв’язування в функціональній мові введемо поняття контексту, під яким будемо розуміти відповідність або зв’язок між змінними та їх значеннями, які є S-виразами. Будемо записувати ці зв’язки у формі

змінна → значення

Якщо задано контекст

$X \rightarrow (P, Q)$

$Y \rightarrow (A, B)$

$Z \rightarrow 5,$

то нижченаведені вирази приймають значення:

Вираз	Значення
cdr(X)	Q
cons(X,Y)	((P,Q) A B)
Z-7	-2

Блоки **нехай** і **де** забезпечують утворення нових зв'язків і розширюють контекст. У випадку коли локальні змінні мають ті ж імена, що й в контексті, то старі зв'язки витісняються новими.

Обговорення проблеми зв'язування в контексті функцій почнемо з таких, що не містять вільних (глобальних змінних). Функції означують за допомогою **λ**-виразів, а функціональне значення зв'язується зі змінними за допомогою вище розглядуваних блоків. Так у випадку виразу

{ нехай f = λ(x,y) 2 *x+y f(2,3) + f(3,2) }

функція f вираховується двічі: один раз в контексті x→2, y→2; а другий – x→3, y→2. Звідси значенням всього виразу буде 15. Таким чином, механізм блоків дає ім'я функції, з яким зв'язується функціональне значення. При відсутності глобальних змінних **λ**-вираз є цим значенням. За наявності глобальних змінних в якості функціонального значення використовується модифікований **λ**-вираз, в якому входження глобальних змінних замінено їх значеннями. Так, якщо вираз

{ нехай f = λ(y) 2xy+z
{нехай z=5
f(3)}}

вираховується в контексті z→1, то спочатку f зв'язується з функцією **λ (x) 2 xx+1**, далі Z зв'язується з 5 утворюється контекст

f→λ(y) 2xy+1
z→5

в якому обчислена f(2) дає 5. Такий процес зв'язування, як вже відзначалося раніше, в якому глобальним змінним надаються значення не в момент виклику функції, а в момент вирахування **λ**-виразу носить назву статичного.

Еквівалентний спосіб опису функціональних значень носить назву замикання, під яким будемо розуміти λ-вираз і контекст, який визначає значення глобальних змінних.

Розглянемо загальну форму виклику функції, коли вона задається λ-виразом з конкретним значенням аргументів:

(LAMBDA (X1 ... XN) E (E1 ... EN))

По суті тут функція **λ(X1, ..., XN)e** безпосередньо застосовується до аргументів e1, ..., eN і видається як значення всього виразу. В такій формі вираз має таке ж значення як подання

(НЕХАЙ E (X1 E1) ... (XN EN))

Тобто, можна зробити висновок, що блок **НЕХАЙ** не є обов'язковим елементом мови. Однак розділення формальних і фактичних аргументів в λ-виразі робить локальну форму **НЕХАЙ** більш легкою і зрозумілою на практиці, що й зумовлює її частіше використання на практиці.

5.4.3. Правила зв'язування у рекурсивних блоках

Перейдемо до розгляду правил зв'язування в більш складних рекурсивних блоках, коли всі виклики є взаємно рекурсивними.

{НЕХАЙРЕК

$X_1=e_1$

і $X_2=e_2$

.....

і $X_K=e_K$

e}

Складність тут в тому, що змінні X_1 ... X_N є доступними не лише в виразі e, але і e_1, ..., e_N і на них необхідно посилатися, коли вони ще не є довершеними. Будемо розглядати лише такі блоки, в яких e_1, ..., e_K є λ-виразами. В такому випадку значенням e_i є замикання $[e_i, \alpha]$, де α - контекст, в якому кожна з x_1 ... $x_к$ зв'язані з e_1, ..., $e_к$, тобто

$\alpha \equiv \{ x_1 \rightarrow [e_1, \alpha]$

.....

$x_к \rightarrow [e_к, \alpha] \}$.

З таким рекурсивним визначенням легко поводитися досліднику і набагато складніше комп'ютеру. Адже тут просто відзначається факт, що

контекст має ім'я α, але не вказується значення, яке він набуде. Коли обчислюється значення e (чи будь-яке інше e_i) і зустрілася змінна x_i, то вона набуває значення замикання $[e_i, \alpha]$. А тому послідовні входження x_1 ... $x_к$ в e приймуть значення в контексті α, що й хотілося зробити. Розглянемо приклад:

{ нехайрек f = λ(x) якщо атом (x) то x інакше f(cdr(x))
f(z)}.

При цьому заданий контекст

$$\mathbf{z \rightarrow (A\ B.C)}$$

Для обчислення значення f(z) беремо контекст

$$\mathbf{\alpha \equiv \{ f \rightarrow \lambda(x)\ якщо\ атом\ (x)\ то\ x\ інакше\ f(cdr(x)),\ \alpha\}}$$
$$\mathbf{z \rightarrow (A\ B.C)\ \}}$$

Виклик f(z) змушує нас ввійти в функцію з контекстом

$$\mathbf{x \rightarrow (A\ B.C)\ \}}$$
$$\mathbf{f \rightarrow [\lambda(x)\ якщо\ атом\ (x)\ то\ x\ інакше\ f(cdr(x)),\ \alpha]}$$
$$\mathbf{z \rightarrow (A\ B\ C)}$$

Оскільки x не є атом, то наступний рекурсивний виклик **f(cdr(x))** поповнює контекст α, а значення функції **f** залишиться без зміни. Таким чином, ми викликаємо функцію з контекстом

$$\mathbf{z \rightarrow (X\ B.C)\ \}}$$
$$\mathbf{f \rightarrow [\lambda(x)\ якщо\ атом\ (x)\ то\ x\ інакше\ f(cdr(x)),\ \alpha]}$$
$$\mathbf{z \rightarrow (A\ B.C)}$$

Отже, ми переконалися в тому, що маємо те ж визначення функції і рекурсія працює потрібним чином, оскільки ще через два виклики X зведеться до **NIL**, який і буде виданий як результат функції.

При побудові інтерпретатора Ліспу наведена конкретна реалізація цих правил зв'язування.

5.4.4. Контрольні питання та вправи

1. Дати визначення конкретної та абстрактної форми подання функціональних програм.
2. Що таке правильний функціональний вираз?
3. Побудувати функцію **ПАЛІНДРОМ (X, У)**, яка істинна у випадку,якщо елементи списку **У** подано у зворотньому порядку до списку X.
4. Подати у вигляді S-виразу програму обертання складеного списку елементів на довільному рівні ієрархії.
5. Суть аналогії між **λ-**виразом та простим блоком.
6. В чому суть замикання **λ-**виразів?
7. Перерахувати правильні вирази інструментального Ліспу.
8. Вирахувати значення виразу

   {НЕХАЙ N=1 {НЕХАЙ f=λ(x) x + N {НЕХАЙ N=2 f(10) }}}.
9. Особливості процесу зв'язування змінних в рекурсивних блоках.
10. В чому суть статичного зв'язування змінних ?
11. Привести конкретну та абстрактну форму подання функції **тотожність (x, y)**, яка відає значення **T**, якщо S-вирази x і y співпадають поелементно і **NIL** в іншому випадку.

5.5. Побудова інтерпретатора функціональної мови

5.5.1. Варіант реалізації асоціативного списку

Особливою рисою мов функціонального програмування є та легкість, з якою він використовується для запису свого власного інтерпретатора. По суті справи, це основний напрямок використання функціональних мов.

В цьому параграфі продемонструємо основну техніку інтерпретації вихідних програм на інструментальному Ліспі. Випадки, які будуть тут проаналізовані, повторюють перелік правильних виразів, які розглядалися в п. 2.4. Якщо **e** відповідає правильному виразу, то необхідно визначити функцію **вирахувати**, яка видає значення правильного виразу. Вираз **e** може містити і вільні (глобальні змінні), тому його значення визначене лише в тому випадку, якщо заданий контекст, згідно якого здійснюється зв'язування кожної змінної з **e** з певним значенням. Визначення контексту та процедура зв'язування змінних розглянута в попередньому параграфі. Одна з можливостей задання такого контексту – це створення асоціативного списку, один із варіантів якого розглядається в п. 5.3.1. По суті, ця реалізація

запропонована Дж. Маккарті в його оригінальному інтерпретаторі. Ми ж розглянемо ефективнішу версію реалізації асоціативного списку. З цією метою введемо в розгляд список імен, в якому перераховані всі змінні, які входять в програму. Список імен має структуру списку із списків атомів, де кожен атом відповідає змінній. Так вираз

((X Y Z) (F) (A1 X1)

є список імен. В програмі на інструментальному Ліспі кожен з підвиразів входить в деякий контекст, де діють деякі зі змінних, які доступні для даного підвиразу. Ці змінні доцільно впорядкувати так, щоб локальні змінні були ззовні, а більш глобальні – всередині. Для інтерпретації програм необхідно задати ще й список значень, конгруентний списку імен. Так, наведеному вище списку імен відповідає список значень

(1 (A B) 5) ((A C D)) ((CONS 5 7) -1 2))

Ці два списки забезпечують контекст

X → 1
Y → (A B)
Z → 5
F → (A C D)
A1 → (CONS 5 7)
X1 → -1 2.

Для доступу до списку значень необхідно визначити функцію, назвемо її **асоц(x, n, v),** яка за іменем **x** із списку імен **n** видає відповідне їй значення із списку значень **v**. Її функціональне визначення є наступне:

асоц(x, n, v) ≡ якщо елемент (x, car(n)) то місце ((x, car(n),
car(v))
інакше асоц(x,cdr(n),cdr(v))
місце (x,l,m) ≡ якщо рівно (x, car(l)) то car(m)
інакше місце(x,cdr(l),cdr(n))
елемент (e, S) ≡ якщо рівно (S, NIL) то NIL
інакше елемент(x,cdr(S))

Алгоритм роботи функції наступний: почергово перевіряється наявність елементу **x** в кожному з підсписків списку **n**. Якщо такий підсписок виявлений, то функція **місце (x, l, m)** визначає його позицію в підсписку l і видає його відповідне значення із списку **m**. Відзначимо, що **асоц(x, n, v)** передбачає наявність елемента **x** в списку **n**, бо в противному випадку її значення є невизначене. Щоб забезпечити простоту і наочність програми інтерпретації, в подібних випадках не будемо ускладнювати підфункцій зайвою деталізацією.

5.5.2. Побудова функції вирахувати (e, n , v)

Перейдемо до побудови функції **вирахувати (e,n,v)**, яка як аргументи має правильний вираз **e**, список імен змінних **n** і конгруентний йому список значень **v**. Для побудови функції розглянемо всі можливі випадки правильних виразів. Для спрощення побудови інтерпретатора кожен з випадків запишемо у вигляді рівняння

вирахувати [(car(x),((x)),(((A.B))))]=A

щоб вказати значення виразу **(CAR E)** в контексті, що зв'язує змінну **X** зі значенням **(A.B)**.

Правильні вирази будемо позначати буквами **e, e1, e2, ... ek**; як змінні будемо використовувати атоми **x, x1, x2, ... xk**; списки імен - **n, n1, n2, ... nk**; **v, v1, v2, ... vk** – списки значень; інші букви будемо використовувати для позначення довільних S-виразів. Почергово розглянемо всі правильні вирази, наведені в табл. п.26. Значенням змінної **x** є те значення, яке можна вибрати з асоціативного списку, тобто

вирахувати [x,n,v]=асоц(x,n,v)

Значення правильного виразу **(КОД S)** є S незалежно від його значення, тобто

якщо вирахувати [e,n,v]=a
то вирахувати [car E,n,v]=car(a)

Аналогічно визначається правило вирахування **cdr(e)**. Для виразу **cons(e1,e2)** можна записати таке правило

якщо вирахувати [e1,n,v]=a1
і вирахувати [e2,n,v]=a2
то вирахувати [(CONS E1 E2), n, v]=cons(a1,a2)

Правила для визначення вирахування функцій **атом, рівно** є цілком аналогічні до вищенаведених.

Умовна форма **якщо e1 то e2 інакше e3** дещо складніша для вирахування і потребує пояснень. Оскільки обчислення **e2** і **e3** одночасно не потрібно, то достатньо вирахувати спочатку **e1**, а потім прийняти рішення про обчислення **e2** або **e3**. Отже, маємо форму

якщо вирахувати [e1,n,v]=a1
то вирахувати [(ЯКЩО E1 E2 E3), n, v]=якщо a1
то вирахувати [e2, n, v] інакше вирахувати (e3, n, v)]

Тут вираховується один з виразів **e2** або **e3**, що стає очевидним, якщо переписати останнє визначення у вигляді

якщо вирахувати [e1,n,v]=a1
то вирахувати [(ЯКЩО E1 E2 E3), n, v]=
вирахувати [якщо a1 то e2 інакше e3, n, v)]

Для інтерпретації λ-виразу скористаємося складним об'єктом, що є комбінацією λ-виразу і контексту, утворюючи структуру, яка називається замиканням. Замикання містить не тільки тіло λ-виразу, але й імена зв'язаних змінних і контекст, в якому вираховується тіло функції. Цю інформацію запишемо у вигляді:

вирахувати [LAMBDA (Y E), n, v]=
= cons(cons(y,e), cons(n,v))

Необхідність збереження всієї цієї інформації стане очевидним з подальшого розгляду.

При виклику функції **e(e1, ...,ek)** її значення є замикання, а значення **e1,...,ek** її значення є значення аргументів функції. Для вирахування аргументів введемо в розгляд ще одну функцію

вирсписок (l, n, v)≡якщо рівно (l, NIL) то NIL
інакше cons(вирахувати(car(l), n, v), вирсписок(cdr(l), n, v))

і приймемо щодо цієї функції те саме допущення стосовно квадратних дужок. Тоді можна дати визначення виклику функції у такій формі:

якщо вирахувати [e', n, v]=((y·ez)·(n'·v)
і вирсписок [(E1 ... EW), n, v]=z
тоді вирахувати [(E E1 ... EK), n, v]=
вирахувати [ez, (y·n') (z·v')]

Отже, спочатку обчислюється функціональне значення виразу **e** у вигляді замикання, і значення аргументів функції у вигляді списку **z**. Маючи ці значення, можна записати значення повного виклику функції **(E E1 ... EK)** в контексті зі списком імен **(y·n')** і списком значень **(yz·v)**.

Формули для вирахування простого блока **НЕХАЙ** дуже подібна на виклик функції і має форму:

якщо вирсписок [(e1 ... ek), n, v]=z
то вирахувати [(НЕХАЙ E(X1·E1) ... (XK·EK)), n, v]=
вирахувати [e, ((x1 ... xk)).n), (z·v)]

Це правило встановлює еквівалентність співвідношення

(НЕХАЙ E (X1 E1) ... (XK EK))≡((ЛЯМБДА (X1 ... XK) E) E1 ... EK)

в тому розумінні, що ліва і права частини набувають однакових значень.

5.5.3. Інтерпретація рекурсивного блока

Перейдемо до вирахування найскладнішої форми правильного виразу, а саме рекурсивної локальної форми

(НЕХАЙ РЕК E (X1 E1) ... (XK EK))

Труднощі, які тут виникають, пов'язані з тим, що вирази **e1 ... ek** обчислюються в оточенні, де є доступ до змінних **x1 ... xk**, які ще не обчислені.

Для адекватної інтерпретації рекурсивного блока введемо поняття псевдофункції **завершення(x, y)**, результат якої зводиться до видачі списку **x**, в якому **car(x)** замінюється на **y**. Формально цю функцію можна замінити викликом **cons(y, cdr(x))**. Але в момент виклику **завершення(x, y)** величина **car(x)** є спеціальним значенням **ω**, на яке посилаються як на незавершене, що замінюється значенням **y**. Якщо виклик функції **завершення** відбувається в момент, коли **y** ще не обчислене, то значення функції **завершення** є невизначеним.

Функція **завершення** має одну цікаву властивість, яка дозволяє будувати циклічні структури. Нехай заданий контекст

{ нехай в=cons(ω,β)
завершення(в,в) }

Значенням цього виразу, очевидно, є S-вираз **cdr** якого є **β**. Якщо взяти **car** цього виразу, то це буде S-вираз **в**. Таким чином, побудована структура **α** значенням якої є **(α.β)**. Отже, якщо **x=(α.β)**, то

cdr(car(car(car ... (car(α) ...))) = β.

Фактично, скільки би раз ми не брали **car** від **α**, ми завжди отримаємо **α**. Якщо записати вираз

{ нехай в=cons(ω, NIL)
завершення(в, з'єднати((А В С Д), в))},

то він генерує структуру, яка є нескінченним S-виразом

((А В С Д) (А В С Д) (А В С Д) ...)

Можливість будувати циклічні структури дає змогу здійснити інтерпретацію рекурсивного блока. Вирахування визначальних виразів **e1 ... ek** обчислюються зі списком значень, де значення **X1 ... XK** ще не завершені. Використовуючи псевдофункцію **завершення**, накладемо на використання

рекурсивної форми таке обмеження, при якому, обчислюючи значення **e1 ... ek** не вимагається негайного доступу до змінних **x1 ... xk**. Приходимо до визначення

якщо v1=(ω·v) і n1=((x1 ... xk)·n)
і вирсписок[E1 ... EK), n'1, v1]=z
тоді вирахувати[(НЕХАЙРЕК E (X1 E1) ... (XK EK), n, v]=
вирахувати[E, n1, завершення(v1, z)]

У цьому визначенні введений фіктивний список значень **v1**, оскільки значення **ω** ще не завершене. Список **v1** є конгруентним списку значень **n1**, а незавершене значення **ω** в підсумку буде замінено на обчислення значення виразів E1 ... EK, які поміщені в список **z**.

Визначення блоку **НЕХАЙРЕК** можна здійснити і без введення псевдофункції **завершення**, однак така структура є значно складніша для розуміння і вимагає розгляду обчислень з затримкою.

Фактичний розгляд всіх можливих правильних виразів та процесу їх вирахування є завершений і можна дати функціональне визначення результату обчислення всіх правильних виразів S-Ліспу при заданому списку імен змінних, що входять у вираз та конгруентному йому списку імен. Нам ще будуть потрібні дві функції, які розщеплюють список змінних і значень виду

((X1·E1) (X2·E2) ... (XK·EK))

на два списки: список змінних **(X1 ... XK)** і список значень **(E1 ... EK)**. Назвемо ці функції **змінні(d)** і **знач(d)** та відразу дамо їх функціональне визначення, оскільки їх побудова не викликає ніяких труднощів:

змінні(d)≡якщо рівно(d, NIL) то NIL інакше
cons(car(car(d), змінні(cdr(d)))
знач(d)≡якщо рівно(d, NIL) то NIL інакше
cons(car(cdr(d), знак(cdr(d))
.

При записі програми **вирахувати(e, n, v)** для позначення фіктивного (незавершеного значення) будемо користуватися змінною **W**. Приходимо до такого функціонального визначення:

```
вирахувати(n, e, v)≡якщо атом(e) то асоц(e, n, v)
                    інакше якщо рівно(car(e), КОД) то car(cdr(e))
                  інакше якщо рівно(car(e), CAR) то
                      car(вирахувати(car(cdr(e)), n, v)
                     інакше якщо рівно(car(e), CDR) то
                                cdr(вирахувати(car(cdr(e)), n, v))
                    інакше якщо рівно(car(e), CONS) то
                                 cons(вирахувати(car(cdr(e)), n, v))
                    інакше якщо рівно(car(e), АТОМ) то
                                атом(вирахувати(car(cdr(e)), n, v))
                    інакше якщо рівно(car(e),РІВНО) то
                                      рівно(вирахувати(car(cdr(e)), n,
                                                                  v),
                                вирахувати(car(cdr(cdr(e))), n, v))
                    інакше якщо рівно(car(e), ЯКЩО) то
                                {(вирахувати (якщо
                          вирахувати(e1, n, v) то e2 інакше e3, n, v)
                                       де e2=car(cdr(cdr(e)))
                                        і  e3=car(cdr(cdr(cdr(e))))
                                        і  e1=car(cdr(e))}
                    інакше якщо рівно(car(e), ЛЯМБДА) то
                    cons(cons(car(cdr(e)), car(cdr(cdr(e)))), cons(n, v))
                    інакше якщо рівно(car(e),НЕХАЙ) то
                    {(вирахувати(car(cdr(e)), cons(y, n), cons(z, v))
                            де y=змінні(cdr(cdr(e)))
                              і z=вирсписок(знач(cdr(cdr(e))), n, v)}
                    інакше якщо рівно(car(e),НЕХАЙРЕК) то
                     {{(вирахувати(car(cdr(e)), cons(y, n),
                                                  завершення(v1, z))
                    де z=вирсписок(знач(cdr(cdr(e))),cons(y, n), v1)}
                            {де y=змінні(cdr(cdr(e)))
                              і v1=cons(w, v)} інакше
                {вирахувати (cdr(car(c)), cons(car(car(c),car(cdr((c))),
                            cons(z, cdr(cdr(c))))
                            де c=вирахувати(car(e), n, v)
                            і   z=вирсписок(cdr(e), n, v)}
```

Наведені визначення є точною копією раніше прийнятих рішень у вигляді формул з квадратними дужками. Зміст такого функціонального запису в тому, що стала ясною форма інтерпретації правильних виразів на комп'ютері.

5.5.4. Використання функції застосувати (f ,x)

Оскільки при інтерпретації правильності того чи іншого виразу слід не вираховувати, а лише перевірити правильність синтаксичного запису, то визначимо функцію **застосувати (f, x)** так

застосувати (f, x)≡{вирахувати(cdr(cur(c)), cons(car(car(c)),
cur(cdr(c))), cons(x,cdr(cdr(c))))
де c=вирахувати(f, NIL, NIL)

В цьому визначенні вираховується **f** і утворюється замикання **c=((y·e)(n·v),** а далі обчислюється значення виразу **вирахувати [e, (y·n), (x·v)]**. Оскільки список змінних і список значень є порожніми списками, то нічого вираховуватися не буде, а лише буде перевірено синтаксис функції **f**, яка може бути довільним виразом із наведених у визначенні функції **визначити(e, n, v)** або їх довільною комбінацією. Саме цим цілям і служить інтерпретатор будь-якої мови програмування.

Побудований інтерпретатор для базових функцій **CAR, CDR, CONS, ATOM, РІВНО** надає їм ті значення, які ми вкладали в функції **car, cdr, cons, атом, рівно**. Тому не має змісту міняти ці функції. Інтерпретатор визначає лише те, що функції слід застосовувати до обчислення значень аргументів, а не до просто їх символьного подання. Не становить проблем розширити запропонований інтерпретатор довільним довершеним S-виразом. Для прикладу, якщо би ми розширили вихідну мову логічною функцією **(НУЛЬ E),** яка є істинною, якщо E дорівнює 0 і не істинною в противному разі, то цілям інтерпретації служила б така фраза

... якщо рівно(car(e), НУЛЬ) то нуль(вирахувати (car(cdr(e)), n, v)
інакше ...

яку слід ввести в середину побудованої програми інтерпретації. Зміст такої

фрази в тому, що **НУЛЬ** означає те ж саме, що означає нуль і в принципі можна додати будь-які інші правильні S-вирази.

Безсумнівно, можна побудувати і будь-який інший інтерпретатор, надавши всім конструкціям мови той зміст, який ми в них вкладаємо. Переписавши наведені функціональні визначення у вигляді S-виразів, ми отримуємо програму інтерпретації мови Лісп, написану на самому Ліспі. Аналогічно можна написати на Ліспі програму інтерпретації будь-якої іншої мови програмування – чи то алгоритмічної (Алгол, Фортран, Паскаль), чи описової (Пролог, Рефал, Редюс). Для цього слід переписати твердження цієї мови у вигляді S-виразів. Реальна програма інтерпретатора має ряд особливостей, які не носять принципового характеру, але б значно ускладнили розгляд побудови програми інтерпретації функціональних програм.

5.5.5. Контрольні питання та вправи

1. В чому суть інтерпретації програм?
2. Дати визначення псевдофункції **завершення(x,y)**.
3. Навести приклади побудови нескінченних циклічних структур.
4. Дати визначення асоціативного списку та навести приклади його реалізації в S-Ліспі.
5. Розширити інтерпретатор функціональної мови введенням умовної форми **УМОВА**, яка задовольняє таку умову еквівалентності **(УМОВА(Х1 Е1)) ... (ХК ... ЕК)≡(ЯКЩО Х1 Е1 (ЯКЩО Х2 Е2 (ЯКЩО ... (ЯКЩО ХК РК Ω) ...).**
6. **З**апишіть фрагмент інтерпретації правильного виразу **(ЗНАЧ Е)**, який видає значення змінної з іменем Е.
7. Запропонуйте інтерпретацію логічних операцій **(І Е1 Е2); (НЕ Е); (АБО Е1 Е2)**, де **Е, Е1, Е2** – довільні правильні вирази функціональної мови.
8. Пояснити суть функції **замикання** та псевдофункції з**авершення**.
9. В чому суть вирахувань з затримкою і яка їх ціль?
10. Запиати визначення функції на трьох різних мовах програмування (Паскаль, Сі$^{++}$, Лісп) та подати у игляді – виразу їх інтерпретацію.
11. Записати інтерпретацію примітивних функцій Ліспу українською мовою, вибравши для них імена, що відповідають змісту цих функцій.

Розділ 6. Сучасні технології функціонального програмування та засоби для їх реалізації

6.1. Структурно-оптимальне програмування. Функціонали та макроси

6.1.1. Особливості структурно-оптимального програмування

Засвоївши особливості мов функціонального програмування, які описані в попередніх двох розділах пыдручника і реалізовані на версії строго функціональної мови S-Lisp, можна переконатися в тому, що без багатьох структур управління, які характерні для процедурних мов, у функціональному програмуванні можна обійтися. Це передусім стосується функцій передачі управління GOTO, RETURN, PLOG, DO. Врахувавши, що для розгалуження обчислень у функціональному програмуванні є універсальна форма – функція **COND**, можна відмовитися від структур IF, WHEN, UNLESS, CASE. Оскільки повторні обчислення здійснюються через механізм рекурсії, який є базовим принципом функціонального програмування, то можна відмовитися від структур циклів DO, LOOP, WHILE, замінивши їх рекурсивним викликом. Використовуючи принцип вкладеності виразів, можна звести майже до нуля зв'язок змінних за допомогою функції SET, тим самим відмовитися від оператора присвоєння.

Отже, ми майже повністю відійшли від загальноприйнятих прийомів процедурного програмування. Програма, що написана без використання прийомів процедурного програмування з мінімальним застосуванням операцій присвоєння значень змінних, відсутністю додаткових (паразитних) змінних, мінімальним використанням структур управління, прийнято називати програмою, що написана засобами структурно-оптимального програмування. З одного боку, така програма є добре структурованою, легко зрозумілою, наочною, компактною, а з іншої – дає змогу оптимально реалізувати процедуру її відлагоджування і модифікації та перевірки працездатності чи доведення її правильності. Мови функціонального програмування ідеально пристосовані до написання оптимально-структурованих програм.

Як приклад наведемо програму обчислення N! в середовищі S-Lisp:

*** (DE N! (N) (COND ((EQ N O) 1) (T(TIMES N (N! (MINUS N1)))))**

Як бачимо, тут основну роботу виконує рекурсивний виклик.

6.1.2. Основні принципи побудови структурно-оптимальних програм

Отже, побудову оптимально-структурованих програм у функціональному програмуванні можна звести до виконання таких принципів:

- робота з константами із зв'язування змінних;
- послідовне виконання;
- розгалуження обчислень із застосуванням функції **COND**;
- рекурсивний виклик;
- композиція функцій.

Робота з контекстом забезпечується функціями:

- SET, SETQ, SETQQ - зв'язування даних;
- QUOTE - блокування обчислень;
- LAMBDA - виклик λ- виразу та виклик імені функції;
- LET - одночасне зв'язування всіх змінних та доступ до них у визначальному виразі.

Послідовне виконання реалізується функціями:

- PROG1, PROG2, PROGN, які призначені послідовно виконувати записані в них вирази і як результат відокремити один з них (відповідно перший, другий чи N-ний).

Формат функції COND такий:

(COND (L1 E1) ... (LN EN)),

що забезпечує послідовну перевірку умов L1...LN і при виконанні I-тої умови LI буде видане значення EI як результат.

Рекурсивний виклик забезпечується звертанням функції до самої себе. Необхідно потурбуватися про правильність роботи рекурсії (відсутність зациклювання, правильність умов, порядок слідування) та завершення її роботи. У вищенаведеному прикладі правильність роботи рекурсії гарантується тим, що йдуть послідовні виклики функції N! до все меншого значення аргументу N і рано чи пізно ми прийдемо до N = 0, в наслідок чого останній виклик N! видасть значення 1, а результатом всіє функції N! буде добуток чисел від 1 до N.

Принцип композиції функцій реалізується укладанням функцій одна в одну. Можливість такого укладання на довільну глибину забезпечує побудову складних функцій вищих порядків, про що йшлося у четвертому розділі підручника.

6.1.3. Поняття про функціонали та макроси

Ґрунтуючись на єдиному принципі подання даних і програм у функціональному програмуванні, можна як аргумент задати функцію. Аргумент, значенням якого є функція, називають функціональним аргументом або функціоналом, а відповідно визначену функцію – функціональною. В деяких версіях LISP використовують нечіткі визначення функції, тобто виклик функції здійснюється за допомогою аргумента. Тут всі функції можуть бути як функціями вищого порядку, так і першого. У діалектах MU-LISP та COMMON -LISP такого механізму немає. Тут для визначення функціонала застосовуються спеціальні функції **APPLY** та **FUNCALL**, які змушують функцію з заданим іменем працювати з іншими елементами як з аргументами.

Формат запису цих функцій такий:

(APPLY function ‘ (arg1, ... argN));

(FUNCALL function arg1, ... argN).

Приклади використання:

```
* ( APPLY ‘ CONS ‘ ( A ( B C D ))
  ( A B C D )
*  ( APPLY ‘ CONS ‘ ( A B ))
   ( A. B )
* ( APPLY ‘ TIMES ‘ (  5 5 ))
  25
* ( SETQ F ‘ PLUS )
*  ( APPLY F ‘ (7 9 ))
   16
* ( FUNCALL ‘ CONS ‘ A ‘ B )
  ( A.B )
*  ( FUNCALL ‘ PLUS 2 3 )
   5
* ( FUNCALL ‘ APPLY  ‘ LIST ‘ ( A B ) )
  ( A B )
* (APPLY  ‘ FUNCALL ‘ ( LIST ‘ ( A B ) ) )
  ( ( A B ) )
```

При створенні композицій функцій часто буває корисно не виписувати обчислювальний процес, а сформувати його у вигляді програм. Ця ідея найдоцільніше може бути реалізована за допомогою створення макросів (macros), які забезпечують автоматизацію динамічного програмування.

При вирахуванні макросу спочатку з його аргументів будується форма, що задається визначенням макросу, а потім в результаті виклику макросу повертається значення цієї форми. Для визначення макросу застосовується функція:

(DEFMACRO < ім'я макросу > < списки аргументів > < тіло макросу >)

За своєю формою це визначення еквівалентне визначенню функції **DE** чи **DEFUN**, але його обчислення істотно відрізняється від вирахування виклику функції тим, що:

1) аргументи не вираховуються;

2) вирахування макросу реалізується в два етапи:

- на першому етапі вираховується тіло макросу (його розширення);
- на другому етапі вираховується отримана при розширенні форма, значення якої повертається як значення свого макровизначення.

Як приклад визначимо макрос **SETQQ**, що діє як функція **SETQ** з тією різницею, що блокується вирахування другого аргумента.

```
* (DEF MACRO  SETQQ (x y)
        ( SET x ' y )
SETQQ
* (SETQQ LIST ( A B C ))
( A B C )
* LIST
( A B  C ).
```

Можна сформулювати наступні композиції функцій:

1. Звичайний виклик
   (defun f ... (q) ...)
2. Рекурсивний виклик
   (defun f ... (f) ...)
3. Вкладений рекурсивний виклик
   (defun f ... (f ... (f)...)...)
4. Функціональний аргумент
   (defun f (q ...) ... (applay q) ...)
5. Рекурсивний функціональний аргумент
   (defun f (...q ...) ... applay f...f...) ...)

П'ятий тип композиції часто називають автофункціями, тобто такими, що відображають функціонали.

6.1.4. Контрольні питання та вправи

1. Дати визначення структурно-оптимального програмування.
2. Сформулювати основні принципи структурно-оптимального програмування.
3. Пояснити суть дії узагальненої форми **COND** для організації розгалуження обчислень.
4. Яка принципова відмінність між визначенням функції та макросу?
5. Дати визначення функціоналу.
6. Пояснити роботу функції **APPLY** та **FUNCALL**.
7. Суть композиції функції та основні типи композиції.
8. У вигляді макросу задати форму еквівалента оператора **IF** мови Паскаль.
9. Означити макрос, який здійснює копіювання стану функціональної програми.

6.2. Програмування, яке керується даними

6.2.1. Доцільність програмування, яке керується даними

Метод програмування, в якому дані управляють виконанням програми або дані інтерпретуються як програми, називається методом програмування, яке управляється даними (date driver). Цей метод забезпечує можливість конструювати основний алгоритм програми в найзагальнішому вигляді, а дані реалізації такого алгоритму вносити в дані, які можна постійно змінювати, поповнювати, не перероблюючи основну програму. Такий підхід не може бути реалізований в термінах традиційних мовах програмування, в той час як у функціональних мовах програмування це є можливим завдяки спеціальним функціям **(APPLY, FUNCALL),** функції **EVAL** та макровизначення **DEFMACRO.**

Програмування, яке управляється даними, набагато більше ніж просте операторне програмування підходить до розв’язання задач штучного інтелекту, оскільки дає змогу розробити достатньо універсальні алгоритми, постійно їх розширювати та модифікувати.

Цей підхід найдоцільніший, коли існує багато різноманітних, але аналогічних типів даних, які можна класифікувати за певними ієрархічними критеріями. Зокрема, це стосується розроблення програм опрацювання людської мови, розпізнавання почерку, розпізнавання зображень,

ідентифікація об'єктів, побудова експертних систем в тих галузях людської діяльності, які не піддаються строгому математичному опису.

На цьому програмуванні базуються деякі спеціальні мови, такі як ATN (augnehted transitation network grammar) – обробка мереж перехідних станів, які описуються спеціальними граматиками та метод класифікаційних сіток, в якому дані спочатку аналізуються, а вже потім відповідно до результатів аналізу опрацьовуються.

Близьким до цього є програмування, яке управляється подіями (even-action draven). Це такий стиль програмування, коли дії активізуються лише при виникненні конкретних ситуацій, при виклику заданої функції чи визначенні конкретної змінної. Функція або змінна, що виконується, називається демоном. Така назва зумовлена тим, що демон ніби чекає, коли та чи інша подія дасть змогу діяти і йому. Принцип реалізації такого підходу при незначних модифікаціях може ґрунтуватися на програмуванні, яке управляється даними.

6.2.2. Алгоритм організації програми, що керується даними та приклад реалізації

Реалізація цього методу програмування, яке управляється даними, може бути реалізована за схемою:

1. Створюється функція або макровизначення привласнення даним (символу) значення, функції чи властивості у вигляді виразу або спеціально розробленої функції.

2. Створюється управляюча функція, яка здійснює виконання функції, яка визначена для даних в пункті 1.

3. За допомогою виклику функції або макровизначення, що створено у пункті 1 визначається ряд символів з заданими значеннями чи властивостями у вигляді списку виразів, які виконують дії або у вигляді іншої заздалегідь розробленої функції.

Як приклад реалізації підходу розглянемо фрагмент розробки функції диференціювання виразу, коли в нього входять змінні, константи та операції додавання та множення.

Приклад реалізації на основі розгалуження має вигляд:

```
( defun dif  ( l x )
 ( cond
          ((atom l) (if (eg e x) 10 )))
```

```
(( eg ( car e ) ‘ f ) ( list ‘ f ( dif ( cadr 1 ) x ) ( dif ( caddr e ) x)))
(( eg ( car e ) ‘ * )  ( list ‘ f ( list ‘ * dif ( cadr l ) x) ( caddr e )) ( list ‘ *
(cadr e ) ( dif ( caddr e ) x ))))
( t NIL )
* ( dif ‘ ( + x ( *2x )) x )
 ( +1 ( + ( * 0 x ) ( * 21 )))
```

В цій програмі «+» відповідає операції додавання, «*» – операції множення і в S-виразі дозволено використання малих літер. При потребі розширення можливості програми необхідне внесення змін в програму.

Якщо застосувати функціонал для вирахування похідної, то приходимо до визначення:

```
(defun dif 1(l x)
(cond
((atom l) (if (eg l x) 01))
(t (funcall (eval (car e)) (cdr e) x))))
(setg f + ’ div’’ f * ’div *)
(defun div + (lx)
(list ‘ +(dif 1(  car e) x ) ( dif 1 (cadr e) x )))
(defun dif * (lx )
(list ‘ + (list ‘ *(dif 1 (car e) x ) (cadr e))
(list ‘ *(car e) (dif 1 (cadr e) x))))
*(dif 1 ‘  (F+ x (f*2 x )) x )
(+ 1 (+ (* 0 x) (x 21 )))
```


В цьому прикладі операція «+» замінена символом F+ і «*» символом F*, оскільки не у всіх версіях LISPу можна здійснити визначення значень для символів + та *. Цей приклад ілюструє метод програмування, що керується даними у вигляді імен функцій, які задані значеннями символів F+ та F*. Якщо необхідне розширення програми, головна програма не змінюється, а лише доповнюється функціями диференціювання для конкретних операцій.

6.2.3. Суть методу зіставлення зі зразком

В програмуванні, що управляється даними, широко використовується метод зіставлення зі зразком . Зіставлення зі зрізком – це процедура, при якій здійснюється порівняння (зіставлення) деякого зразка заданої структури даних з конкретним образом для виявлення їх відповідності.

Цей метод широко використовується в галузях, які пов’язані з розпізнаванням образів (pattern recognition), зокрема в системах штучного

зору, ідентифікації людської мови, автоматизованого спілкування людини з машиною, системах перекладу, розпізнавання почерку людини, продукційних та логічних системах програмування для створення експертних систем.

Розглянемо розпізнавання спискових образів, що має багато спільного з розпізнавання образів, які представлені в іншому вигляді, оскільки в мові LISP існує достатньо багато методів перетворення різноманітних структур даних в списковy форму та в зворотному напрямку.

Найпростіше процедуру зіставлення можна реалізувати, поставивши вимоги розпізнавання однакових елементів списку. Для більшої гнучкості програми введемо символи – замінювачі. Розрізнятимемо такі випадки:

? - довільний непорожній елемент образу;

_ - довільний елемент образу, можливо і порожній;

& -непорожня послідовність елементів;

! - довільна послідовність елементів, яка може бути порожньою.

Введемо формат функції **зістав:**

(зістав зразок образ)

Приклади:

```
* ( зістав '' ? ? р ? ) (  пара )
T
* ( зістав '' ( & р а ) (  пара )
T
* ( зістав '' ( ? ст & ів ! ) ((  група ) ( студентів )
T
```

Реалізація цієї програми в середовищі MU-LISPу має вигляд:

```
( defun зістав ( n  h )
( cond )
(( null n ) ( null h )
(( null h ) nil )
(( egual ( car n ) ( car h ); якщо голови елементів здійснюються,
 ( зістав (  cdr n ) ( cdr h ))); то переходимо до хвостів
                              ; якщо ні то перевіряємо,
                              чи є у символу зразка властивості
(( and ( atom (car (n)) (get ' (car n ) ' зіставник) h )
                             ; якщо так, то виконуємо властивості
                            ; функціонала FUNCALL
( funcall ( get ( car n ) ' зіставник n h ))
( T NIL )                     ; якщо ні, то повертаємо NIL
```

```
; макровизначення функції зіставник
( def macro  def зіставник ( символ      ключ      властивість )
( list ‘ put символ ‘ зіставник
( list ‘ quote ( list ‘ lambda  ключ властивість )))
; довільних символів - ? :
( def зіставник ? ( n h )
        ( зістав ( cdr n ) ( cdr h )))
; зіставник не пустих послідовностей - &:
( def зіставник & ( n h )
( or ( зістав (cdr n) (cdr h)) (зістав n ( cdr h))))
; зіставник можливо пустих послідовностей - !:
(  def зіставник ! ( n h )
( or ( зістав (cdr n) (cdr h)) (зістав ( cdr n) h) ( зістав n ( cdr h ))))
; зіставник довільних символів, яким можуть бути пусті - -:
(  def зіставник - ( n h ))
( or ( зістав (cdr n) (cdr h)) (зістав ( cdr n) h)))
```

Ця версія працює лише з верхнім рівнем списку, але немає проблем організувати програму **зіставлення** і в глибину списку.

6.2.4. Контрольні питання та вправи

1. Суть методу програмування, яке управляється даними.
2. Алгоритм реалізації програмування, яке управляється даними.
3. Яка особливість програмування, що управляється подіями?
4. Суть методу зіставлення зі зразком?
5. Модифікувати програму **зістав** образу зі зразком так, щоб зіставлення відбувалося по всіх рівнях ієрархії.
6. Забезпечити в програмі **зіставлення** зі зразком можливості зберігати повернення тих елементів, які замінюються символами зіставлення.

6.3. Об’єктно-орієнтоване програмування

6.3.1. Основні поняття

Найабстрактнішу і найрізноманітнішу з моделей програмування, які існують сьогодні, ілюструє об'єктно-орієнтоване програмування (ООП). Основними принципами об'єктно-орієнтованого програмування є: абстрагування, ієрархічність, наслідування, поліморфізм, типізація, паралелізм і тривалість. Ці принципи не є новими, але саме в ООП вони

об’єднанні для вирішення головної задачі і є логічним продовженням ідей структурованого і модульного програмування. ООП спрямоване передусім на полегшення проектування, написання, відладжування, верифікації та супроводу складних програмних систем.

В об’єктному програмуванні не допускається, щоб програма складалася з набору підпрограм і даних. Основними одиницями об'єктно-орієнтованої програми є об’єкти, які містять як процедури, так і дані. Ідея об’єднання даних і програм є основою об’єктного програмування та створення об’єктної моделі.

Об’єкт містить в собі як дані, подібні до структурам, так і дії, подібні до функцій і підпрограм. Дії, які передбачені в об’єкті, відображаються за допомогою методів (methods). Дані, що містяться в об’єкті, знаходяться в змінних, які називаються змінними примірника або властивостями.

У об’єктах є визначення і виклик, за допомогою якого створюється конкретний або фактичний об’єкт чи примірник.

Визначення об’єкта – це абстрактний опис типу об’єкта, що описує всі об’єкти цього класу. Тип об’єкта часто називають класом об’єктів. Клас – це безліч об’єктів, що можуть виникнути з його визначення.

6.3.2. Створення об’єктно-орієнтованої програми

Для створення об’єктно-орієнтованої програми в MU-LISP потрібно за допомогою команди **LOAD** запустити файл flavors.lsp:

(LOAD ‘flavors.lsp) .

В системі flavors форма визначення класу така:

(DEFLAVOR клас (властивість)
(підкласи)
(режими)
)
Приклад опису класу
DEFLAVOR проект
(назва теми
Керівник
Учасники
Початок - роботи
Кінець - роботи
Обсяг - фінансування
( ); нічого не слідує

; ..... опцій

)

Можливі опції

: GETTABLE - INSTANCE - VARIABLE

: SETTABLE - INSTANCE - VARIABLE

Якщо опції вказані, то автоматично створюються методи доступу до властивостей об'єкта.

Якщо опції ввести в програму, отримаємо такі методи:

*** для питання**

: назва теми, : керівник,: учасник і т.д.

*** для запису**

: SET - назва теми,: set : - керівник і т.д.

Для отримання доступу до окремих властивостей об'єкта використовуємо опцію:

(: GETTABLE - INSTANCE - VARIABLE < змінні >).

Метод – це функційні властивості об'єкта, що визначають його застосування у вигляді процедури. Метод можна активізувати, якщо послати відповідне повідомлення, що складається з назви методу та його аргументів.

Крім основного, існують методи-демони (або допоміжні), які оголошуються разом з основним. Демони бувають двох сортів: перед і постдемони (before і after damon). Визначення методу має таку форму:

(DEFMETOD (<клас> <тип методу> <метод>)

<лямбда список>

<форма>

).

Тип методу використовують для опису перед і постдемона

: BEFORE - переддемони

:AFTER – постдемони**.**

В середині опису методу властивості об'єкта доступні як оновленні змінні по імені. Нові значення властивостям присвоюються за допомогою SETQ.

Як приклад опишемо клас **крапка**

(DEFFLAVOR крапка

(x y); кординати крапки

: GETTABLE - INSTANCE - VARIABLE

: SETTABLE - INSTANCE – VARIABLE)

Для руху крапки опишемо метод:

: MOVE _ TO
(DEFMETOD (крапка: MOVE _ TO)
(NEW -X) (NEW - Y)
(SETQ X NEW-X)
(SETQ Y NEW-Y))

Необхідно забезпечити, щоб координати крапки були позитивні. Для цього зробимо перевірку у вигляді додемона.

(DEFMETOD (крапка: BEFORE: MOVE _ T[8])
(NEW - X NEW - Y
((OR (MINUS P NEW - X)
(MINUS P NEW - Y)
) (BREAK (LIST NEW - X NEW - Y) ‘’ координати = < 0 ‘’))

При створені нового примірника об’єкта, нові значення властивостей об’єкта присвоюються в методі: **INIT**. Доступу в цей метод немає, але можна описати метод : **AFTER**: **INIT** і там виконати всі необхідні дії після ініціалізації. Створюється конкретний примірник об’єкта за допомогою функції **MAKE-INSTANCE.** Продемонструємо роботу функції на практиці з об’єктом класу **проект**:

(SETQ ПРОЕК 10 (MAKE-INSTANCE
’ ПРОЕКТ
: назва теми « Побудова ЕС »
: керівник « Товранов А.І. »
: учасник « (Ольга, Іван, Андрій) »)

Виклик **(MAKE-INSTANCE клас)** повертає конкретний примірник об’єкта класу. В **MAKE-INSTANCE** не обов’язково вказувати всі властивості об’єкта, якщо якісь із них мають значення по замовчуванню.

Так, при описанні класу:

(DEFLAVOR крапка
(X 0); (0,0) початкове
(Y 0); значення ( ))

Крапка з координатами (0 0) створюється так:

(SETQ крапка_ 0 0 (MAKE-INSTANCE ’ крапка))

Метод об’єкта можна активізувати, якщо надіслати йому відповідне повідомлення. Повідомлення складається з імені методу і аргументів. У файлі FLAVORS. LST це робиться за допомогою функції **SEND**:

(SEND <об’єкт> <метод> <аргумент>)

Наприклад:

(SEND КРАПКА 1 : MOVE_TO 40 50)

Приймаючий об'єкт містить механізм, який розрізняє повідомлення, вибирає відповідний метод, активізує його і передає відповідні параметри, що містяться в повідомленні. Обчислення методу може змінити стан об'єкта і в результаті побічних ефектів бути причиною надсилання нових повідомлень іншим об'єктам.

За допомогою **SEND** викликати явно демон неможливо. Проте в середині описання методу можна використати **SELF**, щоб об'єкт надіслав повідомлення самому собі:

(**SEND SELF <метод><аргумент>)**

6.3.3. Приклад реалізації

Як приклад запишемо програму, яка створює два вікна, використовуючи віконну систему WINDOWS

(SETq main (MAKE-INSTANCE 'WINDOWS
: top 12
: left 0
: height 12
: width-60
: title '' MAIN ''
: frame '' duble - frame ''))
(Setq small (MAKE-INSTANCE' WINDOWS
: top 12
: left 0
: height 12
: width-60
: title '' MAIN ''
: top 2
: left 10
: height 8
: width-60
: title '' SMALL ''
: windows-color' (71)
: lorder-color' (71)
: titele-color' (141)
: frame ''single-frame''
(SEND small : expose)

(SEND main : expose)
(SEN small : princ' hello 2 2)

6.3.4. Контрольні питання та вправи

1. Основні принципи об'єктно-орієнтованого програмування.
2. Визначити поняття об'єкта, класу, підкласу.
3. Ввести форму для опису класу.
4. Поняття методу в ООП та класифікація методів
5. Описати процедуру створення нового примірника об'єкта.

6.4. Програмні засоби мови MU-LISP і COMMON-LISP та їх основні функції

6.4.1. Опис середовища MU-LISP та COMMON-LISP

Матеріал пыдручника, який викладений в четвертому та п'ятому розділі відноситься до строго функціональної мови, без будь-яких сторонніх ефектів. Описані тут програми реалізовані в середовищі S – LISP, яке максимально наближене до строго функціональної мови, в якій достатньо одного інтерпретатора, щоб давати строгі визначення функцій.

Необхідність розроблення потужних функціональних програм, потребує засобів швидкого редагування, відлагоджування, модифікації та супроводження програмних продуктів. Це, в свою чергу, вимагає відходу від строгого визначення функцій і може бути реалізовано в середовищі MU-LISP та COMMON-LISP.

Базовий склад пакета MU-LISP та COMMON-LISP IBM PC version 6.10 складають наступні файли:

I. Виконавчі файли:

1) MU LISP.COM – компілятор та інтерпретатор системи;

2) COMTOEXE.COM – утіліта перетворення ком-файлу в exe-файл.

II. Файли редактора і відладки:

1) EDIT.LISP- вхідний файл редактора;

2) DEBUG.LSP- вхідний файл відладки.

III. Файли бібліотек:

1) INTEPLISP.LSP – функції та утиліти INTER-LISP;

2) COMMON.LSP – функції, утиліти, макроси COMMON.LSP;

3) GRAFICS.LSP – графічні функції;

4) MOUSE.LSP– функції інтерфейсу миші.

IV. Файли системи навчання:

1) LESSON.LSP – файл основних знань;

2) MULISP.LSP – текстові файлі для навчання.

V. Додаткові файлі для розширення можливостей:

1) **FLAVORS.LSP–** функції об'єктно –орієнтованого програмування;

2) STRUCTUR.LSP– функції визначення структури.

Для запуску MU-LISP треба ввести команду

>[pash] MULISP,

де **pash** – ім'я пристрою і підкаталогу, де знаходиться основний виконавчий модуль MULISP.EXE або MULISP.COM.

Після запуску на екрані з'являється повідомлення про необхідність вводу номера комп'ютера. Для IBM сумісних комп'ютерів вводиться 2.

Для виходу з системи необхідно ввести

$ (SYSTEM)

Для виклику редактора вводиться функція

$(LOAD ' EDIT)

Для запуску програми використовується ця сама функція

$(LOAD ' PROGRAM)

LISP-програми можна писати, використовуючи будь-якій текстовий редактор і зберігати в файлах з розширенням LSP або використовувати редактор EDIT.LISP, який також написаний на LISPI.

З системою MU-LISP також можна працювати в інтерактивному режимі, вводячи послідовно завдання та отримуючи результат.

Інтерпретатор MU-LISP дає можливість зберігати файли у вигляді COM, EXE та SYS. Для цього використовується функція **SAVE**

(SAVE<ім'я файлу>.[COM])

Якщо розширення відсутнє, то створюється SYS-файл, який зберігає середовище, що створилося в даному сеансі роботи. COM-файл, крім середовища, зберігає ще й інтерпретатор MU-LISP. При розмірі COM-файла, більшому за 64 Кб, програма не зможе працювати, тому передбачена можливість його перетворення в EXE-файл за допомогою утиліти COMTOEXE.COM:

COMTOEXE.COM<COM-файл><EXE- файл >

Для редагування та зберігання файлів використовується EDIT.LISP, який запускається як звичайна програма, оскільки він написаний на LISPi. Після входу в редактор з головного меню вибирається A(alfa). Після цього система переходить в режим редагування.

Основні команди редактора

CTRL + N – вставка рядка**,**

CTRL + Y вилучення поточного рядка**,**

CTRL + K+B – відзначення початку блоку,

CTRL + K+K – відзначення кінця блоку,

CTRL + K+C – копіювання блоку.

Для переходу в середовище MU-LISP з головного меню вибирається H. Для повторного повернення до редактора необхідно викликати функцію (**RETURN (RETURN))** або натиснути **ESC**. Для завантаження файла в редактор необхідно вибрати з головного меню **T (TRANSFER)**, потім **L (LOAD**) і ввести ім'я файла. При записі відредагованого оригіналу вибрати **S (SAVE)** з головного меню. Для виходу з редактора вибрати **Q (QUIT)** в головному меню.

6.4.2. Арифметичні функції MU-LISP та COMMON-LISP

Для зв'язування S-виразів із символами використовують функції SET та SETQ:

(SET <S – вираз><S – вираз>)

(SETQ<символ><S – вираз >).

При цьому функція SET обчислює перший елемент перед зв'язуванням, а функція SETQ зв'язує символ, не вираховуючи його. Наприклад:

$(SET ' A '5)

5

$A

5

$(SET(CAR(F A B))(+ 2 9))

7

$f

7.

Найвживаніші арифметичні функції MU-LISP:

1. (+N_1 N_2 .. N_M)- повертає суму чисел від N_1 до N_M,

2. (*N_1 N_2 .. N_M)- повертає добуток чисел від N_1 до N_M,

3. (- N_1 N_2 . . . N_M)- повертає різницю чисел між N_1 та суму від N_2 до N_M,

4. (- N_1 N_2 .. N_M)- повертає результат ділення чисел N_1 на добуток чисел від N_2 до N_M. Якщо функція має один аргумент, то повертає обернене число.

5. (ADD1 N) – повертає N +1,

6. (SUB1 N) – повертає N – 1,

7. (MOD N_1N_2) – повертає залишок від ділення N_1 на N_2,

8. (MAX X_1….X_N) – повертає найбільший з аргументів,

9. (MIN X_1….X_N) – повертає найменший з аргументів,

10. (SIGNUM X) – повертає 1, якщо x>0; 0, якщо x=0;-1, якщо x<0,

11.(ABS X) – повертає абсолютне значення X,

12.(TRUNCATE X) – повертає цілу частину від X.

В MU-LISP відсутні математичні функції, хоча всі можуть бути побудовані на основі наявних арифметичних. Але в цьому не має потреби, оскільки всі функції визначені в COMMON-LISP. Для розширення MU-LISP до COMMON-LISP необхідно виконати такі дії:

$(LOAD ` COMMON).

Основні математичні функції COMMON-LISP такі

a. **(EXP X) –** повертає e^x,

b. **(EXPT X^N) –** повертає X^N,

c. **(LOG N M) –** повертає $\log_M N$,

d. **(LN X) –** повертає ln X,

e. **(SQRT N) –** повертає корінь квадратний з N,

f. **(PI) –** повертає значення числа π,

g. **(SIN X) –** повертає sin X (x в радіанах),

h. **(COS X) –** повертає cos X (x в радіанах),

i. **(TAN X) –** повертає tg X(x в радіанах),

j. **(ATAN X) –** повертає artg X(x в радіанах),

k. **(ASIN X) –** повертає arsin X(x в радіанах),

l. **(ACOS X) –** повертає arcos X(x в радіанах),

m. **(RANDOM X) –** повертає випадкове число X($0<X\le N$, N – ціле число).

6.4.3. Предикати мови

Основні числові предикати, які використовуються для порівняння чисел:

1. **(= $N_1N_2…N_N$) –** повертає T, якщо числа рівні,
2. **(/= $N_1N_2…N_N$) –** повертає T, якщо числа не рівні,
3. **(< $N_1N_2…N_N$) –** повертає T, якщо числа зростають,
4. **(> $N_1N_2…N_N$) –** повертає T, якщо числа спадають,
5. **(<= $N_1N_2…N_N$) –** повертає T, якщо числа не зменшуються,
6. **(>= $N_1N_2…N_N$) –** повертає T, якщо числа не зростають.

Наведені функції здійснюють порівняння попарно. Предикат повертає T, якщо всі порівняння істинні, в протилежному випадку повертається NIL, який є ознакою фальшивості предиката . Для розпізнавання типів даних використовується предикати:

1. **(INTEGEPP X) –** повертає T, якщо X – ціле число,
2. **(NUMEPP X) –** повертає T, якщо X – число,
3. **(ZEROP X) –** повертає T, якщо X = 0,
4. **(PLUSP X) –** повертає T, якщо X ≥ 0,
5. **(MINUSP X) –** повертає T, якщо X < 0,
6. **(ODDP X) –** повертає T, якщо X – непарне число,
7. **(EVENP X) –** повертає T, якщо X – парне число,
8. **(ATOM X) –** повертає T, якщо X – не є списком.

Найвживаніші логічні предикати:

(AND < S – вираз>< S – вираз >) – логічне “і”,

(OR < S – вираз>< S – вираз >) – логічне “або”,

(NOT < S – вираз>) – логічне “ні”.

Якщо в наведених функціях S-вираз не повертає NIL, то він вважається істинним. Предикати порівняння символів та S-виразів такі

1. **(EQL <символ>< символ >) –** повертає T, якщо символи ідентичні,
2. **(EQUAL < S – вираз ><S – вираз>) –** повертає T, якщо S-вирази індентичні,
3. **(NULL <S – вираз >) –** повертає T, якщо S-вираз є порожнім списком.

6.4.4. Функції роботи з рядками

В COMMON-LISP є рядковий тип STRING, який позначається \STRING. В цьому виразі рядок є константою і не може набувати значення, властивість чи бути функцією. До рядкових функцій належать:

1. (**UNPACK <ATOM>) –** повертає список символів з іменами атомів. Якщо аргумент не атом то вертає NULL,
2. **(PACK< символ >)** – повертає символ, який складається з зчеплених елементів списку.

Наприклад:

(UNPACK ‘ABCD) ----- (ABCD)
(UNPACK - 345) -----– (-\3 4 \5)
(PACK ‘(ABC)) -----– ABC
(PACK’ (-\3 4 \5)) ----- – - 345.

6.4.5. Функції опрацювання списків

Як і в S- LISPі так і в MU-LISPі базовими є п’ять функцій для опрацювання S-виразів (атомів, списків, крапкових пар):

1. **(CAR X) –** повертає перший елемент списку,
2. **(CDR X) –** повертає залишок списку без першого елементу,
3. **(CONS <S- вираз><список>) –** формує новий список, де S-вираз стає першим елементом списку,
4. **(ATOM < S- вираз >) –** повертає T, якщо S- вираз є атомом,
5. **(EQL< символ >< символ >)–** повертає T, якщо символи і ідентичні.

Відзначимо, що у випадку використання в якості S- виразів крапкових пар, другий аргумент **cons** може бути і атомом. Наприклад:

$(CONS’A’B)
(A.B).

Інші найбільш вживані в MULISP функції для опрацювання S- виразів:

1. **(EQUAL <S-вираз><S-вираз>) –** повертає T, якщо S-вирази є ідентичні.
2. **(NULL X) –** повертає T, якщо X є порожнім списком.
3. **(LIST<S-вираз1><S-вираз2>...<S-виразN>) –** формує єдиний список.

Наприклад:

(LIST ‘A’B) ----- (AB)

(LIST(LIST A)(+23)) ----- ((A)5).

4. (LAST<список>) – видає останній елемент списку.

5. (APEND <список1><список1>) – видає новий список, де перераховані складові двох списків.

6. (NTH n <список>) – видає n-й елемент списку.

Для роботи з асоціативним списком пар типу

((A1 E1)(A2 E2)…(AN EN))

передбачені такі функції:

1. (PAIRLIS <ключі><дані><а-список>) – заносить в <а-список> список пар на основі списку ключів та списку даних.

2. (ASSOC <ключ><а-список>) – видає пару з списку що відповідає ключу.

3. (ASONC X Y<а-список>) – додає нову пару (XY) в початок а-списка.

В MU-LISP передбачені функції роботи зі списками властивостей символів. Список властивостей може складатися з крапкової пари, де на першому місці йде ім'я властивості, а на другому – її значення або з прапора, який позначається вільним атомом. Наприклад, символ НУЛПІ може мати наступний список властивостей:

(місце..Львів)(тип.Політехнічний) Національний)

До цих функцій належать:

1. (PUT <символ><ключ ><значення>) – задає значення властивості.

2. (GET<символ><ключ >) – повертає значення властивості.

3. **(FLAG<символ><атом >) –** повертає <атом > і дописує в початок списку цей прапор.

4. (FLAGP<символ><атом >) – перевіряє наявність <атом > в списку.

5. (REMPROMP<символ><ключ >) – повертає значення властивості та видаляє її з списку.

6. **(REMFLAG<символ><атом >)–** повертає <атом > і видаляє його зі списку.

6.4.6. Структури керування

До структур керування відносяться такі групи:

1) Робота з контекстом:

- **SET, SETQ –** зв'язування змінних,

- **QUOTE –** блокування вираховувань,
- форми **LET,**
- виклик функцій чи **λ - виразу.**

2) Послідовне виконання
- форми **PROG1, PROG2, PROGM.**

3) Розгалуження обчислень
- умовні форми **COND, IF, WHEN, UNLESS;**
- форми вибору **CASE;**
- неявні форми **COND.**

4) Ітерації
- циклічні форми **DO, LOOP.**

5) Підпрограми та передача управління
- **PROG, GO, RETURN.**

6) Динамічне вирахування
- **THROW, CATCH.**

Для доступу до всіх керуючих структур необхідно завантажити модуль **COMMON.LSP.**

При роботі з контекстом, на відміну від функції SET, мова про яку вже велася, форма LET встановлює всі зв'язки одночасно, а вже після цього виконуються вирази. Формат форми LET такий:

(LET((X1 E1)(X2 E2)…(XN EN)) ВИРАЗ1 ..... ВИРАЗ N)

При послідовному керуванні послідовно виконуються всіх вирази у формі PROG. Різниця цих форм в тому, що повертають вони відповідно як результат <вираз1> <вираз2> та <виразN>. Формат форми такий:

(PROG1 <ВИРАЗ1> < ВИРАЗ2> < ВИРАЗN>

для всіх модифікацій.

Універсальною умовною формою є функція **COND**:

(COND
(P1 A11 A12….)
(P1 A21 A22….)
……..
(PN AN1 AN2….)
).

Якщо умова P_i виконується, то обчислює ряд виразів $A_{i1}.....A_{iN}$ і як результат видається значення останього A_{iN}. Неявний COND має такий же вигляд аргументів, тільки COND з відповідною відкриваючою і закриваючою дужкою опускаються.

Інші умовні форми IF,WHEN, UNLESS є частковими випадками COND і відповідають відповідним операторам процедурних мов.

Формат пропозиції **CASE** такий

(CASE<ключ>)
<список ключів 1> m11 m12…
<список ключів 2> m21 m22…
……..
<список ключів N> mN1 mN2…

Значення <ключ> порівнюється зі списками ключів. Якщо знайдений K_i, то використовують вираз mI1, mI2…, останній з яких і є результатом даної пропозиції.

Для організації ітераційного процесу існує пропозиція циклу DO

(DO((<var1> <значення1> [крок1])…)
(<умови закінчення> <вираз1>…<виразN>)
(<вираз21> <вираз22>...).

Тут varI – змінні і значенняI – значення змінних на першому кроці циклу, [крок1] – необов'язковий параметр, який задає значення на наступних кроках циклу. Якщо умова закінчення циклу не виконується, то виконується вираз 21, вираз22 і останній з них є результатом циклу. При виконані умови враховується вираз1, вираз2 і останній з них є результатом форми.

Пропозиція LOOP має формат

(LOOP<форма1><форма2>...<формаN>)

і повторно в послідовному порядку враховує всі форми доти, доки не зустрінеться умовна форма, що не дорівнює NIL. Після цього враховується тіло умовної форми і видається як результат.

Для використання програм, які раніше написані на процедурній мові, існує функція PROG, яка має такий формат:

(PROG
(var1, var2…)
<вираз1>
m
<вираз2>
…….
(IF<умова>(RETURN Z)
…….
(GO m)
)

Всі параметри varI є локальними і доступні лише всередині форми PROG; m – набуває як символьне, так і числове значення і служить міткою оператора GO.

Ще використовуються оператори (RETURN РЕЗУЛЬТАТ) для виходу з форми і видачі її результату.

Для організації динамічного управління існує функція CATCH, що має такий формат:

((CATCH мітка_1 форма_1 форма_2 ..)

Ця функція виконується доти, поки в одній із форм не зустрінеться вираз

(TROW<мітка> < значення>)

і мітка_1збігається з обчислювальною міткою. У такому разі виконання CATCH припиняється і як значення видається другий параметр функції TROW. Отже, це дає можливість перейти на вищий рівень вкладення.

6.4.7. Функції вводу та графічного опрацювання інформації

Для спілкування з користувачем передбачені функції вводу-виводу. Відкриття файлів реалізується такими функціями:

(OPEN – INPUT – FILE <ім'я файлу>) – для файла вводу,

(OPEN – OUTPUT – FILE< ім'я файлу >) – для файла виводу.

Аналогічні функції закриття файлів вводу-виводу:

(CLOSE – INPUT – FILE <ім'я файла>)

(CLOSE – OUTPUT – FILE<ім'я файла>).

Якщо для вводу використовується клавіатура, то за допомогою функцій (CLEAR – INPUT) можна очистити буфер вводу. Для зчитування рядка з вхідного пристрою використовується функція (READ BYTE). Функція (READ BYTE) повертає код введеного символу з клавіатури

Функції роботи з екраном консолі створені для виведення на екран тексту та його форматування. Функція

(SET – CURSOR <рядок > колонка) – переміщає курсор в заданий рядок і колонку і видає T,

(ROW) – повертає положення курсора в рядку,

(COLOMN) - повертає положення курсора в колонці,

(CLEAR SCREEN) – очищає вікно, переміщує курсор в верхній лівий кут екрану і повертає T,

(MAKE_WINDOW <рядок><колонка><рядок><колонка>) – створює на екрані прямокутну область як поточне вікно, переміщує курсор в верхній лівий кут екрану і повертає T,

(MAKE_WINDOW) – повертає список з 4 параметрів,

(FOREGROUND – COLOR<N>) – встановлює колір N переднього плану і повертає попереднє значення,

(BACKGROUND – COLOR<N>) – встановлює колір N заднього плану і повертає попереднє значення.

Якщо системна змінна "BLINK" не NULL, то повідомлення будуть блимати, в протилежному разі ні.

Якщо системна змінна "HIGH - INTENSITY" не NULL, то нові символи будуть світитись яскравіше, якщо NULL – яскравість буде понижена.

При роботі MU-LISP на IBM PC з кольоровим графічним дисплеєм різним значенням <N> відповідають кольори

0	**Чорний**	**8**	**темно-сірий**
1	**Синій**	**9**	**ясно-блакитний**
2	**Зелений**	**10**	**світло-зелений**
3	**Ціановий**	**11**	**світло-ціановий**
4	**Червоний**	**12**	**світло-червоний**
5	**Фукстовий**	**13**	**світло-фукстовий**
6	**Коричневий**	**14**	**жовтий**
7	**світло-сірий**	**15**	**білий**

Для створення графічних програм в LISPі існує набір графічних функцій. Графічні примітиви можна створювати за допомогою таких функцій:

(PLOT_DOT X Y color) – малює крапку у вказаних координатах з зазначеним кольором,

(PLOT_LINE X1 Y1 X2 Y2 color) – малює відрізок прямої з відповідними координатами з зазначеним кольором,

(PLOT_ CIRKLE X Y color) – малює коло з зазначеним кольором,

(READ_DOT X Y) – повертає колір крапки.

Ці функції; крім READ_DOT, працюють як в графічному режимі так і в текстовому. Підключення графічного режиму реалізує функція:

(VIDEO_MODE N),

де N приймає такі значення:

0 – текстовий режим 40*25*16
1 – текстовий режим 40*25*26
3 – текстовий режим 80*25*16
4 – графічний режим 320*200*4
5 – графічний режим 320*200*4
6 – графічний режим 640*200*2
13 – графічний режим 320*200*26
14 – графічний режим 640*200*16
16 – графічний режим 640*350*16.

Для відтворення звуку існує функція

(TONE <time><tonus>),

де **<time>** – час звучання в мікросекундах,

<tonus> – тональність звуку.

6.4.8. Застосування функціоналів та макросів

Середовище MU-LISP дає змогу створювати функції вищих порядків, тобто функції, які застосовують інші як параметри. Передача функції як параметра іншій є основою нових технологій програмування, таких, як програмування, що керується даними (data driven programmng) та об'єктно-орієнтованого програмування (objects programming). Аргумент, значенням якого є функція, називається функціональним, а відповідно функція – функціоналом. Цілям створення функціоналів служить функція **APPLAY** та **FUNCALL**

Формат цих функцій є такий:

(APPLAY<функція> <список параметрів>) – змушує <функцію> працювати з елементами списку аргументів як з її аргументами,

(FUNCALL <функція><арг1><арг 2><арг 3>...<арг N>) – аналогічно APPLAY, лише аргументи для функцій викликаються не списком, а кожен окремо. Наприклад:

```
$(FUNCALL 'CONS'A'(A  B  C   D))
(AABCD)
$(FUNCALL (CAR (+ - 1) 2  3)
5
$(FUNCALL 'LIST'(AB))
```

((AB))
$(FUNCALL ‘LIST’ APPLAY’(A B))
(APPLAY’(A B))
$(APPLAY’*’(5 6))
30
$(APPLAY’APLY’(+(6 3)))
9.

Функції, що відображають (map) список у нову послідовність, або породжують побічний ефект, називаються MAP – функціями. Функція

(MAPCAR <функція><арг1><арг 2><арг 3>...<арг N>) –

виконує послідовно дію <функція> над кожним саг-елементом <арг> доти, доки не вичерпається весь список аргументів. Наприклад:

$(SET Q X’(ABC))
(ABC)
$(MARCAP’ATOM X)
(T T T)
$(MARCAP’CONS’ X’(123))
((A.1)(B.2(C.3)).

Функцію MAPCAR можна було би перевизначити за допомогою функціоналу

(DEFUN MAPCAR 1 (FN S))
(IF (NULL S) NIL
(CONS (FUNCALL FN(CAR S)(MAPCAR1 FN(CDR S)))

Середовище MU-LISP дозволяє реалізовувати ідею автоматичного динамічного програмування введенням макросів (MACRO). Після виклику макросу спочатку будується форма, що задається визначенням макросу і як результат повертається значення цієї форми. Формат макросу такий:

(DEFMACRO <ім’я макросу><параметри макросу><тіло макросу>)

Визначення і виклик макросів за формою збігається з визначенням та викликом функції, але в макросі аргументи не враховуються, а спочатку визначається форма макросу (етап розширення), далі йде врахування цієї розширеної форми, яка повертається як результат макросу.

Як приклад опишемо макрос копіювання списку:

(DEFMACRO COPY_LIST(X))
((NUL X) NIL)
((LIST ‘CONST(LIST ‘QUOTE(CAR X))
(LIST ‘COPY_LIST(CDR X))))

6.4.9. Контрольні питання та вправи

1. Основні команди редактора MU-LISP.?
2. Суть аналогії між формою LET та λ - викликом .
3. Пояснити роботу універсальної форми COND явної та неявної .
4. Дати визначення функціоналу.
5. Принципова відмінність між функціями APPLY та FUNCALL .
6. Дати означення макросу та пояснити його відміниться від визначення функції.
7. Основні арифметичні функції MU-LISP.
8. Які функції дозволяють опрацьовувати властивості списків?
9. Характеристика форм PROGRAM1, PROGRAM2, PROGRAMN .
10. Перерахувати основні керуючі структури MU-LISP.
11. Визначити функціонал FUNCALL через функціонал APPLY.

6.5. Мова програмування AUTO-LISP та її основні функції

6.5.1. Загальні відомості та особливості мови

Мова AUTO-LISP є модифікацією однієї з версій мов LISP з доповненням засобів AUTOCAD. Наявність цього інтерпретатора мови LISP значно розширює можливості системи AUTOCAD та забезпечує можливість розробки спеціальних систем конструювання, які налаштовані на специфіку конкретного виробництва. Користуватися мовою AUTO-LISP можна у всіх режимах. Повідомлення про те, що ми є в середовищі AUTO-LISP, є введення відкритої круглої дужки “(”. Фірма AUTODESK, яка є розробником цієї мови, визначає її такі переваги:

1) LISP добре працює з наборами різнорідних об’єктів, які утворюють групи різного розміру, з якими працює AUTOCAD .

2) Інтерпретатор LISPу ідеально пристосований до проектування.

3) LISP є однією з найлегших мов програмування.

4) LISP – одна з головних мов у галузі штучного інтелекту.

5) Інтерпретатор мови займає набагато менше пам’яті завдяки простоті синтаксису мови.

Ця мова дає змогу:

1) Виконувати оперативні обчислення в процесі конструювання, тобто використовуватися як калькулятор без виходу з режиму проектування.

2) Здійснювати блочні обчислення з привласненням графічних імен та використовувати їх надалі, звертаючись до них по імені.

3) Використовувати в командах AUTOCAD змінні, вислови та імена графічних об'єктів.

4) Створювати нові функції та команди в середовищі AUTOCADу.

5) Проектувати предметно-орієнтоване середовище для конкретного виробництва.

В 1992 р. цією фірмою AUTODESK запроваджений компілятор AUTO-LISP, який дав значний виграш при експлуатації мови, особливо в швидкості (5-10 разів).

Відмінності AUTO-LISP від інших версій в тому, що він не використовує багатьох властивостей LISP, а саме:

- символ AUTO-LISP має тільки ім'я та значення; відсутні властивості та визначення окремо від значення – функція визначається як значення;
- відсутнє визначення макросу.

Ці особливості дещо звужують можливості мови, але зберігають основні механізми функціональної мови програмування.

Запуск програми AUTO-LISP можна здійснювати в усіх режимах AUTOCAD – з графічного редактора при введенні імені програм з клавіатури, з файлу сценарію .crt, з файлу меню .hnu та файлу LISP –програми .lsp.

Файл acad.lsp завантажується та запускається автоматично при вході в графічний редактор. Інші файли, наприклад: user.lsp потребують завантаження за допомогою окремої функції завантаження load, наприклад: LOAD "USER".

При цьому розширення .lsp можна не вказувати.

AUTO-LISP підтримує такі типи даних:

INT – цілі числа;

REAL – дійсні числа;

STR – рядкові константи;

SUM – символи;

LIST – списки;

SUBR – вбудовані функції;

FILE – дескриптори файлів;

ENAME – імена примітивів;

PICKSET – набори примітивів.Коментар починається з символу ";" і все, що записано після нього до кінця рядка не опрацьовується інтерпретатором LISPу.

Концептуальним у мові LISP як мові функціонального програмування, а також AUTO-LISPі є поняття вислову як списку, першим елементом якого є

функція (якщо перед списком не стоїть спеціальна функція), а інші елементи є аргументами цієї функції. Функцією, що відміняє сприйняття списку у вигляді вислову є функція guote. Результатом, який повертається, є сам список. Функція guote має альтернативне визначення у вигляді знаку апострофа, який ставиться перед списком, наприклад ‘(+5 2) ідентичне (guote +5 2)).

6.5.2. Основні групи функцій. Характеристика арифметичних функцій

AUTO-LISP має широкий спектр вбудованих функцій. Ці функції AUTO-LISPу можна розбити на такі групи:

1) привласнення;
2) арифметичні;
3) порівняння;
4) логічні (булеві);
5) алгебраїчні та арифметичні;
6) геометричні;
7) дії з рядковими змінними;
8) перетворення типів даних;
9) дії з файлами;
10) вводу-виводу;
11) дії зі списками;
12) циклу та умови;
13) запуску команд AUTOCADу;
14) визначення та виклик функції;
15) визначення нових команд AUTOCADу;
16) доступ до системних змінних AUTOCADу;
17) доступ до графічного та текстового екрана;
18) доступ до імен та даних примітивів;
19) маніпуляції з наборами примітивів;
20) спеціальні функції.

До функцій привласнення відносять функції SET і SETQ, дія яких аналогічна, як і в S-LIPSI. В AUTO-LISP існує ще спеціальна функція **EVAL**, яка повертає значення вислову, тим самим дає змогу переглянути значення змінної.

До арифметичних функцій AUTO-LISP належать такі :

(+ <число> <число>...) – повертає суму всіх чисел;

(- <число> <число>…) – повертає результат послідовно виконаного віднімання всіх чисел зліва направо;

(* <число> <число>...) – повертає добуток всіх чисел;

(/ <число> <число>...) – повертає результат послідовного ділення всіх чисел зліва направо;

Особливістю функції **SETQ** в AUTO-LISPі є те, що можна задавати довільну кількість символів та висловів, наприклад:

(SETQ <S1> <V2> <S2> <V2>…)

привласнює символу S1 значення V2, символу S2 значення V2 і так далі, а як результат буде видано значення останнього вислову.

6.5.3. Логічні функції та функції порівняння

Логічні функції для роботи зі списками, так само, як і функції порівняння, повертають символ T – якщо результатом є істинне твердження або NIL – якщо результатом є хибне твердження.

До логічних функцій належать:

(AND <вислів>...) – логічне І над списками, тобто повертає T, якщо жоден з висловів немає значення NIL;

(OR <вислів>...) – логічне АБО над списками, тобто повертає T, якщо хоча б один з виразів не має значення NIL;

(NOT <вислів>...) – логічне НІ, тобто повертає T, якщо вислів має значення NIL;

До функцій порівняння належать такі:

(= <атом 1> <атом 2>...) – істина, якщо всі атоми еквівалентні;

(/= <атом 1> <атом 2>...) – істина, якщо два атоми не є еквівалентні;

(< <атом 1> <атом 2>...) – істина, якщо кожен попередній елемент є менший від наступного

(<= <атом 1> <атом 2>...) – істина, якщо кожен попередній елемент є менший або дорівнює наступному;

(> <атом 1> <атом 2>...) – істина, якщо кожен попередній елемент є більший від наступного;

(>= <атом 1> <атом 2>...) – істина, якщо кожен попередній елемент є більший або дорівнює наступному;

(MINUSP <число>) – істина, якщо число є від'ємне;

(NULL <атом >) – істина, якщо атом має значення NIL;

(NOMBERP <атом >) – істина, якщо атом є повний;

(EQ <вислів 1> <вислів 2>) – істина, якщо два вислови істинні;

(EQUWL <вислів 1> <вислів 2><допуск>) – істина, якщо вислови повертають результати в межах заданого допуску;

(ZEROP <число>) – повертає T, якщо число дорівнює нулю;

6.5.4. Алгебраїчні, тригонометричні та геометричні функції

До цього класу функцій належать такі:

(ABS<число>) – повертає ATAN < абсолютне значення числа;

(**число1> <число2>) –** повертає арктангенс числа1 в радіанах, якщо число2 не задане. За наявності числа2 повертає арктангенс результату ділення числа1 на число2;

(COS <кут>) – повертає косинус кута, який заданий в радіанах;

(EXP <число>) – повертає в степені число;

(SIN <кут>) – повертає синус кута, який заданий в радіанах;

(LOG <число>) – повертає натуральний логарифм числа;

(EXPT <основа> <степінь>) – повертає основу, піднесену до степеня;

(SQRT <число>) – повертає квадратичний корінь числа;

(MIN <число1> <число2> …) – повертає мінімальне значення з певного ряду чисел;

(MAX <число1> <число2> …) – повертає максимальне значення з певного ряду чисел;

(**REM <число1> <число2> …) –** повертає залишок від послідовного ділення заданих чисел;

(GCS <ціле1> <ціле2>…) – повертає найбільший спільний дільник;

(**ANGLE <точка1> <точка2>) –** повертає кут в радіанах між віссю X та заданим вектором, що виміряний проти годинникової стрілки в поточній системі координат <T1> < користувача;

(**DISTANSE T1 T2>) –** повертає відстань між двома точками;

(INTERS <T1> <T2> <T3> <T4>) – повертає точку перетину двох відрізків T1T2 та T3T4 або NIL, якщо вони не перетинаються;

(POLAR <точка> <кут> <відстань>) – повертає точку на заданій відстані від заданої точки під заданим кутом в радіанах до осі X у поточній системі координат користувача.

6.5.5. Функції перетворення типів змінних та роботи з текстовими змінними

Для перетворення типів змінних в AUTO-LISP передбачені такі функції:

(**FIX<число>) –** перетворює число в ціле;

(**FLOAT<число>)** – перетворює число в дійсне;

(**ITOA<число>)** – перетворює ціле в текст;

(**RTOS<число><подання><точність>)** – перетворює число в текст у вигляді**:**

<подання>	формат
1	науковий
2	десятковий
3	інженерний (фути, дюйми)
4	архітектурний (фути та дробові дюйми)
5	дробові частини

Змінні подання та точність відповідають системним змінним LANITS та LUPREC.

(CHR<ціле>) – перетворює ціле в текст згідно коду ASCII.

Для перетворення текстових змінних передбачені функції:

(ASCII<текстові константи>) – перетворює текст в ASCII-символьний код у вигляді цілого числа;

(ATOF<текстові константи>) – перетворює текст в дійсне число;

(STRCASE<текст>) – перетворює всі символи тексту в нижній регістр;

(STRCAT<текст1><текст2>...) – приєднує тексти;

(STRLEN<текст1>) – підраховує кількість символів у тексті;

(SUBSTR<N><M>) – повертає текст з символу за номером N і довжиною M. Якщо M відсутнє, то повертається вся частина тексту після номера N.

6.5.6. Функції вводу-виводу та роботи з файлами

Всі функції вводу-виводу можна розбити на два класи: функції вводу-виводу, що починаються символами **Get,** та функції, які працюють з файлами, клавіатурою та текстовим екраном. Слід зауважити, що при роботі з файлами треба задавати не ім'я файла, а його дескриптор. Тому для роботи з файлом його необхідно відкрити функцією **OPEN**, а потім визначити дескриптор файла функцією **SETQ.** Наприклад,

(SETQ f1(OPEN "f1.txt")).

Після закінчення роботи файл необхідно закрити (CLOSE f1). Пошук файла здійснюється функцією (FINDFILE <ім'я файлу>), а завантаження – (LOAD <ім'я файлу>), при цьому аналізується синтаксис файла.

Основні функції вводу-виводу такі:

(**GETINT <підказка>)** – забезпечує паузу перед введенням цілого числа і видає значення підказки у вигляді командного рядка;

(**GETREAL <підказка>)** – аналогічна getint для дійсних чисел;

(**GETPOINT <точка><підказка>)** – створюється пауза для вводу точки будь-яким засобом AUTOCADу; якщо вказана підказка, то вона виводиться в командному рядку; якщо задана точка, то висвітлюється нитка від поточного положення курсора до точки;

(**READ_CHAR <дескриптор файлу>)** – зчитує символ з клавіатури при відсутності дескриптора файла або з файла, в противному випадку повертає код ASII;

(READ_LINE<DF>) – зчитує рядок з клавіатури або файла та повертає текст;

(**WRITE_CHAR<ціле> <DF>)** – зчитує символ ASCII, код якого відповідає цілому, у файл або на екран і повертає ціле;

(WRITE_LINE<текст><DF>) – записує текст у файл або на текстовий екран і повертає його;

(PRINT1<вислів><DF>) – виводить вислів на екран та в файл;

(PRINT<вислів><DF>) – друк відбувається з нового рядка з наступною прогалиною;

(VER) – виводить повідомлення про версію AUTO-LISPу.

6.5.7. Функції опрацювання списків

Для опрацювання списків передбачені функції:

(**APPEND** <**список 1**> <**список 2**> …) – повертає єдиний список, в якому перераховані всі елементи заданих списків;

(**ASCOC** <**e**> <**структурний список**>) – переглядає структурний список та повертає підсписок, перший елемент якого збігається з **e**, якщо такого підсписку не існує, повертає **NIL**.

(**ATOM** <**вислів**>) – повертає **NIL**, якщо вислів є списком, інакше **T**;

(**CAR** <**список**>) – повертає перший елемент списку;

(**CDR** <**список**>) – повертає повний список без першого елементу;

(**CADR** <**список**>) – аналогічно (**CAR** (**CAR**<**список**>));

(**CDAR** <**список**>) – аналогічно (**CDR**(**CAR** <**список**>));

Поєднання **CAR** і **CDR** можливе до четвертого рівня вкладення.

(**CONS** <**e**> <**s**>) – додає елемент **e** до списку **s** і видає повний список;

(**LAST** <**список**>) – видає останній елемент списку;

(**LENGTH** <**список**>) – повертає кількість елементів списку;

(**LIST** <**вислів 1**> <**вислів 2**> ...) – повертає список, елементами якого є задані вислови;

(**LISTP** <**e**>) – повертає **T**, якщо **e** – список, інакше **NIL**;

(**MEMBER** <вислів> <список>) – переглядає список до вислову та повертає частину <списку>, починаючи з заданого вислову;

(**NTH** <**N**> <**S**>) – повертає **N** - ий елемент списку **S;**

(**REVERSE** <**S**>) – виданий елемент списку **S** у зворотній послідовності, починаючи з останнього;

(**SUBST** <новий елемент> <старий елемент> <список>) – замінює старий елемент на новий у всьому списку і повертає модифікований список; за відсутності старих елементів список повертається незмінним.

6.5.8. Функції циклу та умови

Для організації циклів та виконання умов передбачені функції:

(FOREACH <ім'я> <список> < вислів> <вислів>...) – обчислює всі задані <вислови> стільки разів, скільки елементів у <списку>, присвоюючи змінним висловів, імена яких збігаються з заданим аргументом <ім"я>, значення елементів списку; як результат повертається останнє значення вислову. Наприклад:

(Forech y (a b c) (sety y(1+y))) еквівалентне таким виразам :

(setQ a (1+a))

(setQ b (1+b))

(setQ c (1+c))

Відмінність лише в тому, що forech поверне останнє значення вислову (1+c).

(MAPCAR <функція> <список1>... <список N>) – виконує функцію над елементами списків як аргументами та повертає список результатів; кількість списків повинна відповідати кількості аргументів <функцій>;

(CONS (<тест1> <вислів1>) (<тест2> <вислів2>)...) – функція вибору та виконання вислову з умови: переглядаються усі тести, що є першими елементами списків; якщо тест NIL, то інші елементи цього списку ігноруються; перегляд здійснюється доти, доки не поверне значення =NIL;

виконуються всі вислови цього списку і як результат видається значення останнього вислову;

(**IF<тест-вислів> <вислів-тоді> <вислів-інакше>**) – функція виконання вислову з умови – якщо <тест-вислів> не NIL, то виконується <вислів-тоді>, в протилежному випадку виконується <вислів-інакше>;

(**REPEAT <N> <вислів1> <вислів2>...**) – кожен вислів послідовно обчислюється N-разів та повертається результат останнього вислову.

(**WHILE <тест-вислів> <вислів1> <вислів2>...**) – послідовно виконуються всі вислови доти, доки <тест-вислів> не поверне NIL/.

6.5.9. Функції визначення, виклику та трасування функцій

Для означення нових функцій в AUTO-LISPі існують такі функції:

(DEFUN <ім'я функції> <список аргументів> <вислів1> <вислів2>...) – визначає нову функцію з іменем, що стоїть за символом DEFUN; список аргументів косою рискою розділяється на глобальні та локальні; після виходу з неї локальні параметри дорівнюють NIL; аргументи можуть бути і відсутні; функція повертає <ім'я функції>.

Якщо це ім'я зустрілось в наступних висловах, як перший елемент списку, то відбувається визначення аргументів, зв'язування їх в список аргументів, передача цього списку у функцію та вирахування всіх висловів. Наприклад, задаємо функцію

(DEFUN f (x y z) (f x y z)),

яка при виклику

(f 1 2 3)

як результат видає 6.

Функція DEFUN завантажується автоматично, якщо вона знаходиться у файлі ACAD.LSP, в іншому випадку для завантаження використовується функція

(**LOAD <ім'я файлу>).**

Інші вбудовані функції виклику функцій:

(LAMBDA <аргумент> <вислів>…) – непоіменована функція, на яку немає посилань з інших місць програми;

(APPLY <функція> <список>) – функція виконання функцій, що задані як текстова константа; аргументи визначаються у вигляді списку;

(MAPCAR <функція> <список1> <список2>...) – виконує функцію над елементами списків та повертає список результатів; кількість списків

дорівнює кількості аргументів функції;

(TRACE <функція>…) – включає трасування заданих функцій – використовується при відлагоджені;

(UNTRACE <функція>…) – виключає трасування заданих функцій.

6.5.10. Спеціальні функції AUTO-LISP

Для доступу до системних змінних передбачені функції:

(**GETVARE <ім'я системної змінної>)** – повертає поточне значення системної змінної.

(SETVARE < ім'я системної змінної >< значення>) – присвоює поточне значення системній змінній.

Доступ до графічного та текстового екрана забезпечують функції:

(GRAPHSHR) – відновлення графічного екрана;

(TEXTSCR) – перехід до текстового екрана;

(REDRAW) – регенерує креслення.

При завантаженні креслення AUTOCAD заповнює графічну базу даних і кожному графічному примітиву присвоює своє ім'я у вигляді 16-значної константи. У кожному сеансі роботи ці імена різні. Крім того, кожен примітив має свою мітку, яка не змінюється від сеансу до сеансу. Набір імен примітивів об'єднується в одну групу під одним іменем і з ним можна працювати за допомогою спеціальних функцій, що починаються з символів SS. Для доступу до імен примітивів передбачені функції:

(**ENTLAST**) – повертає ім'я останнього примітива, що не вилучений;

(**ENTNEXT < ім'я примітиву>**) – повертає ім'я 1-го невилученого примітива за відсутності аргументів, – за наявності наступного за ним;

(**ENTSEL**) – повертає список, що складається з імені примітива та координати точки, за допомогою якої примітив був вказаний;

(**HADENT "мітка**") – повертає ім'я примітива, що зв'язаний з цією міткою;

(**ENTDEL<примітив>) –** вилучає або відновлює після вилучення заданий примітив;

6.5.11. Контрольні питання та вправи

1. Класифікація груп функцій мови AUTO-LISP.
2. Навести опис математичних та арифметичних функцій .
3. Які функції роботи з файлами є в середовищі AUTO-LISP?

4. Функції організації циклів та умовних переходів, які є вбудованими в AUTO-LISP.

5. Описати найвживаніші спеціальні функції AUTO-LISP.

6. Як в середовищі AUTO-LISP є функції виклику функцій і яке їх призначення?

7. Перерахувати функції, які застосовуються в середовищі AUTO-LISP для опрацювання списків.

8. До якого рівня глибини можна вкладати примітивні функції **CAR** і **CDR** , застосовуючи скорочену форму запису?

6.6. Тенденції розвитку мов та методів функціонального програмування

6.6.1. Доцільність освоєння нових мов програмування

Будь-яку систему позначень для опису алгоритмів і структури даних можна назвати мовою програмування, хоча, як правило, вимагається реалізація мови на комп'ютері.

Сьогодні розроблено сотні різних мов програмування. Ще в 1969 р. Саммет [1] наводить список з 120 мов, які доволі широко застосовуються. Це приголомшлива кількість мов програмування протирічить тому, що більшість програмістів в своїй практиці використовує декілька мов програмування, а значна їх частина – одну або дві мови програмування. Виникає питання про доцільність освоєння різних мов програмування, якщо навряд чи буде можливість їх реалізації. Тим не менше, якщо не обмежуватися поверховим ознайомленням з мовою, а мати глибокі уявлення про поняття, які лежать в основі конструювання програм, то можна навести, як мінімум, п'ять тверджень на користь доцільності освоєння різних мов програмування:

1. Внаслідок вивчення різних мов програмування покращується розуміння конкретної мови, її понять та основних методів і прийомів, що в ній використовуються. Типовим прикладом може бути рекурсія. При правильному її використанні можна отримати елегантну ефективну програму, а застосування її до простого алгоритму могло б привести до астрономічного збільшення часових затрат. З іншого боку, недоступність використання рекурсії в таких мовах, як Фортран, Кобол та розуміння основних принципів і

методів реалізації рекурсії може внести ясність в певні обмеження мови, які, на перший погляд є надуманими.

2. Значення мов програмування розширює запас корисних конструкцій для програміста і сприяє розвитку мислення. Працюючи з структурами даних однієї мови, виробляють і відповідну структуру мислення. Вивчаючи конструкції інших мов та методи їх реалізації, розширюють тезаурус програміста.

3. Знання великої кількості мов програмування дає змогу обґрунтовано вибирати певну мову програмування для розв'язання конкретної задачі.

4. Освоєння нової мови програмування, як і природної мови людського спілкування, завжди легше, якщо відомими є декілька мов.

5. Знання принципів побудови різноманітних мов програмування полегшує розробку нової мови програмування.

Конструювання сучасних мов програмування сьогодні далеке від досконалості. Кожна з відомих мов має свої недоліки та переваги. Для визначення доцільності використання тієї чи іншої мови слід виходити з таких міркувань:

1. Ясність, простота і узгодженість понять мови. Очевидно, слід уникати тонких і каверзних обмежень мови. Вона не повинна бути також двозначною. Семантична ясність мови – це те, що визначає її цінність.

2. Ясність структури програми. Ця вимога забезпечується синтаксичною ясністю програм, записаних цією мовою. Мова повинна бути такою, щоб конструкції, які відрізняються семантично, відрізнялись і синтаксичним записом.

Дуже важливо, щоб структура програми відображала структуру самого алгоритму, що дозволяється при розробці програми дотримання принципів структурного програмування, ієрархічного конструювання програм – зверху вниз. При такому підході структура програми легкозрозуміла, доступна для діагностики та модифікації.

3. Природність у використанні. Мова повинна забезпечувати при розв'язанні задачі найвдаліші структури даних, операції, керуючі структури і легко зрозумілий синтаксис. Все це істотно спрощує створення програмних засобів у заданій галузі.

4. Легкість розширення. Програмні засоби, які створюються на цій мові, можна розглядати як розширення мови. В принципі більшість мов програмування дає програмісту механізми для створення підпрограм. Тим не

менше, властивості самої мови можуть полегшити або ускладнити їх використання.

5. Зовнішнє забезпечення. Це один із аспектів, який впливає на ефективність використання мови. За наявності потужних засобів тестування, редагування, збереження, модифікації програмних засобів можна зробити слабку мову зручнішою для експлуатації, ніж потужна мова без відповідного технічного забезпечення.

6. Ефективність створення, тестування, трансляції, виконання, модифікації та використання програм.

6.6.2. Переваги функціональних мов над процедурними

Більшість сучасних мов програмування є універсальними, оскільки дають змогу записати будь-який алгоритм цією мовою, якщо не накладати обмежень на час виконання програми та місткість пам'яті. Якщо хто-небудь запропонує нову мову програмування, то вона, очевидно, буде універсальною, якщо ігнорувати обмеження на пам'ять або час. Порівнюючи різні мови програмування, слід виходити не з кількісного співвідношення того, що вони дозволяють зробити, а з якісних відмінностей, що визначають елегантність (короткість і наочність), легкість (прозорість) та ефективність (швидкодія та технічні засоби) програмування на них. Це порівняння слід здійснювати в контексті конкретної сфери застосування.

Традиційні (алгоритмічні) мови програмування є доволі об'ємні і громіздкі, бо не дають змоги:

- максимально використовувати можливості сучасної комп'ютерної техніки для забезпечення ефективності програмних засобів;
- ясно і наочно відобразити алгоритми програми, щоб забезпечити легкість перевірки та модифікації останніх.

Строго функціональні мови програмування є доволі простими, і лише цим забезпечують достатньо високий ступінь виразності програм порівняно з традиційними мовами. Ряд функціональних програм можуть ефективно працювати на сучасних комп'ютерах, тим не менше не так ефективно, як відповідні програми з оператором присвоєння. Це пов'язано зі структурою архітектури сучасних комп'ютерів. Окрім того, вибір дещо іншої структури представлення даних, ніж це прийнято в інструментальному Ліспі, забезпечує як більшу ясність представлення програм, так і підвищення їх ефективності з використанням сучасних комп'ютерів старої архітектури.

З одного боку, сучасні мови програмування повинні ефективно використовувати сучасні машини, а з іншого – ясно виражати програмні алгоритми, щоб полегшити перевірку останніх. Строго функційна мова, будучи простою за своєю структурою, демонструє вищий ступінь виразності порівняно з традиційними мовами, де існує оператор присвоєння. Це зв'язано, великою мірою, зі способом вибору структур даних. Дещо інший їх вибір можна забезпечити підвищенням ефективності функціональних програм і на сучасних комп'ютерах. Зокрема, розглянемо проблему використання функцій зі складовим результатом, оскільки тут є важливий елегантний розв'язок.

Одна з фундаментальних властивостей мови програмування, що дає можливість ясно описати обчислення на цій мові – простота семантики мови. Велика перевага функціональної мови в тому, що тут є кілька основних понять, кожне з яких має просту семантику. Зокрема, семантика нашої мови розуміється в термінах значень, які мають вирази, але аж ніяк не в термінах дій і послідовності їх використання. Але з практичного погляду було б справедливіше зробити висновок, що строго функційна мова є надзвичайно елементарною і деякі її розширення значно б підвищили ефективність і ясність деяких класів вирахувань. Очевидно, необхідно розрізняти суто синтаксичні розширення і розширення, які вимагають зміни семантики мови. Зміна семантики вимагає обережності, оскільки це ускладнює розуміння (ясність) вже відлагоджених функціональних програм. Те, що синтаксичне розширення дає змогу підвищити ясність виразів, продемонструємо на прикладі функцій зі складовим результатом.

6.6.3. Розширення строго функціональної мови

Розширимо нашу строго функціональну мову, даючи змогу функціям видавати складові результати. Будемо вважати правильними виразами списки правильних виразів, які поміщені в кутові дужки. Так, вираз < e1, e2, ..., ek > є правильним, якщо правильними є вирази e1, e2, ..., ek. Друге розширення те, що у лівій частині локальних форм дозволимо використовувати не лише прості змінні, але й список ідентифікаторів, які поміщені в квадратні дужки. Спосіб використання цього комбінованого розширення демонструє приклад

```
{ НЕХАЙ f=λ(x,y) <x зал y, x діл y>
        { НЕХАЙ <r, d>=f(u, v)}}
```

Тобто тут список значень **f** точно буде скопійований в список змінних і внутрішній блок зв'яже з **r** значення **u зал v**, **u** з **d** – значення **u діл v**. Це суто синтаксичне розширення, оскільки цього результату ми могли б досягнути, використовуючи операцію **cons**:

{ НЕХАЙ f=λ(x,y) cons(x зал y, cons(x ціл y), NIL
{ НЕХАЙ S=f(u, v)
{ НЕХАЙ r=car(S) і d=car(cdr(S))}}}.

Для демонстрації того, наскільки зросла ясність представлення виразів при такому розширенні, розглянемо функціональне визначення порозрядного складання з переносом одиниці. Нехай маємо функцію, яка підсумовує три числа по модулю m:

сум1(a, в, c)=<(a+b+c) діл m, <(a+в+c) зал m>

Тут **a, в** – цифри, що підсумовуються і не перевищують **m, c** – одиниця переносу. Результат функції **сум1** – це пара значень, де на першому місці цифри переносу з цього розряду і цифри суми. Тепер побудуємо функцію **сум N(a, в)**, яка в якості аргументів видає як результат список цифр суми і кінцеву цифру переносу. Визначення є очевидним і має вигляд:

сумN(x,y)≡якщо cdr(x)=NIL то cdr(y))
інакше сум1(car(x), car(y)), 0)
інакше {нехай <c_i, S>=сумN(cdr(x), cdr(y))
{нехай <c_0, S>=сум1(car(x), car(y), c_i)
<c_0, cons(d,S)>}}

Якщо в списках **x** і **y** лише по одній цифрі, то можна використати **сум1** з цифрою переносу, що дорівнює 0. Інакше використовується **сумN(x)**, щоб скинути всі цифри, крім старших, і отримати цифри переносу $\mathbf{c_i}$ і суму **S**. Потім використовується **сум1,** щоб одержати суму цифр найстаршого розряду і цифру переносу в старший розряд $\mathbf{c_i}$ і отримати цифру переносу зі старшого розряду $\mathbf{c_0}$. Старша цифра суми є **d**, тому весь список складання цифр **x** і **y** отримується шляхом використання операції **cons** до **d** і **S**.

Безсумнівно, можна написати цю програму, не використовуючи запропонованого розширення мови. Другою альтернативою може бути

реалізація двох функцій, одна з яких вираховує переноси, а друга – суми від списків задання цифр по модулю m. Хоч ясність такого представлення, можливо не зменшується, але ефективність значно погіршиться, оскільки потрібно буде багатократно повторити вирахування одних і тих самих переносів у молодших розрядах для кожної з підфункцій.

Розглянемо третій варіант, коли не потрібно розширювати мову, який є доволі конкурентоздатний в розумінні ясності представлення.

Кожну з функцій **сум1** і **сумN** розширюємо введенням її як параметра функції **f**, яку назвемо продовженням. Так, виклик **сумN(x, y, f)** означає складання списків **x** і **y** і використання функції **f** до цього результату.

Природно, що функція продовження **f** вимагає двох параметрів, відповідно значення цифри переносу і списку цифр суми. Нове визначення матиме вигляд:

сум1(a, в, c, f)≡f((a+в+c) діл m, (a+в+c) зал m)
сума(x, y, f)≡якщо cdr(x)=NIL то сум1(car(x)), car(y), 0, f)
інакше сумN(cdr(x), cdr(y),
{λ(c_i, S) сум1(car(x), car(y), c_i,
{λ(c_0, d) f(c_0, cons(d, S))})})

Відзначимо, що фактичне продовження, яке замінює вкладені виклики **сум1** і **сумN,** записане як **λ**-вирази з формальними параметрами, які мають ті самі імена (і виконують ту саму роль), що відповідні локальні визначення в попередній програмі з кутовими дужками. Цю програму можна вважати достатньо конкурентною з попередньою за ефективністю, але вона втрачає ясність через введене продовження, а тому є менш привабливою.

6.6.4. Шляхи вдосконалення функціональних мов

Принциповою характеристикою сучасних комп'ютерів є те, що вони зберігають вирахувані значення в комірках пам'яті, час від часу замінюючи їх вміст або перезаписуючи їх. Ця властивість відображена в конструкції присвоєння, яка характерна для традиційних мов програмування. Є ще один аспект реальних машин, завдяки якому традиційні мови ефективніші, порівняно з функціональними мовами програмування. Це поняття доступу до структур даних за допомогою індексування. В зв'язку з цим функціональні програми не можуть досягнути такої самої ефективності на сучасних

комп'ютерах, як програми алгоритмічними мовами програмування. Можна вказати, принаймні, три виходи з такої ситуації:

1) при неготовності змиритися з деякою втратою ефективності (при виграші в якості і елегантності програми) відмовитися від функціональних програм;

2) розробляти методи організації функціональних програм так, щоб їх виразність і ясність поєднувалися з ефективністю, співмірною з програмами, написаними традиційними мовах програмування;

3) перебудувати структуру сучасних комп'ютерів так, щоб вони відповідали цілям інтерпретації функціональних програм.

З певного погляду кожен з цих підходів має право на існування.

Операція присвоєння є тісно пов'язана зі зв'язуванням значення зі змінною у функціональній мові. Можна показати, що будь-яка програма, де є операція присвоєння змінній певного значення, може бути трансформована у функціональну програму. Тоді така функційна програма з певними обмеженнями на її структуру може бути скомпільована в програму з присвоєнням. Отже, можна стверджувати, що функціональні мови не виключають можливості ефективного використання сучасних комп'ютерів. Але це лише частина проблеми, оскільки операція присвоєння значення елемента масиву виду

a[i]:=1

a[i+1]:=a[i]+1

істотно відрізняється від операції присвоєння значення простій змінній. У строго функціональній мові відсутнє поняття вирахування послідовністю дій і з масивами поводяться як з цілими масивами значень. Основними операціями над масивами є операція індексування для отримання значення компоненти і конструювання нового значення масиву із старого. Якщо **a** – значення масиву, **i** – його індекс, **x** – значення, то запис

обновити (a, i, x)

означає збереження значення масиву **a** зі змінною його **i** - вої компоненти на значення **x**. А вираз

обновити (a, i, a[j]), j, a[i])

означає заміну місцями значень **i**-го і **j**-го елементів масиву **a**.

Тому проблема корекції інтерпретації правильності функції **обновити** є доволі складною при врахуванні того, що одне і те саме значення масиву може мати різні імена. Операція оновлення значень масиву вимагає багатократного копіювання цих значень. Виникає запитання: чи можна вибрати таку структуру даних, щоб цілком або частково уникнути цих

копіювань? Проблеми не виникає, якщо для цих цілей використовувати операцію **cons**, оскільки структури, які є компонентами **cons** не копіюються, а зберігаються лише вказівники на них в структурі, яка є результатом **cons**. Але не хотілося б так представляти реалізацію значень масивів, оскільки втрачається основна перевага сучасних комп'ютерів – можливість індексування. Дійсно, при послідовному доступі в лінійних списках час доступу до **i**-го елемента пропорційний до кількості елементів **i**. Водночас при розташуванні масиву в комірках пам'яті одночасно доступний будь-який елемент незалежно від позиції. Можна масиви подати у вигляді бінарного дерева; час доступу до елемента пропорційний $\mathbf{ln_2 n}$. Тим не менше це є непорівнянним з одиницею часу доступу, яка необхідна при елементарному оновленні в операторі **a[i]:=x**.

Функціональні мови за ефективністю на сучасних комп'ютерах не можуть конкурувати з традиційними мовами, якщо ефективність є абсолютною вимогою. Однак програми, написані з елементним присвоєнням структурованих значень, є програмами низького рівня. Вимагають значних зусиль для їх побудови, а тому є важкими для правильної інтерпретації.

Підвищення швидкості комп'ютерів та місткості їх пам'яті на сучасному етапі робить доступними ряд важливих функціональних програм прикладного характеру. Але якісного стрибка у використанні функціональних мов можна очікувати лише з появою комп'ютерів нової архітектури, яка орієнтована не на використання технічних засобів, а на особливості самої мови, яка будується, виходячи з природних потреб користувача.

6.6.5. Контрольні питання та вправи

1. Які можливості підвищення ефективності функціональних програм?
2. Дати визначення функції порозрядного підсумування чисел за модулем **m,** не використовуючи запропонованого розширення мови.
3. Дати визначення функції **обновити (a, i, x)** в термінах базових функцій.
4. Побудувати функцію **паліндром(x)**, яка перевіряє чи послідовність символів в списку **x** є еквівалентна зворотній послідовності цих символів. Запропонувати варіанти реалізації функції на основі базових функцій та шляхом введення додаткових підфункцій. Порівняти ефективність, алгоритмічну складність та форму запису функцій у цих підходах.
5. Вказати перспективні напрямки розвитку функціонального програмування
6. Розширити S-Лісп, ввівши українську абетку та малі літери. Запропонувати варіант побудови інтерпретатора.

Розділ 7. Лабораторні роботи в середовищі Ліспу

7.1. Лабораторна робота № 7. Програмування за допомогою функцій та процедур

Мета: Співставити ефективність функціональних програм та традиційних програм мовами високого рівня.

7.1.1. Теоретичні відомості

Функція – фундаментальне поняття математики. Функцію можна означити як правило, згідно з яким кожному елементу деякого класу (області визначення) відповідає єдиний елемент іншого класу (області значень). Існує багато способів опису простих функцій: графічний, табличний, аналітичний та ін. Кожен з них має той недолік, що важко перевірити неявно задану властивість однозначної відповідності області визначення і області значень функції. Коли ж область визначення нескінченна, то й зовсім неможливо використовувати вищезгадані способи опису. Для прикладу, функція **квадрат** має областю визначення клас всіх цілих чисел, а областю значень – клас всіх невід'ємних чисел. Очевидно, зобразити цю всю відповідність неможливо. Але повністю можна визначити всю цю відповідність, написавши правило (означення):

КВАДРАТ (X) = X*X

При написанні цього правила ми використали змінну **X** для позначення будь-якого елемента з області визначення і записали правило обчислення відповідного елемента з області значень у вигляді виразу, що використовує цю змінну. Таку змінну називають параметром означення.

Визначене правило, або означення, яке наведене для функції **КВАДРАТ**, є тим стандартним способом за допомогою якого може бути подана будь-яка інша функція.

Розглянемо другий приклад. Функція м**акс**, що застосована до пари чисел, як результат, видає найбільше з них. Так, приклад

МАКС(1,3)=3, МАКС(1,7, -2,5)=1,7

Означення цієї функції найбільш доцільно задати через два параметри:

МАКС(X,Y) = X, якщо X > Y

Y, в іншому випадку

Громіздкості в означенні можна уникнути, якщо використати умовну форму **якщо....., то.....інакше,** яка зустрічається в традиційних мовах програмування:

МАКС(X,Y) = якщо X > Y то X інакше Y

Означення функції по своїй суті є програмою, яка сприймає вхідний сигнал (її аргумент) і виробляє вихідний (її результат). Щоб мати можливість конструювати більш потужні програми, необхідно вміти визначити нові функції через старі. Розглянемо, наприклад, таке означення:

НАЙБ (X,Y,Z) = МАКС(МАКС(X,Y),Z)

Очевидно, дана функція визначає максимальний з її трьох аргументів. Цей приклад ілюструє фундаментальний принцип побудови нових функцій із старих. Це метод композиції функцій. Вкладенням одного використання функції **МАКС** в інше отримуємо композицію функції **МАКС** з собою ж. Функції можуть бути вкладені на довільну глибину і складати довільно глибокі композиції.

Для того, щоб здійснювати програмування за допомогою функцій, необхідно мати можливість визначати множину базових або основних функцій і використовуючи принцип композиції визначати нові функції у термінах старих функцій. Таким чином, можна побудувати цілу бібліотеку функцій.

У всіх сучасних мовах програмування високого рівня є конструкції, які називаються процедурою або підпрограмою, поняття якої базується значною мірою на ідеї функції. Раніше введена функція **МАКС** може бути записана на Фортрані, ПЛ/1 чи Паскалі, реалізуючи строге означення функції. Але в цих мовах може бути визначений лише обмежений клас функцій, а тому в практичних реалізаціях використовуються процедури, які не видають результат, а присвоюють його значення одному із своїх параметрів. Це приводить до програм, як· є важкими до розуміння. Строго функціональна мова не допускає побічних ефектів. Програміст визначає функції, які вираховують значення, що єдиним чином визначаються через значення аргументів.

7.1.2. Послідовність виконання роботи

1. Повторити з курсу лекцій теоретичний матеріал, пов'язаний з програмуванням за допомогою функцій та процедур.

2. Згідно з варіантом записати означення функції.

3. Записати відповідну процедуру та підпрограму мовами високого рівня – Паскалем та Фортраном.

4. Порівняти ефективність програм.

5. Оформити звіт про роботу.

7.1.3. Контрольні питання

1. Суть програмування за допомогою функцій.

2. Що таке строго функціональна мова програмування?

3. Основні переваги функціонального програмування перед традиційними мовами високого рівня.

4. Що таке композиція функцій?

7.2. Лабораторна робота № 8. Побудова елементарних функцій

Мета: Закріпити знання про представлення функціональних програм у вигляді S-виразів та набути навиків введення простих функціональних програм у комп'ютер.

7.2.1. Теоретичні відомості

Областю визначення для функціональних програм є клас символьних виразів, який введений Маккарті в мову Лісп і носить назву S-виразів. Кожен ·з наведених рядків містить S-вираз:

(ПЕТРО 33 РОКИ)
((ІВАН 30) (МАРІЯ 10) (ОКСАНА 20) (КІМНАТА МАЄ (СВІТЛІ ВІКНА)))

Загальна риса S-виразів – наявність дужок. Якщо змінити положення або кількість дужок, то, зміниться і структура і зміст S-виразу. S-вирази складені з елементів, які мають назву атомів. Атоми бувають двох типів – символьні і числові.

Символьні атоми, як правило, є послідовністю букв, хоча можуть містити й інші символи, включаючи і цифри, але обов'язково повинні містити хоча б один символ, відмінний від цифри. Числові атоми є послідовністю цифр, можливо зі знаком. Загальні правила побудови S-виразів можна дати в такому означенні:

1) атом є S-вираз;

2) послідовність S-виразів, поміщена в дужки є S-виразом

(списком).

У форматі S-виразів можна представляти будь-які дані. Безсумнівно, існує багато різних способів представлення одних і тих же даних. Конкретний вибір даних може бути придатний для одних цілей, але цілком непридатний для інших.

Для оперування з S-виразами введемо деякі примітивні функції, які здійснюють розчленування цих S-виразів. У термінах цих примітивних функцій можна буде визначати нові функції. Цим примітивним функціям дані їх традиційні імена мови Лісп, хоча на перший погляд і вони є незвичними. Перша функція, яка необхідна для розчленування S-виразу дозволяє вибрати перший елемент із заданого списку. Це функція **car**. її результат – найперший (крайній лівий) елемент в списку. Вибраний елемент в результаті застосування функції **car** може бути атомом або списком. Функція **car** для атому не визначена.

Друга примітивна функція, яка є доповненням до функції car, це функція **cdr**. Функція **cdr** від списку є список, який отримується із вихідного шляхом відкидання його першого члена. Для атому функція **cdr** не визначена. Якщо вихідний список складається з одного атома, то значенням функції **cdr** є спеціальний атом НІЛ /NIL/, який відповідає пустому списку. Результат застосування цих функцій до S-виразів наведено нижче:

X	**car**(X)	**cdr**(X)
(A B)	A	(B)
(A B C)	A	(B C)
((A B)(C D))	(A B)	((C D))

За допомогою цих функцій можна вибрати будь-який елемент, знаючи, що він є в списку. До елементарних функцій належить і примітивний конструктор – функція **cons**. Ця функція в якості аргументів бере два S-вирази і з'єднує їх у єдиний S-вираз таким чином, щоб компоненти цього S-виразу отримувалися застосуванням до результату функцій **car** і **cdr**. Другий аргумент функції **cons** повинен бути списком або спеціальним атомом НІЛ.

Результатом **cons** тоді буде список, який отримується шляхом включення першого аргументу в якості першого елементу цього списку. Результат застосування функції **cons** до двох аргументів наведено нижче.

X	Y	**cons**(X,Y)
A	(B C)	(A B C)
(A B)	(C D)	((A B)C D)
(A B)	((C D))	((A B) (C D))
A	NIL	(A

Якщо Y є список довжини n, то **cons** (X,Y) буде списком довжини n+1 незалежно від значення X. Тому зручно і загальноприйнято атом НІЛ розглядати як список нульової довжини, або пустий список.

Для опрацювання списків різної структури необхідно:

а) розпізнавати чи є значення S-виразу атомом або списком;

б) встановлювати рівність атомів.

Цим цілям служить функція **атом.** Функція **атом**(X) називається предикатом, оскільки вона видає значення істина(T) або фальш (NIL). Функція **атом**(X) приймає значення T тільки в тому випадку, якщо X є числовим або символьним атомом. Ця функція не розрізняє числових і символьних атомів.

Для порівняння двох атомів використовується предикат **рівно** (X, Y). Якщо обидві порівнювальні величини є списками, то результат функції є невизначеним. Результат **рівно** (X, Y) приймає значення T тільки в тому випадку, якщо X і Y є одним і тим же атомом, і значення NIL, якщо це різні атоми, або один із аргументів не є атомом.

Маючи базовий набір цих елементарних функцій, можна здійснювати побудову нових функцій.

7.2.2. Послідовність виконання роботи

1. Повторити з курсу лекцій теоретичні відомості про вибір бази елементарних функцій і побудову за їх допомогою нових функцій.

2. У відповідності з варіантом побудувати функцію на основ заданої бази функцій.

3. Перевірити працездатність функції, ввівши її в комп'ютер.

4. Оформити звіт про роботу.

7.2.3. Контрольні питання

1. Що таке S-вираз?
2. Які є типи атомів?
3. Дати характеристику елементарних селекторів.
4. Вказати взаємозв'язок між примітивними конструкторами та функціями **car** і **cdr**.
5. Дати характеристику елементарних предикатів.
6. В якому випадку результат предиката **рівно** (X, Y) є не визначений?

7.3. Лабораторна робота № 9. Побудова рекурсивних функцій

Мета: Освоєння ефективних способів побудови рекурсивних функцій.

7.3.1. Теоретичні відомості

Рекурсія – це зворотний рух, повернення. У мовах програмування рекурсія – це природний спосіб подання структур даних і програм, який широко використовується в області нечислового програмування.

Рекурсія використовується у двох випадках: для опису структур даних, які мають в якості компонент інші структури, і для опису програм, виконанню яких передує виконання їх власних копій. На перший погляд, незвично визначити деяке відношення саме через себе.

У якості першого прикладу розглянемо визначення функції **довжина** (X), яка застосовується до списку і в якості результату видає кількість елементів у ньому. Тобто слід побудувати функцію такого типу:

X	**довжина** (X)
(А В С Д)	4
(В С Д)	3
(С Д)	2
(Д)	1
NIL	0

З наведеної послідовності значень бажаної функції випливає такий алгоритм її побудови: послідовно беремо **cdr** від x і підраховуємо скільки раз необхідно це зробити, поки не дійдемо до пустого списку, тобто спеціального атома НІЛ. Але перш, ніж приступати до побудови функції

слід виокремити два випадки:

1) вихідний список x = NIL,

2) x ≠ NIL.

Оскільки можна взяти **cdr** від x і це буде списком (можливо пустим), то можна застосовувати функцію, яка буде визначатися рекурсивно. Гарантією завершення рекурсії є те, що з кожним рекурсивним викликом список стає на один елемент коротшим, а тому через скінчене число кроків прийдемо до пустого списку. Алгоритм побудови функції можна записати у вигляді:

випадок (1): x = НІЛ довжина(x) = 0

випадок (2): x ≠ НІЛ нехай довжина(cdr(x))=n

тоді довжина(x) =n+1

Перший випадок x = НІЛ є тривіальним. У випадку x ≠ НІЛ передбачається наявність результату рекурсивного виклику, який застосовано до (**cdr**(x)) = n і в цьому випадку довжина(x) повинна бути на одиницю більша. Таким чином, означення функції **довжина**(x) може бути записане у вигляд·:

довжина(x) = якщо рівно(x, НІЛ) то 0 інакше довжина(cdr(x)) + 1.

Цей метод може бути використаний для побудови широкого класу функцій. У якості ще одного прикладу побудови рекурсивних функцій розглянемо функцію **з'єднати** (x, y), яка в якості аргументів бере два списки x і y і видає як результат єдиний список, що містить послідовно всі елементи списку x і всі елементи списку y:

x	Y	**з'єднати** (x, y)
(A B C)	(Д Е)	(A B C Д E)
(A B)	(C Д E)	(A B C Д E)
NIL	(A B C)	(A B C)
NIL	NIL	NIL

Оскільки обидва аргументи функції **з'єднати** є списками, то необхідно провести аналіз випадків для кожного з цих аргументів:

випадок (1) x = NIL підвипадок (1.1) y = NIL з'єднати (x, y) = NIL

підвипадок (1.2) y ≠ NIL з'єднати (x, y) = y

випадок (2) x ≠ NIL

підвипадок (2.1) y = NIL з'єднати (x, y) = x

підвипадок (2.2) y ≠ NIL нехай з'єднати(cdr(x),y) = z

тоді з'єднати(x, y) = cons (car(x),z)

Тут у всіх випадках, окрім (2.2), можна виписати означення. У випадку (2.2) використовується рекурсія, і гарантією її закінчення є скінченність списку x. По ходу такого аналізу ми на кожному кроці дивимося, що маємо в своєму розпорядженні і аналізуємо, чи можливо дати явне означення. Якщо відомо, що деякий список не NIL, то аналізуємо, чи не можна записати неявну відповідь в термінах **cdr** від цього списку. По суті, у випадку (2.2) є три кандидати на вирахування значення функції рекурсивним викликом:

з'єднати (**cdr**(x),y);

з'єднати (x, **cdr**(y));

з'єднати (**cdr**(x), **cdr**(y))

Усі вони приводять до одного і того ж результату. Обґрунтувавши використання першого кандидата на вирахування функції, можна її означення записати у вигляді:

з'єднати(x, y) = якщо рівно(x, NIL) то
якщо рівно(y, NIL) то NIL інакше y
інакше якщо рівно(y, NIL) то x інакше
cons(car(x), з'єднати(cdr(x),y))

У цьому означенні зроблений перерахунок усіх розглянутих випадків. Але її можна спростити, зауваживши, що при x = NIL функція **з'єднати**(x, y) = y незалежно від того, чи буде y рівне NIL, чи ні. Таким чином, маємо друге означення цієї функції:

з'єднатити (x, y) = якщо рівно (x, NIL) то y інакше
якщо рівно (y, NIL) то x інакше
cons (car(x), з'єднати (cdr(x), y)).

Нарешті, використовуючи той факт, що при x = NIL

з'єднати (x, y) = cons(car(x), з'єднати (cdr(x),y))

незалежно від того, чи буде y рівне NIL чи ні, отримуємо третю версію означення функції **з'єднати**:

з'єднати(x, y) = якщо рівно (x, NIL) то y інакше
cons (car(x), з'єднати (cdr(x),y))

Усі ці три означення визначають еквівалентні функції і приводять до одного і того ж результату. Кожне із означень має свої недоліки та переваги. Перше означення явно перераховує всі можливі випадки для аргументів функції, а тому найпростіше для розуміння. Третьому означенню можна віддати перевагу за короткість. Друге і третє означення мають суттєво різну ефективність. Друга версія уникає обчислень у випадку y = NIL ціною зайвих

перевірок, коли y = NIL. Третя ж версія уникає додаткових перевірок y, але вимагає здійснення рекурсивної перебудови x навіть при y = NIL.

Один із способів спрощення при побудові рекурсивних функцій – це введення додаткових функцій. Так, при визначенні функції **з'єднати** (x, y) ми могли б ввести додаткову функцію **з'єд**(x, y), яка з'єднує x і y в припущенні, що y = NIL. Тоді ми б мали означення

з'єднати (x, y) = якщо рівно (y, NIL) то x інакше

з'єд(x, y)

з'єд(x, y) = якщо рівно(x, NIL) то y інакше

cons(car(x) з'єд(cdr(x),y))

Дане означення поєднує переваги другої і третьої версій вище записаних функцій. Це загальноприйнятий прийом у функціональному програмуванню при визначенні головної функції визначати нові функції в термінах старих, тим самим встановлюючи множину функцій, які визначаються одна через одну . Таким чином, функціональна програма складається із множини функцій, одна з яких розглядається як перша причина інших. Ця функція, яку програміст буде вираховувати, є головною програмою, в той час як усі інші функції служать для неї підпрограмами.

Проблеми вибору підфункцій при розробці головної функції є споконвічною проблемою якісного структурування програми. Вдалий підбір підфункцій спеціального призначення може суттєво спростити структуру множини функцій в цілому. Для того, щоб створити добре структуровану програму, можна дати одну пораду: намагатися безперервно вдосконалювати те, що зроблено.

7.3.2. Послідовність виконання роботи

1. Повторити по курсу лекцій теоретичні відомості про рекурсивне визначення функцій та способи їх вдосконалення .

2. Для заданого варіанту проаналізувати можливі випадки зміни аргументів функції і записати алгоритм програми.

3. Записати функціональне означення програми.

4. Перевірити працездатність функціональної програми, ввівши її в комп'ютер.

5. Оформити звіт про роботу.

7.3.3. Контрольні питання

1. Дати означення рекурсії.

2. Записати означення функції **сума**, яка в якості аргументу має список цілих чисел і як результат видає с

уму цих чисел.

3. Що таке підфункція головної функції?

4. Як побудувати добре структуровану програму?

5. Яка причина введення додаткових функцій при побудові функціональних програм?

7.4. Лабораторна робота № 10. Використання параметрів нагромадження у функціональному програмуванні

Мета: Освоєння методу з параметрами нагромадження при побудові функціональних програм**.**

7.4.1. Теоретичні відомості

Основна ідея методу з параметрами нагромадження полягає в тому, щоб визначити допоміжну функцію ·з зайвим параметром, який використовується для нагромадження необхідного результату.

Проілюструємо ідею методу, проектуючи функцію **обернути** (x), яка видає список з елементами, перерахованими в зворотному порядку:

X	**обернути** (x)
(A B C)	(C B A)
((A B)(C Д)	((C Д)(A B))
NIL	NIL

Відзначимо, що якщо елементи списку**,** в свою чергу**,** є списками, то елементи останніх не обертаються. Оскільки x є список, то необхідно зробити аналіз випадків. Якщо x є NIL, то **обернути** (x) також NIL. Якщо x = NIL, то можна використати **обернути** (**cdr**(x)), а для завершення побудови функції **обернути** необхідно мати функцію, яка би могла помістити **car**(x) в кінець **обернути** (**cdr**(x)). З цієї точки зору корисною є функція

добавити (x, y)

яка отримує список x і елемент y та робить у новим останнім членом списку x:

x	y	**добавити** (x, y)
(A B C)	Д	(A B C Д)
((A B) (C Д))	(E K)	((A B)(C Д)(E K))
NIL	A	(A)

Вважаючи, що ми маємо таку функцію можна завершити опис алгоритму побудови функції обернути(x):

випадок (1) **x = NIL обернути (x) = NIL**
випадок (2) **x ≠ NIL**
нехай обернути (cdr(x)) = z
тоді обернути (x) = добавити (z, car(x))
Таким чином, отримуємо функціональне означення
обернути (x) = якщо рівно (x, NIL) то NIL інакше
добавити (обернути(cdr(x)), car(x))

Тепер побудуємо функцію **добавити**. Це функція з двома аргументами, перший – описок, а другий – деякий S-вираз. Тому наш аналіз випадків стосується лише першого аргументу:

випадок (1) x = NIL добавити (x, y) = cons (y, NIL)
випадок (2) x ≠ NIL нехай добавити (cdr(x), y) = z
тоді добавити (x, y) = cons(car(x),z)

Остання побудова приводить до функціонального визначення

добавити (x, y) = якщо рівно(x, NIL) то cons (y, NIL) інакше
cons(car(x), добавити (cdr(x), y))

Тепер означення функції **обернути** використовує **добавити** як підфункцію, а тому ці функції утворюють програму обертання списку. Відзначимо, що можна б було визначити функцію **добавити** через раніше означену функцію **з'єднати** наступним чином:

добавити (**x, y**) = **з'єднати** (**x, cons** (**y, NIL**)).

Тоді наша програма обертання списку складалася б з трьох функціональних означень. Більш звично включити виклик **з'єднати** в функцію **обернути** та позбутися функції **добавити**, що приводить до такої програми

обернути(x) = якщо рівно(x, NIL) то NIL інакше
з'єднати (обернути(cdr(x)),cons(car(x), NIL))
з'єднати (x) = якщо рівно(x, NIL) то y інакше cons(car(x), з'єднати (cdr(x),y)

Цікаво підрахувати кількість викликів функції **cons** при реалізації цієї програми. Функція о**бернути** робить n викликів функції **cons**, якщо довжина списку n, але ж вона робить і n викликів функції **з'єднати**. Для кожного із цих викликів **з'єднати** довжина першого її аргументу рівна відповідно 0, 1, 2, … n-1. Тому кількість викликів **cons** функцією **з'єднати** від імені **обернути** є

$$0+1+2+.....+n-1 = n(n-1)/2.$$

Загальна кількість викликів **cons** в функції **обернути** буде $n + n(n-1)/2 = n(n+1)/2$.

Це число викликів є досить великим. Можна, використовуючи ідею методу параметрів нагромадження, перепрограмувати функцію **обернути**, щоб уникнути зайвих викликів функції **cons**, тобто зробити програму більш ефективною.

Визначимо функцію **обр**(x, y) з тією властивістю, що x – є список, який підлягає обертанню, а y – параметр, який використовується для нагромадження оберненого списку. Маємо означення функції

обр(x, y) = якщо рівно(x, NIL) то y інакше обр(cdr(x), cons(car(x),y))

через яку можна визначити функцію **обернути**(x):

обернути(x) = обр(x, NIL)

Коли викликана функція **обр**(x, y), список y вже нагромадив у зворотному порядку всі розглянуті до цього моменту елементи списку, який обертається. Таким чином, коли x є NIL, то y нагромадив весь результат в цілому, а якщо y = NIL, то ми можемо нагромадити в y **car** від x · рекурсивно викликати функцію **обр** для обробки **cdr** від x.

Проблема побудови таких функцій з параметрами нагромадження полягає в підборі підфункцій з параметром, який вирахує потрібний результат.

Підрахуємо кількість викликів функції **cons** в останньому варіанті означення функції **обернути**. Очевидно, функція **обернути**(x) робить свої

виклики **cons** у функції **обр**(x, NIL). Функція **обр**(x, y) викликає себе рекурсивно n разів, де n–довжина списку x. Отже, **обр**(x, y) викликає **cons** n разів, стільки ж разів викликає **cons** і функція обернути. Якщо порівняти з кількістю викликів функції **cons** n(n+1)/2 у першому означенні, то маємо суттєву економію при виконанні програми.

7.4.2. Послідовність виконання роботи

1. З курсу лекцій повторити теоретичні відомості про застосування методу з параметрами нагромадження до побудови функцій.

2. Для заданого варіанту підібрати підфункцію з параметром нагромадження і записати алгоритм програми шляхом аналізу можливих випадків.

3. Записати функціональне означення програми з параметрами нагромадження.

4. Перевірити працездатність програми, ввівши її в комп'ютер.

5. Оформити звіт про роботу.

7.4.3. Контрольні питання

1. Суть методу побудови функцій з використанням параметрів нагромадження.

2. Записати функцію, яка здійснює обертання як елементів вхідного списку, так і елементів внутрішніх підсписків.

3. Які переваги і особливості методу з параметрами нагромадження?

4. Означити арифметичну функцію з двома параметрами, яка в першому нагромаджує суму цілих чисел від ділення кожного елементу вхідного списку на два, а в другому добуток залишків від такого ділення.

7.5. Лабораторна робота № 11. Побудова функцій вищих порядків

Мета: Набуття практичних навиків побудови функцій вищих порядків.

7.5.1. Теоретичні відомості

Функції, як· використовують інші функції в якості аргументів**,** є прикладом того, що прийнято називати функціями вищих порядків. В функціональному програмуванні розумне використання функцій вищих порядків може привести до надзвичайно простих і потужних програм.

Розглянемо функцію, яка після застосування до списку цілих чисел, видає список тієї ж довжини, але з елементами на одиницю збільшеними порівняно з вихідним списком. Дію функції можна проілюструвати таблицею:

X	**збільшити (x)**
(1 2 3)	(2 3 4)
(0 -1 -2)	(1 0 -1)
(129)	(130)

Функціональне визначення цієї функції **збільшити**(x) можна записати у вигляді:

збільшити(x) = якщо рівно(x, NIL) то NIL інакше cons(car(x)+1, збільшити (cdr(x)))

Аналогічно функція

залишок(x) = якщо рівно (x, NIL) то NIL інакше cons(car(x) залишок 2, залишок (cdr(x)))

вираховує по вихідному списку список залишків від ділення кожного його елемента на 2, як показано в таблиці нижче:

X	**залишок (x)**
(0 2 3)	(0 0 1)
(0 -1 -3)	(0 -1 -1)
(129)	(1)

Подібність цих двох функцій та аналогічних їм наводить на думку, що можна ввести деяку загальну функцію, для якої конкретна арифметична операція, що використовується над кожним елементом списку (збільшення на одиницю, залишок від ділення на 2), є параметром функції. Да

вши цій узагальненій функції ім'я **відобразити**, можна дати таке її означення:

відобразити(x, f) = якщо рівно(x, NIL) то NIL інакше cons(f(car(x)), відобразити(cdr(x),f))

Тепер можемо перевизначити раніше описані функції за допомогою в**ідобразити** таким чином:

зб(y) = y+1
збільшити(x) = відобразити(x, зб)
зал(z) = z **залишок** 2
залишок(x) = **відобразити** (x, **зал**)

Таким чином, визначені функції **залишок** і **збільшити** є функціями вищих порядків, оскільки використовують інші функції в якості аргументів.

В якості другого прикладу функції вищого порядку розглянемо операцію редукції. Визначимо функцію **редукція** (x, g, a), де x – список, g – бінарна функція, a – постійна таким чином, що список x = (x ... x) приводиться до вигляду

g(x ,g(x),...g(x ,a)...)

Побудуємо цю функцію безпосередньо із аналізу можливих випадків:

випадок (1) x = NIL редукція(x, g, a) = a
випадок (2) x = NIL
нехай редукція (cdr(x), g, a) = z
тоді редукція (x, g, a) = g(car(x),z)

Отже, можна записати означення:

редукція(x, g, a) = якщо рівно (x, NIL) то a інакше g(car(x), редукція(cdr(x), g, a))

Маючи цю функцію вищого порядку, можна тривіально визначити функції **сума** і **добуток**, які відповідно видають суму і добуток від вихідного списку цілих чисел:

плюс(x, y) = x +y;
сума(x) = редукція(x, плюс, 0);
множити(x, y) = x*y;
добуток(x) = редукція(x, множити, 1)

Деякий недолік при використанні функцій вищих порядків зв'язаний з необхідністю давати імена функціям, які використовуються як фактичні параметри, Цього недоліку можна уникнути, якщо використати 2 вирази, значення яких є функцією. Функція є правилом для вирахування значення по деяких аргументах. λ-вираз містить у собі це правило, виділяючи імена параметрів · вираз, який заданий в термінах цих параметрів. Так λ-вираз

λ (z) z+1

вираховує функцію, яка, будучи викликаною при конкретному аргументі, видає це значення функції, збільшене на одиницю. Таким чином, найбільш доцільне використання λ-виразів у якості фактичних параметрів функцій вищих порядків.

Функції вищих порядків можна визначати таким чином, що вони беруть інші функції в якості аргументу і в якості результату. Визначимо так звану **композицію** або добуток функцій:

композиція (f, y) = (x)f(g(x))

Ця функція бере в якості аргументів дві функції і видає як результат також функцію, яка послідовно застосовує обидві вихідні функції, тобто функція **композиція** застосовує f до результату g. Наприклад,

збобр(x) = **композиція**(**збільшити**, **обернути**)

є функцією, яка спочатку обертає список, а потім збільшує його елементи на одиницю:

X	**збобр** (x)
(1 2 3)	(4 3 2)
(127 -127	(-126 128)

7.5.2. Послідовність виконання роботи

1. З курсу лекцій повторити теоретичний матеріал про побудову функції вищих порядків.

2. Згідно з варіантом записати алгоритм програми побудови функції вищого порядку.

3. Записати означення функції вищого порядку у термінах S-виразів.

4. Перевірити працездатність програми, ввівши її в комп'ютер.

5. Оформити звіт про роботу.

7.5.4. Контрольні питання

1. Дати означення функції вищих порядків.
2. Що таке λ-вираз?
3. Які особливості використання функцій вищих порядків?
4. Що таке глобальна змінна?
5. Побудувати функцію **сумдоб**(x), яка в якості результату видає двочлен, в якого перший елемент є сумою, а другий – добутком елементів x.

7.6. Лабораторна робота № 12. Структурне відлагодження програми

Мета: Набуття практичних навиків структурної побудови функціональної програми та її оптимального відлагодження.

7.6.1. Теоретичні відомості

При побудові функціональних програм головна програма може потребувати побудови ряду підфункцій, причому одна функція може бути вкладена в другу (функція вищих порядків) або сама в себе (рекурсія) на довільну глибину. Виникає задача вибору правильної структури побудови функціональної програми і оптимального відлагодження. Очевидно, щоб оптимальним чином провести відлагодження програми, необхідно побудувати повну схему викликів підфункцій, які необхідні для опису головної функції. На цій схемі, що виражається графом, вершини позначені іменами функцій, а дуга з однієї вершини в другу відображає той факт, що функція, яка стоїть біля вершини, з якої стрілка виходить, викликає функцію, що стоїть біля вершини, в яку стрілка входить.

Відлагодження програми можна проводити повністю методом знизу вверх, або можна включати елементи аналізу зверху вниз. При відлагоджені знизу вверх спочатку досліджуємо функції, які не викликають інших функцій, потім ті функції, які викликають вже відлагоджені, і так поступово до головної функції. Тестові дані вибираються таким чином, щоб при виявленні помилки були всі підстави сподіватися, що є неточності при побудові функції більш високого рівня, яка є новою для даного тесту. Таким чином, помилка локалізується. Функцію варто відлагоджувати лише після того, як відлагоджені всі підфункції, які вона викликає. Рекурсія дещо заплутує цей процес. Але завжди можна підібрати дані так, щоб освоїти рекурсивну функцію по частинам, заставляючи її викликати себе 0, 1 і більше число разів.

Аналогічно проводиться процес відладки взаємно рекурсивних функцій, підбираючи відповідно дані. Наприклад, якщо f викликає g і g викликає f таким чином:

f(...) = якщо ... то ...g(...) інакше... g(...) інакше якщо ... то ...f(...) інакше...

то ми повинні f і g відлагоджувати одночасно. Але завжди можна підібрати дані так, щоб f спочатку не викликала g, потім викликала g один раз, а g в свою чергу не викликала f і так далі, наближаючись поступово до повної відладки функцій f і g.

Як приклад структурованої програми напишемо функціональну програму, яка здійснює аналіз розмірностей. По суті справи треба написати функцію, яка б сприймала арифметичний вираз і видавала як результат його розмірність. Для побудови програми слід вибрати представлення для арифметичних виразів і розмірностей. Обмежимося арифметичними виразами, які побудовані зі змінних, констант і операцій +, -, *, /. Тоді представлення у вилці S-вираз в матиме вигляд:

правильний вираз	Представлення
змінні і константи	атом
e1 + e2	плюс (e1 e2)
e1 - e2	мінус(e1 e2)
e1 * e2	множ (e1 e2)
e1 / e2	діл (e1 e2)

Структура даних для представлення розмірностей може мати вигляд M^{n1} L^{n2} T^{n3}, де постійні n1 n2 n3 можуть бути від'ємні, додатні або рівні нулю. Тобто розмірність представлена трьох елементним списком. Так, швидкість L T представляється списком (0 1 -1), сила M L T – списком (1 1 -2). Для побудови функції розмірності слід ще передбачити структуру даних, що визначають розмірність змінних, що входять у формулу. Оскільки кожній змінній відповідає одна розмірність, то цей факт можна відобразити списком пар, кожна з яких відповідає змінній і зв'язаній з нею розмірності. Такий список називається асоціативним і має структуру

((В1 Д1)(В2 Д2)...(ВК ДК)),

де В1,...ВК – змінні (атоми), Д1,...ДК – їх розмірності.

Основна функція, за допомогою якої оперують з таким асоціативним списком це **асоц**(В, А), де В – змінна, розмірність якої треба видати, А – асоціативний список. Якщо допустити, що В входить у список, то можна записати функціональне означення для асоц:

асоц(В, А) = якщо рівно (В, car(car(А))) то
car(cdr(car(А))) інакше асоц(В, cdr(А))

Тепер можна розглянути побудову основної функції, розглянувши п'ять випадків правильних арифметичних виразів, які ми допускаємо. Назвавши цю функцію **розм(Е ,А),** де Е – вираз, який аналізується, а А – асоціативний список, можна записати такий алгоритм її побудови:

випадок (1)
Е – змінна тоді розм(Е, А) = асоц(Е, А)
випадок (2)
Е має форму (ПЛЮС Е1 Е2)
нехай розм(Е1, А) = Д1 і розм(Е2, А) = Д2 тоді Д1 і Д2 рівні
і розм(Е, А) = Д1
випадок (3)
Е має форму (МІНУС Е1 Е2) аналогічно (2)
випадок (4)
Е має форму (МНОЖ Е1 Е2)
нехай розм(Е1, А) = Д1 і розм (Е2, А) = Д2
тоді розм(Е, А) = множ (Д1, Д2)
випадок (5)
Е має форму (ДІЛ Е1 Е2) · розм(Е2, А) = Д2
нехай розм(Е1, А) = Д1 тоді розм(Е, А) = діл(Д1, Д2)

Тут випадки (2)–(3) є аналогічні в тому плані, що розмірність складного виразу є функцією розмірності його компонент. У випадку операцій МНОЖ і ДІЛ відповідні функції **множ**(Д1, Д2) і **діл**(Д1, Д2) поки що невизначені. Функція, яка використовується для визначення збігу розмірностей також ще не визначена. Назвавши її **збіг**(Д1, Д2) можна записати означення функції **розм**(Е, А):

розм(Е, А) = якщо атом(Е) то асоц(Е ,А) інакше
нехай Д1 = розм(car(cdr(е)), а) і
Д2 = розм(car(cdr(cdr(е))), а) якщо car(е) = ПЛЮС, то

якщо збіг(Д1, Д2) то Д1 інакше Ф
інакше якщо car(e) = МІНУС, то
якщо збіг (Д1, Д2) то Д1 інакше Ф
інакше якщо car(e) = МНОЖ то множ(Д1 Д2)
інакше якщо car(e) = ДІЛ то діл(Д1, Д2)
інакше NIL

Спосіб дії цієї функції достатньо прямолінійний. Спочатку вона присвоює розмірності внутрішнім змінним, потім визначає розмірності більш зовнішніх виразів і, на кінець, визначає розмірність усього виразу. Для завершення побудови програми слід означити функції **збіг**, **множ**., **діл**. Для вибраного представлення розмірності ці функції мають вигляд:

збіг(Д1, Д2) = якщо маса(Д1) = маса(Д2) то
якщо довжина(Д1) = довжина(Д2) то час(Д1) = час(Д2) інакше Ф
маса(Д) = car(Д)
довжина(Д) = car(cdr(Д))
час(Д) = car(cdr(cdr(Д)))
множ(Д1, Д2) = трійка (маса(Д1) + маса(Д2),
довжина(Д1) + довжина(Д2),
час(Д1) + час(Д2))
діл(Д1, Д2) = трійка(маса(Д1) - маса(Д2),
довжина(Д1) - довжина(Д2),
час(Д1) - час(Д2))
трійка(a, b, c,) = cons(a, cons(b,cons(c,NIL)))

Наведені означення говорять самі за себе і особливих пояснень не потребують. Якщо задано вираз

(ПЛЮС U (МІНУС(МНОЖ(F T) V))

і асоціативний список

((U(0 1 -1) (V(0 1 -1) (F(0 1 -2) (T(0 0 1))),

то функція **розм(E, A)** будує такі розмірності:

E	розм (E, A)
F	(0 1 -1)
T	(0 1 -2)
(МНОЖ F T)	(0 0 1)
V	(0 1 -1)
(МІНУС(МНОЖ F T)V)	(0 1 -1)
(ПЛЮС U(МІНУС(МНОЖ F T)V))	(0 1 -1)
	(0 1 -1)

При відлагоджені всієї програми потрібно кожну підфункцію відлагоджувати окремо. Функції **маса**, **довжина**, **час**, **трійка** є достатньо тривіальні, і для них можна уникнути такої перевірки. Однак функції **збіг**, **множ**, **діл** є менш тривіальні, і їх незалежне відлагодження має ту перевагу, що коли перейдемо до відлагодження функції **розм,** то можна покладатися на правильну роботу всіх функцій. Очевидно, функція **асоц** повинна бути відлагоджена незалежно і тоді головну програму можна відлагоджувати послідовно, спочатку з простими даними, а потім з кожною формою виразу почергово.

7.6.2. Послідовність виконання роботи

1. З курсу лекцій повторити теоретичний матеріал, пов'язаний з структурованою побудовою функціональних програм та їх відлагодженням.

2. Згідно з варіантом, вибрати структуру представлення даних та побудувати алгоритм структурованої програми шляхом аналізу можливих варіантів, вибравши найбільш оптимальний.

3. Записати означення головної функції та підфункцій, необхідних для її побудови.

4. Ввести програму в комп'ютер і провести відлагодження знизу вверх з використанням елементів аналізу зверху вниз.

5. Перевірити працездатність програми для різних наборів вхідних даних.

6. Оформити звіт про роботу.

7.6.3. Контрольні питання

1. Що таке асоціативний список?

2. Які є способи відлагодження структурованих функціональних програм?

3. Сформувати основні принципи структурування функціональних програм.

4. Описати спосіб подання розмірності у вигляді п’ятиелементного списку, вибравши величини розмірності в міжнародній системі СІ.

5. Сформулювати альтернативний спосіб подання розмірності довільного типу.

6. Дати альтернативне функціональне означення функції **збіг(d, d2)** без використання функції **дiл(d1, d2)** для випадку подання розмірності N - елементним списком. Порівняти ефективність побудованої функції з наведеною в пункті 5.3.2.

Розділ 8. Системний підхід до викладання декларативних мов програмування

Традиційні (алгоритмічні) мови програмування порівняно з декларативними (описовими) є доволі об'ємні і громіздкі, оскільки не дають змоги:

- максимально використовувати можливості сучасної комп'ютерної техніки для забезпечення ефективності програмних засобів;
- ясно і наочно відобразити алгоритми програми, щоб забезпечити легкість перевірки та модифікації останніх.

Строго функціональні та мови логічного програмування, що відносяться до декларативних мов програмування [1-11], є доволі простими, а тому забезпечують достатньо високий ступінь виразності програм, порівняно з традиційними мовами (Algol, Fortran, Pascal, Ci, Delphi, Ci++). Ряд функціональних програм можуть ефективно працювати на сучасних комп'ютерах, тим не менше не так ефективно, як відповідні програми з оператором присвоєння. Це пов'язано зі структурою архітектури сучасних комп'ютерів. Окрім того, вибір дещо іншої структури представлення даних, ніж це прийнято в інструментальному Ліспі, забезпечує як більшу ясність представлення програм, так і підвищення їх ефективності з використанням сучасних комп'ютерів старої архітектури.

З одного боку, сучасні мови програмування повинні ефективно використовувати сучасні машини, а з іншого – ясно виражати програмні алгоритми, щоб полегшити перевірку останніх. Строго функціональна мова, будучи простою за своєю структурою, демонструє вищий ступінь виразності порівняно з традиційними мовами, де існує оператор присвоєння. Це зв'язано, великою мірою, зі способом вибору структур даних. Дещо інший їх вибір можна забезпечити підвищенням ефективності функціональних програм і на сучасних комп'ютерах. Одна з фундаментальних властивостей мови програмування, що дає можливість ясно описати обчислення на цій мові – простота семантики мови. Велика перевага функціональної мови, як і логічної в тому, що тут є кілька основних понять, кожне з яких має просту семантику. Зокрема, семантика функціональної мови розуміється в термінах значень, які мають вирази, але аж ніяк не в термінах дій і послідовності їх використання. Але з практичного погляду було б справедливіше зробити висновок, що строго функціональна мова є надзвичайно елементарною і деякі

її розширення значно б підвищили ефективність і ясність деяких класів вирахувань. Очевидно, необхідно розрізняти суто синтаксичні розширення і розширення, які вимагають зміни семантики мови. Зміна семантики вимагає обережності, оскільки це ускладнює розуміння (ясність) вже відлагоджених функціональних програм

Будь-яку систему позначень для опису алгоритмів і структури даних можна назвати мовою програмування, хоча, як правило, вимагається ще й реалізація мови на комп'ютері.

Сьогодні розроблено сотні різних мов програмування. Ще в 1969 р. Саммет [1] наводить список з 120 мов, які доволі широко застосовуються. Ця приголомшлива кількість мов програмування протирічить тому, що більшість програмістів в своїй практиці використовує декілька мов програмування, а значна їх частина – одну або дві мови програмування. Виникає питання про доцільність освоєння різних мов програмування, якщо навряд чи буде можливість їх реалізації. Тим не менше, якщо не обмежуватися поверховим ознайомленням з мовою, а мати глибокі уявлення про поняття, які лежать в основі конструювання програм на даній мові, то поза всяким сумнівом можна переконатися у доцільності освоєння різних мов програмування, виходячи з таких міркувань:

1. Внаслідок вивчення різних мов програмування покращується розуміння конкретної мови, її понять та основних методів і прийомів, що в ній використовуються. Типовим прикладом може бути рекурсія. При правильному її використанні можна отримати елегантну ефективну програму, а застосування її до простого алгоритму могло б привести до астрономічного збільшення часових затрат. З іншого боку, недоступність використання рекурсії в таких мовах, як Фортран, Кобол та розуміння основних принципів і методів реалізації рекурсії може внести ясність в певні обмеження мови, які, на перший погляд є надуманими.

2. Значення мов програмування розширює запас корисних програмістських конструкцій і сприяє розвитку мислення. Працюючи з структурами даних однієї мови, виробляють і відповідну структуру мислення. Вивчаючи конструкції інших мов та методи їх реалізації, розширюють програмістський тезаурус.

3. Знання великої кількості мов програмування дає змогу обґрунтовано вибирати певну мову програмування для розв'язання конкретної задачі.

4. Освоєння нової мови програмування, як і природної мови людського спілкування, завжди легше, якщо відомими є декілька мов.

5. Знання принципів побудови різноманітних мов програмування полегшує розробку нової мови програмування.

Конструювання сучасних мов програмування сьогодні далеке від досконалості. Кожна з відомих мов має свої недоліки та переваги.

Більшість сучасних мов програмування є універсальними, оскільки дають змогу записати будь-який алгоритм цією мовою, якщо не накладати обмежень на час виконання програми та місткість пам'яті, алгоритмічну складність тощо. Якщо хто-небудь запропонує нову мову програмування, то вона, очевидно, буде універсальною, якщо ігнорувати обмеження на пам'ять або час. Порівнюючи різні мови програмування, слід виходити не з кількісного співвідношення того, що вони дозволяють зробити, а з якісних відмінностей, що визначають елегантність (короткість і наочність), легкість (прозорість) та ефективність (швидкодія та технічні засоби) програмування на них. Це порівняння слід здійснювати в контексті конкретної сфери застосування.

Очевидно для детального розуміння особливостей, синтаксису, семантики, діапазону прикладних застосувань декларативних мов програмування (Lisp [13-15; 19], Prolog [1-5; 8-11], UML [31-33]), які покликані обслуговувати задачі штучного інтелекту [5, 8, 11], при їх вивчення слід застосовувати системний підхід. Суть підходу зводиться до викладу матеріалу від простого до складного з наведенням численних прикладних задач, які в стані розв'язати на даній мові програмування. Поряд з цим слід чітко уявляти обмеження, які накладені на використовувану мову програмування та сферу її прикладних застосувань.

При системному підході забезпечується можливість постійного порівняння можливостей різних описових мов програмування [31], постійна зміна і оновлення прикладних задач та освоєння нових можливостей логічних предикатів в Пролозі чи функціональних описів в Універсальному Ліспі.

Авторам видається, що при викладанні декларативних мов програмування такий системний підхід можна забезпечити при викладенні лекційного, практичного матеріалу чи постановці лабораторних робіт у вигляді аплікацій, виконаних в мові програмування UML [33] чи іншій об'єктна-орієнтованій мові моделювання [31], яка забезпечує можливість як постійного поновлення підготовленого матеріалу, так і постійного порівняння з іншими підходами, їх оптимізації і подальшого вдосконалення.

Можна дійти висновку, що прогресу як у викладанні, так і у вивченні нового інформаційного матеріалу можна досягнути лише у процесі

постійного удосконалення та оновлення набутих знань, вмінь та навичок. Коли такого матеріалу накопичується значний обсяг, то без системного його впорядкування від простого до складного, класифікації за алгоритмічною складністю, функціональними можливостями та простотою подання і зберігання даних подальший розвиток неможливий.

Пропонований підручник виходячи з вищеописаного системного підходу до викладання та освоєння декларативних мов програмування містить дві частини, одна з яких присвячена викладенню основ та опису основних програмних засобів логічного програмування, а в другій частині подано основи функціонального програмування та програмні і технічні засоби для реалізації і подальшого їх розвитку. Кожна із частин підручника розбита на 16 параграфів, причому весь матеріал підібрано таким чином, що би рухатися від простого до складного, від очевидного до, на перший погляд, незрозумілого і логічно не зв'язаного. У випадку надмірної складності матеріалу параграфи розбиваються на пункти, які в свою чергу містять підпункти. При цьому кожен розділ підручника закінчується низкою контрольних завдань і вправ, які сформовано таким чином, що спочатку вимагається показати знання основних означень та найбільш вживаних предикатів та вбудованих функцій, далі виявити розуміння логіки побудови логічних і функціональних програм, лише після цього формулюються контрольні вправи і завдання на самостійне опрацювання, що забезпечує як краще розуміння викладеного матеріалу, так і сприяє формуванню логічного мислення і творчого підходу до освоєння та застосування декларативних мов програмування.

Виклад матеріалу ілюструється численними прикладами як прикладного, так і теоретичного спрямування. Дається порівняльна характеристика функційних можливостей кожної із мов декларативного програмування, їх переваг та вузьких місць, відзначаються напрями їх доцільного застосування.

У додатках до підручника наведено :

А. Вбудовані предикати Турбо-Прологу. При чому проведено їх розділ як за функційним призначенням, так і наведено їх подання в алфавітному порядку.

Б. Варіанти індивідуальних завдань для виконання лабораторних робіт з логічного та функціонального програмування, які сформовано в міру їх поступового ускладнення.

В. Термінологічний словник термінів та означень, які найчастіше вживаються при використанні описових мов програмування або пов'язані з ними. Наведені терміни сформовано в алфавітному порядку, що спрощує їх пошук.

Г. Перелік основних видань автора (монографії, підручники, посібники) наведено з метою ознайомлення студентів з останніми виданнями в галузі декларативних мов програмування (Лісп, Пролог, Редюс та їх численними модифікаціями), а також зацікавити студентів до проведення наукових досліджень, пов'язаних зі створенням інтерпретаторів мов програмування, включаючи мову Лісп; з побудовою програмних систем підтримкм усного активного діалогу з елементами самонавчання; зі створенням систем розпізнавання почерку за руко моторними реакціями людини; з розробленням експертних систем, здатних імітувати людське вислення, висловлювати гіпотези та доводити теореми.

Велика перевага декларативних мов програмування перед процедурними у їх гнучкості, наглядності, порівняно при нескладних технічних засобах розв'язувати складні науково-прикладні задачі, які прийнято класифікувати як задачі, пов'язані зі створенням систем штучного інтелекту.

У противному разі задача, якщо і буде розв'язана, то з врахуванням тих обмежень, які властиві чи даному програмному забезпеченню, чи комп'ютеру на якому вона вирішується. Власне для розв'язання задач штучного інтелекту потребується створення комп'ютерів, структура яких відмінна від неймановської, а нагадує структуру людського мозку: порівняно невелике число нейронів між якими існує нескінченне число зв'язків. Ось ця нескінченна послідовність зв'язків і дозволяє постійно удосконалювати комп'ютерну систему, потребуючи при цьому практично залишати незмінними набір технічних засобів.

На сьогоднішньому етапі розвитку інформаційних технологій програмне забезпечення повністю оновлюється практично за п'ять років, в той час як технічні засоби зазнають не суттєвих змін за цей проміжок часу.

Список літератури

Основна література з логічного програмування

1. Клоксин У., Меллиш К. Программирование на языке Пролог .– М. : Мир, 1987. – 336 с.

2. Хоггер К. Введение в логическое программирование. – М. : Мир. 1988. – 348 с.

3. Братко И. Программирование на языке Пролог для искусственного интеллекта. – М. : Вильямс, 2004. – 640 с.

4. Макаллистер Дж. Искусственный интеллект и Пролог на микро ЭВМ. – М. : Машиностроение, 1990. – 240 с.

5. Ин Ц., Соломон Д. Использование Турбо-Пролога: Пер. с англ. – М. : Мир, 1993. – 608 с.

6. Бакаев А. А., Гриценко В. И., Экспертные системы и логическое программирование. – К. : Вища школа, 1995. – 246 с .

7. Заяць В. М., Семотюк В. М., Методичні вказівки до виконання лабораторних робіт з курсу "Логічне програмування". – Л. : В-во "Львівська політехніка", 1998. – 23 с.

8. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. – М. : Мир, 1990. – 235 с.

9. Nilsson N. J. Principles of Artificial Intelligence. – Tioga.-Springer-Verlag. – 1980. – 164 с.

10. Заяць В. М. Логічне програмування: Частина 1: Конспект лекцій з дисципліни «Логічне програмування» для студентів базового напрямку 6.08.04 "Програмне забезпечення автоматизованих систем". – Львів : Видавництво Національного університету "Львівська політехніка", 2002. – 48 с.

11. Адаменко А., Кучуков А. Логическое программирование и Visual Prolog. – Санкт-Петербург : БХВ-Петербург, 2003. – 992 с.

Основна література з функціонального програмування

12. Mccarthy J. Recursive functions of symbolic expressions and their computation by machine // Comm. ACM. : – 1960. – Vol. 3. – P. 184–195.

13. Хендерсон П. Функциональное программирование. Применение и реализация. – М. : Мир, 1983. – 349 с.

14. Бадаєв Ю. І. Теорія функціонального програмування. Мови CommonLisp та AutoLisp. – Київ, 1999. – 150 с.

15. Зяць В. М. Конспект лекцій з курсу "Функціональне програмування". – Львів, 1999. – 55 с.

16. Маурер У . Введение в программирование на языке Лисп. – М. : Мир, 1976. – 104 с.

17. Хювенен Э., Слеппян Й. Мир Лиспа. В 2-х т. Т.1. Введение в язык Лисп и функциональное программирование. Пер. с финск. – М. : Мир, 1990. – 447 с.

18. Хювенен Э., Слеппян Й. Мир Лиспа. В 2-х т. – Т. 2 Методы и системы программирования. Пер. с финск. – М. : Мир, 1990. – 447 с.

19. Заяць В. М. Функційне програмування: Навч. підручник. – Львів : Видавництво "Бескид Біт", 2003. – 160 с.

20. Кнут Д. Искусство программирования для ЭВМ: Пер. с англ. – Т. 1. – М. : Мир, 1967. – 734 с.

21. Заяць В. М., Заяць М. М. Логічне і функційне програмування: Навч. посібник. – Львів : Видавництво "Бескид Біт", 2006. – 352 с.

Додаткова література

22. Крюков А. П., Родионов А. Я., Таранов А. Ю., Шаблыгин Е. М. Программирование на языке R-Лисп. – М. : Радио и связь, 1991. – 192 с.

23. Еднерал В. Ф., Крюков А. П., Родионов А. Я. Язык аналитических вычислений REDUCE. – М. : Из-во МГУ, 1984. – 176 с.

24. Лавров С. С., Силигадзе Г. С. Автоматическая обработка данных. Язык Лисп и его реализация. – М. : Наука, 1978. – 176 с.

25. Уинстон П. Искусственный интеллект. – М. : Мир, 1980. – 513 с.

26. Грин Д., Кнут Д. Математические методы анализа алгоритмов. – М. : Мир, 1987. – 120 с.

27. Фостер Дж. Обработка списков. – М. : Мир, 1974.

28. Вирт Н. Алгоритмы + структуры данных = программы: Пер. с англ. – М. : Мир, 1985. – 213 с.

29. Бердж В. Методы рекурсивного программирования. – М. : Машиностроение, 1983. – 248 с.

30. Заяць В. М. Логічне і функціональне програмування. Навч. посібник. – Гриф надано МОН України (протокол № від 13.10.2013). – Рівне, 2016. – 400 с.

31. Erich Gamma, Richard Helm, Ralih Johnson, John Vlissides. Wzorce projektowe, 2012.

32. Заяць В. М. Системний підхід до освоєння та викладання мов логічного і функціонального програмування. – 21-й Міжнародний молодіжний форум «Радіоелектроніка і молодь в XXI столітті». – Харків, ХНУРЕ, 2017. – С. 4–6.

33. Stanislaw Wrycha i inni. UML 2.1. Cwiczenia. ISBN: 978-83-246-0612-2012.

Додаток А

Вбудовані предикати Турбо-Прологу

Вбудовані або стандартні предикати є основою побудови будь-якої логічної чи функціональної мови (оскільки будь-яка функція може бути приведена до логічного подання). В першому розділі додатку А приведені логічні предикати за їх функціональним визначенням. У другому розділі додатку А наведені ці предикати в алфавітному порядку з вказанням їх синтаксичного імені, списку аргументів, їх типу та можливих способів застосування. Наведений також короткий опис результату виклику того чи іншого предикату.

А1. Групи предикатів за функціональним призначенням

Системні предикати:

beep, bios,comline, date, keyword, memword, portbyte, sound, storage, system, time, trace.

Мовні предикати:

bound, exit, fail,findall, free, not.

Предикати перетворення типів:

char_int, str_char, str_int, str_real, upper_lower.

Предикати для роботи з файлами:

closefile, consult ,deletefile, dir, disc, eof, existfile, filemod, filepos, file_str, flush, openappend, openmodify, openread, readdeviuce, renamefile, save, writedevice.

Предикати для читання даних:

readchar, readint, readreal, readterm.

Предикати для запису даних:

nl, write, writedevice, writef.

Предикати для роботи з екраном монітору:

attribute, back, clerwindow, cursor, cursorform< display, dot, edit, editmag, field_attir, field_str, forward, gotowindow, graphics, line, makewindow, pencolor, pendown, penpos, penup, removewindow, scr_char, scroll, shiftwindow, text, window_str.

Предикати для роботи з базами даних:

asserta, assertz, consult, retract, save.

Предикати для роботи з рядочками символів:

concat, frontchar, frontstr, fronttoken, isname, str_len.

А2. Набір логічних предикатів в алфавітному порядку

asserta(<факт>) (dbasedom): (вх) – заносить факт у початок резидентної бази даних (аргумент, позначений як dbasedom, автоматично оголошується для кожного предиката з розділу database);

assertz(<факт>) (dbasedom): (вх) – заносить факт у кінець резиденетної бази даних;

attribute (Attr) (integer): (вх), (вих) – (вх): встановлює значення атрибуту, (вих): присвоює змінній Attr встановлене по замовчуванню значення атрибуту;

back(Steep) (integer): (вх) – зсуває екранне перо на кілька позицій, що визначаються змінною Steep назад. При виході пера за межі екрану предикат неуспішний;

beep – при його виклику подається звуковий сигнал;

bios(Ino,Ri, Ru) (integer, regdom, regdom): (вх, вх, вих) – відбувається звертання до програми системних переривань. Ino – задає номер переривання, Ri – містить нові номери регістрів, **Ru** – старі. Тип regdom в Турбо Пролозі є попередньо визначений у вигляді набору 8 доменів цілого типу:

regdom = (integer, integer, integer, integer, integer, integer, integer, integer,).

В цьому визначенні компоненти цілого типу відповідають регістрам мікропроцесора AX, BX, CX, DX, SI, DI, DS, ES.

bound(Var) (<довільна_змінна>): (вх) – предикат успішний у випадку визначеності змінної Var;

char_int(C, I) (char, integer): (вх, вих), (вих,вх), (вх, вх) – (вх, вих): присвоює змінній I код ASII, якому відповідає значення змінної I, (вих, вх): присвоює змінній C символ, код якого відповідає значенню I, (вх, вх): предикат завершиться успішно, якщо коду символа C відповідає значення I;

clearwindow – очищає активне в момент виклику вікно і заповнює його кольором фону;

closefile (S) (file) : (вх) – закриває фізичний файл, який ототожнений з логічним файлом з іменем **S.** Файл успішний навіть у випадку, якщо файл з логічним іменем **S** закритий в момент виклику предикату;

comline(S) (string) : (вих) – дозволяє полічити параметри керування програми;

concat (S1, S2, S3) (string, string, string) : (вх, вх, вих), (вх, вих, вх), (вих, вх, вх), (вх, вх, вх) – (вх, вх, вих): створює ланцюг символів S3 , склеюючи набори символів S2 і S3, (вх, вих, вх): якщо S2 є частиною S3, то S1 є залишком 3, (вих, вх, вх): якщо 2 є частиною рядочка 3, то 1 є залишком

3, (вх, вх, вх) : предикат успішний, якщо 3 співпадає з результатом склеювання 1 і 2;

cursor(T) (string) : (вх) – загружає в пам'ять файл з іменем T, якщо він текстовий;

cursor(R, C) (integer, integer) : (вх, вх), (вих, вих) – (вх, вх) **:** встановлює курсор в позицію з координатами (R, C), (вих, вих): присвоює змінним R і C значення координат курсора в момент виклику предиката;

date (Y, M, D) (integer, integer, integer) : (вх, вх, вх), (вих, вих, вих) - (вх, вх, вх) : встановлює нову дату в такому порядку – (рік, місяць, день). (вих, вих, вих) **:** зчитує дату, яка визначається вбудованим годинником у комп'ютер;

deletefile(N) (string) : (вх) – видаляє файл з логічним іменем N з відкритої директорії;

dir (P, F, N) (string, string, string) : (вх, вх, вих) – активізує засоби роботи з файлами. Значення змінних P і визначають шлях доступу і розширення файлів, імена яких появляються в активному вікні. Ім'я вибраного користувачем файлу буде присвоєно змінній N;

disc(S) (string) : (вих) - визначає нове ім'я біжучого диску і новий біжучий шлях доступу. (вих) - присвоює змінній S значення заданих по замовчуванню імені диску та шляху доступу;

display(S) (string) : (вих) - висвічує значення змінної S в тому вікні, яке активне в момент виклику;

dot(R, Column, Color) (string) : (вих) - (integer, integer, integer) : (вх, вх, вх), (вх, вх, вих) – (вх, вх, вх) **:** малює на екрані невеликий піксель заданого кольору Color в позиції з координатами (R, Column) за умови, що дисплей знаходиться в графічному режимі. (вх, вх, вих) – змінній Color присвоюється номер кольору пікселя, координати якого задаються змінними R і Column;

edit(Is, Os)(string, string) : (вх, вих) – активізує редактор Турбо Прологу, що дає змогу редагувати рядочок Is. Нова реакції буде присвоєна змінній Os;

eof(F) (file) : (вх) – предикат успішний, якщо курсор знаходиться на мітці кінця файлу F;

existfile(Dos_Name_file) (string) : (вх) – предикат успішний, якщо в біжучій директорії існує файл з іменем, що задається змінною Dos_Name_file;

exit – зупиняє виконання програми і передає керування системі Турбо Прологу, якщо програма запущена в середовищі Турбо Прологу, або DOS, якщо запуск програми здійснювався за межами Турбо Прологу;

fail – викликає механізм повернення при погодженні цільового твердження внаслідок неуспіху цього предикату незалежно від місця його знаходження;

field_str(R, C, L, S) (integer, integer, integer, string) : (вх, вх, вх, вх), (вх, вх, вх, вих) – записує в активне вікно екрану L символів із стрічки S, починаючи з позиції з координатами (R, C,) (поле читання повинне знаходитися в межах активного в даний момент вікна);

filepos(F, N, M) -(file, integer, integer) : (вх, вх, вх), (вх, вих, вх) – (вх, вх, вх): задає в файлі F позицію з якої буде зчитаний символ або в яку буде занесений черговий символ (M= 0 вказує на те, що зміщення буде відраховуватися від початку файлу, M= 1 – від біжучої позиції, M = 2 – від кінця файлу). (вх, вих, вх) : змінна N отримує значення зміщення біжучого положення вказівники файлу. Зміщення береться відносно початку файлу;

file_str(F, S) (string, string) : (вх, вх), (вх, вих) – (вх, вх) : записує рядочок S, обсягом не більше 64 Кб, в файл F. (вх, вих): зчитує в рядочок S символи, які містяться в файлі F, обсягом не більше 64 Кб;

findall(V, <предикат>, L) – записує значення об'єкта V в список L. Об’єкт V повинен бути одним із аргументів вказаного предиката;

flush (F) (file) : (вх) – викликає виведення на біжучий пристрій writedevice вмістиме, що знаходиться в оперативній пам’яті буфера файлу;

forward(S) (integer) : (вх) – в графічному режимі зсуває екранне перо на S позицій вперед;

free (V) (<змінна>) : (вих) – успішний**,** якщо змінна V не означена;

frontchar (S, C, R) (string, char, string) (вх, вих, вх), (вх, вх, вих), (вх, вих, вх), (вх, вх, вх), (вих, вх, вх) – (вх, вих, вх) : присвоює перший символ рядочка S змінній C, а залишок рядочка S – змінній R. Можливі будь-які інші комбінації вхідних і вихідних аргументів, при цьому мусять бути означені або змінна S, або змінні C і R одночасно;

frontstr(N, S1, SS, S2) (integer, string, string, string) : (вх, вх, вих,вих) –присвоює перші N символів рядочка S1 змінній SS, а залишок – змінній S2;

fronttoken(S, T, R) (string, string, string) : (вх, вих,вих), (вх, вх, вих), (вх, вих, вх,), (вх, вх, вх), (вих, вх, вх) – (вх, вих, вих): змінній S присвоюється результат конкатенації T і R. T може бути або групою символів, що задає допустиме в Турбо Пролозі ім’я, або символьним

поданням цілого чи дійсного числа, або одиночним символом, відмінним від пропуску. В усіх інших можливих комбінаціях вхідних і вихідних аргументів повинні бути означені, по крайній мірі, два аргументи;

gotowindow(W) (integer) : (вх): здійснює дуже швидкий перехід з одного вікна до іншого за умови, що ці два вікна не пересікаються. Також може бути використаний для переходу у вікно , що знаходиться позаду активного вікна;

graphics(M, P, B) (integer, integer, integer) : (вх, вх, вх): активізує графічні засоби Турбо –Прологу і встановлює номер режиму роботи M, палітри P і кольору фону B;

isname(S) (string) : **(вх)** – предикат успішний, якщо значенням аргумента S являється допустиме в Турбо Пролозі ім'я;

keypressed(P) (string) : **(вх)** – успішний, якщо буде натиснута довільна клавіша;

line(R1, C1, R2, C2, Color) (integer, integer, integer, integer) : (вх, вх, вх, вх, вх) : у графічному режимі малює лінію між точками з координатами R1, C1 і R2, C2, номер кольору задається аргументом Color;

makewindow(WN, S, F, H, R, C, HT, W) (integer, integer, integer, string, integer, integer, integer, integer) : (вх, вх, вх, вх, вх, вх, вх, вх), (вих, вих, вих, вих, вих, вих, вих, вих) : створює вікно, номер якого задається WN. Координати вікна задаються аргументами R і C, висота і ширина – HT і W. Вікно повинно поміщатись на екрані. Якщо F відмінний від нуля, навколо вікна з'явиться бордюр. H задає назву вікна;

membyte(S, O, B) (integer, integer, integer) : (вх, вх, вх), (вх, вх, вих). (вх, вх, вх) : записує значення B (1 байт) в пам'ять за адресою, що задається за допомогою S і O.(вх, вх, вих) : зчитує в змінну з пам'яті байт, адреса якого задається аргументами S і O. В обох випадках адреса вираховується за даною формулою S*16+O;

memword(S, O, W) (integer, integer, integer) : (вх, вх, вх), (вх, вх, вих) - (вх, вх, вх) : записує значення W (1 слово) в пам'ять за адресою, заданою за допомогою S і O. (вх, вх, вих) : зчитує в змінну W з пам'яті слово, адреса якого задається аргументами S і O. В обох випадках адреса вираховується за даною формулою S*16+O;

n1: відсилає на поточний пристрій W комбінацію спецсимволів "повернення каретки" і "переведення рядка"

not(<твердження>): успішний, якщо <твердження> являє собою ціль, виявлену при доведенні не успішною;

openappend(SymbolicFileName,DOS_FileName) (File,string) : (вх,вх): відкриває для дозапису файл DOS_FileName ототожнює його з логічним файлом SymbolicFileName;

openmodify(SymbolicFileName,DOS_FileName) (File,string) : (вх,вх): відкриває для читання і запису файл DOS_FileName ототожнює його з логічним файлом SymbolicFileName;

openread(SymbolicFileName,DOS_FileName) (File,string) : (вх,вх): відкриває для читання файл DOS_FileName ототожнює його з логічним файлом SymbolicFileName;

openwrite(SymbolicFileName,DOS_FileName) (File,string) : (вх,вх): відкриває для запису файл DOS_FileName ототожнює його з логічним файлом SymbolicFileName;

pencolor(C) (integer) : (вх): задає колір, в який буде зафарбоване графічне екранне перо. Застосовується тільки в графічному режимі;

pendown: після виконання цього предиката стає можливим малювання ліній за допомогою предикатів forward і back;

penpos(R,C,D) c, (вих, вих, вих): (вх,вх,вх): поміщає графічне перо в позицію з координатами(R,C) і орієнтує його в напрямку, заданим аргументом Direction. (вих,вих,вих): присвоює змінним координати поточну позицію пера і номер його напрямку;

penup : припиняє малювання, тобто виконує дію, обернену дію предиката pendown;

portbyte(PN,V) (integer, integer) : (вх, вх), (вх, вих): (вх,вх,вх) : значення V посилається в порт вводу-виводу номер PN. (вх, вих): присвоює V десятинний еквівалент значення байта з порта вводу-виводу номер PN;

ptr_dword(SV,S,O) (string, integer, integer): (вх, вих, вих), (вих, вх, вх): (вх, вих, вих) : видає номер сегмента S і зміщення O означеної змінної SV. (вих, вих, вх) : присвоює змінній SV рядочок символів, розміщених за адресою, заданою аргументами S і O. Адреса вираховується за даною формулою S*16+O. Кінець рядочка визначається пустим бітом;

readchar(C) (char) : (вих): зчитує символ з поточного пристрою зчитування readdevice;

readdevice(S) (symbol) : (вх), (вих): (вх) : назначає для зчитування файл з логічним іменем S.(вих) : змінна S означується логічним іменем поточного пристрою зчитування;

readint(I) (integer) : (вих) : зчитує з поточного пристрою зчитування readdevice ціле число;

readln(S) (string) : (вих) : зчитує з поточного пристрою зчитування readdevice символьний рядок;

readreal(R) (real) : (вих) : зчитує з поточного пристрою зчитування readdevice дійсне число;

readterm(D, T) (<ім'я домена>, <змінна>) : (вх, вих) : дозволяє зчитувати з відкритого файлу будь-який об'єкт, записаний туди за допомогою предиката write. Об'єкт присвоюється змінній T, оскільки він відповідає опису домена цієї змінної;

removewindow – знищує існуюче вікно;

renamefile(OF, NF) (string, string) : (вх, вх) : перейменовує файл з іменем OF, нове ім'я задається змінною NF;

save(F) (string) : (вх) : зберігає твердження динамічної бази даних в текстовому файлі F;

scr_attr(R, C, A) (integer, integer, integer) : (вх, вх, вх), (вх, вх, вих): (вх, вх, вх) : визначає значення атрибуту позиції екрану з координатами R і C. (вх, вх, вих) : присвоює змінній A значення атрибуту позиції екрану з координатами R і C;

scr_char(R, C, C) (integer, integer, char) : (вх, вх, вх), (вх, вх, вих): (вх, вх, вх): записує символ Char в позицію екрана з координатами (R,C). (вх, вх, вих): означує змінну Char символом з позиції екрана з координатами (C, R);

scroll(NOR,NOF) (integer, integer): (вх, вх) : зміщує активне вікно вверх або вниз на кількість рядків, заданих параметром NOR, вліво або вправо на кількість колонок, заданих параметром NOC. Зміщення вверх і вліво відповідає від'ємним числам, вниз і вправо – додатнім;

shiftwindow(WN) (integer), (вх, вих): (вх): активізує вікно з номером WN, а також зберігає вмістиме вікна, активного в момент виклику предиката. (вих): присвоює змінній WN номер активного в даний момент вікна.

sound(D,Fr) (integer, integer) : (вх, вх) : викликає звучання ноти частоти Fr і протяжності D (протяжність задається в сотих долі секунди)

storage(SS,HS,TS) (real,real,real) : (вих, вих, вих) : видає доступний об'єм (в Кб) трьох динамічних областях пам'яті;

str_char(SP, CP) (string, char) : (вх, вих), (вих, вх), (вх,вх): (вх, вих): присвоює CP символ, заданий за допомогою SP. (вих, вх): присвоює SP символ, заданий за допомогою CP. (вх, вх): успішний, якщо SP і CP являє один і той же символ.

str_int(S, I) (string, integer) : (вх, вих), (вих, вх), (вх,вх): (вх, вих): присвоює I подвійний еквівалент десятинного символьного представлення цілого числа S. (вих, вх): присвоює S рядок, який являється десятинним символьним представленням цілого числа I. (вх,вх): успішний, якщо S і I представляють одне і теж число;

str_len(S, L) (string, integer) : (вх, вх), (вх, вих): (вх, вх) – успішний, якщо в рядочку S існує L символів. (вх, вих): присвоює змінній L кількість символів, що містяться в рядочку S;

str_real(S, R) (string, real) : (вх, вих), (вих, вх), (вх, вх): (вх, вих):присвоює R подвійний еквівалент десяткового символьного представлення дійсного числа S. (вих, вх): присвоює S рядочок, що являється десятковим символьним представленням дійсного числа R. (вх, вх) – успішний, якщо S і R подають одне і те ж число;

system(C) (string) : (вх) : дозволяє виконати команду операційної системи, подану у вигляді символьного рядочка;

text – повертає екран в алфавітно - цифровий режим;

time(H, M, S, HS) (integer, integer, integer, integer) : (вх, вх, вх, вх), (вих, вих, вих, вих): (вх, вх, вх, вх): встановлює системний годинник: H – години, M – хвилини, S – секунди, HS – соті долі секунди. (вих, вих, вих, вих): присвоює змінним покази системного годинника;

trace(S) (symbol) : (вх, вих): (вх): запис предиката trace(on) на початок програми задає покрокове її виконання – трасування. Відміна трасування здійснюється за допомогою trace(off). Аналогічні функції має і предикат різниця лише в тому, що кількість інформації яка видається у вікно трасування (Trace Window) дещо зменшується. (вих): присвоює змінній S значення on або off залежно від режиму виконання програми;

upper_lower(SI, SL) (string, string) : (вх, вих), (вих, вх), (вх, вх): (вх, вих): присвоює змінній SL еквівалент рядочка SI, в якому усі великі літери замінені на маленькі. (вих, вх): присвоює змінній SI еквівалент рядочка SL, у

якому всі маленькі літери замінені на великі. (вх, вх): успішний, якщо змінні SI і SL представляють відповідно нижньорегістрову і верхньорегістрову версії одного і того ж рядка;

window_attr(A) (integer): (вх): приводить в відповідність зі значенням А атрибут активного вікна;

window_str(SS) (string): (вх), (вих): (вх): висвічує в активному вікні екрана значення змінної SS. (вих): присвоює змінній SS рядок, висвічений в активному вікні;

write(e1,e2,e3, … ,eN) ((вх)*): здійснює вивід на поточний пристрій writedevice констант або значень. Кількість аргументів довільна; це можуть бути або константи, або змінні, значення яких відносять до одному із стандартних доменних типів;

writedevice(SFN) (symbol): (вх),(вих): (вх): якщо файл з логічним іменем SFN відкритий, то даний предикат перепризначує на нього пристрій виводу writedevice. (вих): присвоює змінній SFN логічне ім'я writedevice;

writef(FS,Arg1, Arg2, Arg3, …) (вх,(вх)*): здійснює форматований вивід інформації. Формати задаються в вигляді рядка FS звичайного тексту, де маркери % відмічають положення аргументів. Допустимі специфікації формату – це "-", "m" "p" "f" "e" "g". "-" означає вирівнювання зліва; цілі числа, які йдуть після "m" і "p", задають кількість цифр зліва і справа від десяткової точки. Специфікації "f" і "e" визначають мантису і порядок числа. Специфікація "g" задає друк в максимально короткій формі;

Додаток Б

Варіанти індивідуальних завдань для виконання лабораторних робіт з логічного та функціонального програмування

Б1. Функціональне програмування

Лабораторна робота № 1

Тема: Програмування за допомогою функцій

Варіанти завдань:

1. Довизначити функцію CAR(X), задавши нове ім'я з буквою M в кінці, щоби вона видавала значення FALSE у випадку, коли X – атом.

2. Довизначити функцію CDR(X), задавши нове ім'я з буквою M в кінці, щоби вона видавала значення FALSE у випадку, коли X – атом.

3. Довизначити функцію CONS(X,Y), задавши нове ім'я з буквою M в кінці, щоби вона видавала значення FALSE у випадку, коли Y – атом.

4. Довизначити функцію ATOM(X), щоби вона видавала значення FALSE у випадку, коли X – не атом.

5. Визначити функцію UNTUATOM(X), щоби вона видавала значення TRUE у випадку, коли X – не атом і FALSE у випадку, коли X – атом.

Лабораторна робота № 2

Тема: Побудова елементарних функцій

Варіанти завдань:

1. Побудувати функцію, яка із заданого списку з 6 елементів вибирає найбільший, використовуючи лише примітивні функції Ліспу та вбудовані в нього арифметичні операції.

2. Побудувати функцію, яка із заданого списку з 5 елементів вибирає найменший, використовуючи лише примітивні функції Ліспу та вбудовані в нього арифметичні операції.

3. Побудувати функцію, яка із заданої структури ((A B) (C D)) видає елемент B.

4. Побудувати функцію, яка із заданої структури ((A B) (C D)) видає елемент C.

5. Побудувати функцію, яка із заданої структури ((A B) (C D)) видає елемент D.

Лабораторна робота № 3

Тема: Побудова рекурсивних функцій

Варіанти завдань:

1. Записати алгоритм, функційне означення та S-вираз функції останній(X), яка видає останній елемент зі списку X. Привести результат роботи програми.

2. Записати алгоритм, функційне означення та S-вираз функції другий(X), яка видає другий елемент зі списку X. Привести результат роботи програми.

3. Записати алгоритм, функційне означення та S-вираз функції обернути(X), яка видає елементи списку X у зворотному порядку. Привести результат роботи програми.

4. Записати алгоритм, функційне означення та S-вираз функції сумдобуток(X), де X – простий список, складений з чисел, яка видає двочлен, у якому на першому місці стоїть сума чисел, заданих у списку, а на другому – добуток цих чисел.

5. Записати алгоритм, функційне означення та S-вираз функції довжина(X), яка видає кількість елементів у списку X. Привести результат роботи програми.

Лабораторна робота № 4

Тема: Побудова складних рекурсивних функцій з параметрами нагромадження

Варіанти завдань:

1. Записати алгоритм, функційне означення та S-вираз функції останній(X), яка видає останній елемент зі складного списку X і сам список X. Привести результат роботи програми.

2. Записати алгоритм, функційне означення та S-вираз функції довжина(X), яка видає кількість елементів зі складного списку X. Привести результат роботи програми.

3. Записати алгоритм, функційне означення та S-вираз функції обернути(X), яка видає елементи зі складного списку X у зворотному порядку і сам список X. Привести результат роботи програми.

4. Записати алгоритм, функційне означення та S-вираз функції сумдобуток(X), де X – складний список, складений з чисел, яка видає двочлен, у якому на першому місці стоїть сума чисел, заданих у списку, а на другому – добуток цих чисел. Привести результат роботи програми.

5. Записати алгоритм, функційне означення та S-вираз функції довжина(X), яка видає кількість елементів у складному списку X. Привести результат роботи програми.

Лабораторна робота № 5

Тема: Побудова програми символьного диференціювання

Варіанти завдань:

Побудувати програму символьного диференціювання математичних виразів, в які входять операції " + ", " – ", " * ", " / " та функції відповідно з варіантом:

1. Функції (EXP X) і (ACOS X).
2. Функція (EXPT X) і (ATAN X).
3. Функція (LOG N M) і (ASIN X).
4. Функція (SQRT N) і (COS X)..
5. Функція (SIN X) і (ATAN X).

Лабораторна робота № 6

Тема: Побудова програми аналізу розмінностей математичних виразів

Варіанти завдань:

Побудувати програму аналізу розмінностей розм.(A, E) математичних виразів, де A – асоціативний список в якому на першому місці – змінна, а на другому її розмірність. В математичні вирази водять чотири арифметичні операції " + ", " – ", " * ", " / ", а розмірність подана трьохелементним списком, де на першому місці одиниці довжини, далі – маси, далі – часу. Результат програми – розмірність виразу, або атом NIL, якщо формула непогоджена за розмірністю. Перевірку провести для формул згідно заданого варіанту:

1. $F = m \cdot a$ – другий закон Ньютона.
2. $E = m \cdot v^2$ – формула кінетичної енергії.
3. $E = m \cdot g \cdot h$ – формула потенціальної енергії.
4. $S = v \cdot t + a \cdot t^2 / 2$ – формула рівноприскореного руху.
5. $S = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$ – формула Герона.

Б2. Логічне програмування

Лабораторна робота № 7
Тема: Робота в середовищі Турбо – Пролог

Завдання:

Написати та ввести в Пролог-систему програму, яка б у першому рядку роздруковувала прізвище, ім'я та по батькові студента, а в другій – його рік, місяць та число народження.

Лабораторна робота № 8
Тема: Введення фактів і правил на Пролозі

Завдання:

1. Створити базу даних мовою Пролог про свою родину: батьків, сестер, братів, дядьків, тіток, дідусів і бабусь.

2. Записати мовою Пролог правила визначення предикатів: бути_сестрою, бути_братом, бути_дядьком, бути_тіткою, бути_дідусем, бути_бабусею.

3. Сформулювати запити про кількість ваших сестер і братів та наявність родичів.

4. Ввести програму в комп'ютер та перевірити її роботоздатність.

Лабораторна робота № 9
Тема: Програма підтримки діалогу на англійській мові з використанням рекурсивно означуваних предикатів

Завдання:

1. Написати алгоритм логічної програми для підтримки діалогу англійською мовою для заданого набору речень згідно варіанту, вибравши для подання речень списокову структуру даних.

3. Перевірити роботоздатність програми, ввівши її в комп'ютер.

4. Сформулювати всі можливі запити до програми та отримати всі можливі варіанти відповідей.

Варіанти завдань:

1. а) Питання: **do you speak english?** Відповідь**:no, i speak german.**

б) Питання: **do you go to school?** Відповідь **no,** :**i steady at institute.**

2. а) Питання: **you are a student.** Відповідь: **no, i am a teacher.**

б) Питання: **do you speak german ?** Відповідь:**yes**, **i speak german.**

3. **а)** Питання: **have you a sister?** Відповідь:**no, i have a brazier.**
   б) Питання: **where do you live?** Відповідь **i live in ukraine**.
4. **а)** Питання: **how old are you?** Відповідь: і **am twenty.**
5. **а)** Питання: **you are a computer.** Відповідь: **yes, i am a teacher.**
   б) Питання: **do you like read a book?** Відповідь:**no**, **i like to solve an algebra equation.**

Лабораторна робота № 10
Тема: Програма користування бібліотечним каталогом

Завдання:

1. Написати та ввести в комп'ютер програму надання послуг читачам, передбачаючи додаткові послуги читачам, які своєчасно повертали всі книжки і навчаються на відмінно.
3. Власне ім'я ввести в базу даних програми відповідно до стану своєї успішності.
4. Відлагодити програму та запустити на виконання.
5. Отримати дані про всі власні неповернуті книги та можливі послуги для себе.

Додаток В

Термінологічний словник термінів та означень

Абстрактна мова. Сукупність використовуваних у мові символів і правил їх застосування. Твердження даною абстрактною мовою означає, що є якесь речення (аналітичний вираз, формула), побудоване за граматичними правилами даної мови. Це твердження, формула може містити варіаційні змінні, так звані конституєнти, що тільки за певних умов роблять значення вислову істинним.

Алгоритм. Набір скінченного числа правил**,** що задають скінченну послідовність операцій, виконання яких забезпечує розв`язання задач певного типу. Особливості алгоритмів: 1) виконання за скінченне число кроків; 2) кожен крок алгоритму має бути точно визначеним; 3) наявність скінченного числа вхідних даних (можливо число даних рівне нулю) або визначення їх в процесі виконання алгоритму; 4) наявність вихідних даних; 5) ефективність, що забезпечується простотою його операторів, які виконуються за скінченний і прийнятний для користувача проміжок часу.

Аналітична модель. Модель, що складається із системи рівнянь, які можна розв'язати аналітичними, числовими, або комбінаційними методами.

Анонімна змінна. Змінна, яка використовується замість поіменованої змінної у тих твердженнях, коли її значення не має подальшого використання. У логічному програмуванні позначається за допомогою символу підкреслення "_".

Атом. Неподільний елемент мови програмування Лісп. Подається у вигляді однієї або сукупності літер, цифр, спеціальних знаків або їх комбінацій, які називаються відповідно символьними, числовими, спеціальними або змішаними.

Архітектура. Структура компонентів у програмі або системі зі всановленням взаємозв`язку між ними та принципів керування, що зумовлено наявністю певних синтаксичних конструкцій.

Атрибути. Властивості або характеристики об`єкта чи їх сукупності.

База даних (БД). Сукупність тверджень, що містять дані, факти.

База знань (БЗ). Динамічна база даних, яка постійно поповнюється, обновлюється та має вбудовані механізми виведення нових знань. Як правило, є складовою експертної системи.

Верифікація. Процес визначення відповідності моделі детальному концептуальному опису, який прийнятий розробником. Ступінь відповідності встановлюється з використанням прийнятих методів програмування.

Внутрішня ціль. Цільове твердження, що записано в програмі. на Пролозі.

Гіпотези. Наукові припущення, що зроблено для пояснення певних явищ, визначення невідомих закономірностей у системі чи об'єкті на основі наявної інформації. За відсутності інформації висувають гіпотези щодо можливих результатів, які потім перевіряють експериментально.

Гра. Спрощене відтворення реального процесу, що використовується для навчання, дослідження, прийняття рішень або розваг.

Детермінований алгоритм. Процес вирахувань, результатом якого є унікальний і передбачуваний результат при заданих вхідних даних.

Детермінований предикат. Предикат, для якого внутрішні уніфікаційні підпрограми можуть знайти тільки один розв'язок. У противному випадку предикат називається недетермінованим.

Динамічна база даних. База даних, яка може добавляти нові факти, коректувати їх та видаляти. Може розташовуватися на диску або в оперативній пам'яті.

Динамічна модель. Модель системи, в якій відбуваються зміни через виникнення подій у часі або рух об'єктів у просторі.

Дискретна модель. Математична або обчислювальна модель, вихідні змінні якої приймають тільки дискретні значення

Домен. Діапазон і тип значень, які визначені для базового типу значень. У Турбо-Пролозі базовими типами значень є *char, integer, real, string, symbol*.

Експертна система. Комп`ютерна система, що імітує побудову експертних оцінок людини-спеціаліста в деякій галузі знань. Має в наявності засоби опрацювання баз даних та засоби отримання логічних висновків з наявних фактів.

Експертна система, що грунтується на логіці. Експертна система, що організована як динамічна база даних, аналогічно як база даних Пролога.

Експертна система, що грунтується на правилах. Експертна система, що використовує правила для отримання висновків. Ці правила називають продукційними правилами.

Запит. Звертання до програми для отримання інформації з бази даних. Формується у вигляді речень або команди як внутрішня або зовнішня ціль

програми. У Турбо-Пролозі після опрацювання інформації, що записана в базі даних, завершується успіхом або неуспіхом

Зіставлення. Процес, який порівнює набір умов заданих у правилі з заданими умовами бази даних.

Змінна. Символічне ім`я. яке в Турбо Пролозі починається з великої букви або символу підкреслення "_".

Імітаційне моделювання. Метод конструювання моделі системи та проведення експериментів на моделі. Особливість методу – опис структури модельованої системи, застосування засобів відтворення поведінки системи на моделі, відображення властивостей середовища, в якому функціонує досліджувана система.

Інтелектуальна база даних. База даних, яка забезпечує для користувача діалог з комп`ютером, на мові близькій до природної і дозволяє заносити нові дані, модифікувати їх та видаляти.

Комп'ютерне моделювання. Реалізація процесу моделювання за допомогою комп`ютера. Особливість комп`ютерного моделювання – це можливість втручання користувача в процес моделювання та впливу на його результати.

Контекстно-вільна граматика. Модель конструювання речень, в якій зміст речення розпізнається безвідносно до контексту, в якому використовуються складові його слова.

Концептуальна модель. Абстрактна модель, яка виявляє причинно-наслідкові зв'язки, властиві досліджуваному об'єкту в межах, визначених цілями дослідження. Включає змістовний опис об'єкта, явища чи процесу, в якому відображені концепції користувача моделі і розробника.

Лямбда – вираз. Непоіменоване значення функції, яке вираховується в момент його виклику і більше не зберігається в пам`яті.

Ліспі (Lisp). Мова програмування,. Походить від скорочення слів (List processing – обробка списків). Структура мови грунтується на засобах опрацювання S-виразів.

Логіка предикатів. Наукова дисципліна, яка вивчає відношення між умовами та висновками.

Логічне програмування. Програмування мовою логічних висловлювань або тверджень, що зводиться до опису об`єктів та зв`язків між ними. Особливості мови – наявність потужного механізму повернення, який дозволяє погоджувати та перепогоджувати цільові твердження та механізму відсічки, який забезпечує припинення процесу пошуку альтернативних

розв’язків, що робить цю мову потужним засобом розроблення складних логічних програм.

Макрос. Програма, в якій реалізована ідея автоматичного динамічного програмування. У Ліспі за формою це визначення збігається з визначенням та викликом функції **DEFUN** без вирахування аргументів. На першому етапі вираховується тіло визначення (етап розширення), а на другому етапі вираховується форма отримана при розширені, що й повертається як остаточний результат.

Математична модель. Абстрактна модель, відображена у вигляді математичних виразів і відношень. При заданні відношень в аналітичній формі розв`язок знаходиться в явній формі відносно шуканих змінних як функції від параметрів моделі або в неявній формі, коли шукані змінні є функцією від одного або багатьох параметрів моделі. До моделей цього класу відносяться диференціальні, інтегральні, різницеві, імовірнісні моделі та інші. Якщо не можна отримати точний розв’язок моделі, то використовуються числові методи та методи комп`ютерного моделювання.

Мова програмування. Формально визначена мова, в якій кожен оператор має строго визначене значення. Застосовуються для опису алгоритму розв’язання тієї або іншої задачі.

Модель. Фізичне, математичне або інше логічне зображення системи, об`єкта або явища.

Неозначена змінна. Змінна, яка в даний момент не приймає ніякого значення. Застосовується у логічному програмуванні при погоджені цільових тверджень.

Правило. Твердження, яке встановлює зв’язки деякого факту з іншими фактами.

Предикат. Твердження про наявність зв’язку (відношення) між об’єктами за допомогою задання імені відношення та опису його доменів.

Програма. Метод вирахувань, виражений на мові програмування.

Продукційне правило. Правило, що використовується в експертній системі і грунтується на правилах. Це правило визначає чи задовольняє об`єкт опису, заданому в самому правилі.

Пролог (Prolog). Мова програмування. Походить від скорочення слів "Programming logic" (програмування за допомогою логіки). Структура мови базується на логіці вирахування предикатів.

Процес. Опис поведінки системи зі встановленням послідовності її станів та зв’язків між ними.

Рекурсія. Властивість структури або правила, що полягає у можливості повторного звертання до себе один, два або більше разів.

S-вираз. Атом або послідовність атомів, заключена в круглі дужки (список). Елементами списку, у свою чергу, можуть бути списки, які вкладені на довільну глибину. У функціональному програмуванні це єдиний спосіб подання даних та програм.

Синтаксичний аналіз. Метод опрацювання текстів, шляхом аналізу синтаксису речень.

Система. Набір компонентів, організованих для виконання певних (заданих або установлених) функцій.

Список. Множина впорядкованих об'єктів деяким логічним способом. Для позначення списку використовують елемент, який називається головою списку. За відсутності елементів список називається пустим. У функціональному програмуванні для його позначення використовують спеціальний атом **NIL** або **( ).** У логічному програмуванні пустий список позначається **[ ]**. Використовується для керування процесом, організації черг, формування компактних баз даних та знань.

Структурна схема (СС). Схема, в рамках якої програма чи структура подається у вигляді ієрархічного набору модулів або компонент структури.

Терми. Правильно побудовані вирази, значеннями яких є об'єкти та зв'язки між ними. Наприклад, назви об'єктів, компоненти речення.

Узагальнене правило рекурсії. Метод програмуванні мовою Пролог при якому використовується рекурсивне правило.

Уніфікація. Процес, який виконує спробу зіставити ціль із твердженнями бази даних.

Функтор. Ім'я складного відношення, яке встановлює зв'язки між термами.

Функціональне програмування.Спосіб побудови програм, в яких єдиною дією є виклик функції. Основні риса мови – простота логіки побудови, компактність транслятора, можливість побудови складних конструкцій на основі вкладання функцій на довільну глибину (застосування функціоналів) та самовикликів (рекурсій).

Функціонал. Функція вищого порядку, де в ролі аргументів виступають інші функції, причому композиція функцій може відбуватися на довільну глибину.

Функція. Відображення деякої множини *X*, яка є областю визначення на деяку множину *Y*, яка називається областю значень. У Ліспі в ролі множин

X і *Y* виступають так звані "атоми" та їх композиції "списки". Відображення задається функцією, яка визначається спеціальними виразами.

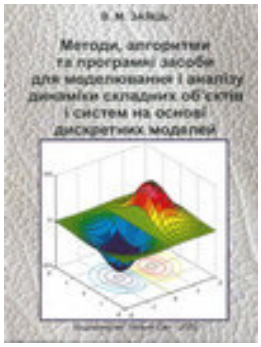
Ціль. Сукупність підцілей, які намагається задовольнити логічна програма. У Турбо-Пролозі буває зовнішньою або внутрішньою.

Часовий ряд. Реалізація випадкового процесу, яка є результатом спостереження за випадковим процесом стохастичної імітаційної моделі і фіксується за допомогою вимірів.

Штучний інтелект. Галузь знань, пов’язана з проектуванням комп’ютерних систем, які мають властивості, що асоціюються з розумною поведінкою людини.

Додаток Д

Перелік основних видань авторів (монографії, підручники, посібники)



З-40. 004.4

Заяць, В. М. Методи, алгоритми та програмні засоби для моделювання і аналізу динаміки складних об'єктів і систем на основі дискретних моделей : [монографія] / В. М. Заяць. – Львів : Новий Світ-2000, 2009. – 400 с. – (Вища освіта в Україні). - ISBN 978-966-418-087-7. Ціна 130 грн.

У монографії запропоновано нові підходи до побудови моделей динамічних систем складної природи, описано алгоритми їх реалізації та розроблено відповідне їх програмне забезпечення.

Заяць В. М.
Код: 978-617-607-411-3
Навчальний посібник. Львів : Видавництво Львівської політехніки, 2013. 144 с. Формат 170 х 215 мм. М'яка обкладинка. Ціна: 65.00 грн. Weight: 0 кг

Викладено основні поняття, методи та засоби розроблення комп'ютерних інформаційних технологій (КІТ) за допомогою комп'ютерної техніки, які мають широке прикладне застосування.

Описано відомі та запропоновано нові підходи до оцінки кількості інформації. Велику увагу звернуто на методи аналізу широкого класу сигналів як основних носіїв інформації та підходи для їх ефективного опрацювання. Наведено комп'ютерну технологію автоматизованого проектування складних об'єктів у середовищі Pspice та визначено особливості створення систем штучного інтелекту як основного стимулу для розроблення КІТ.

Для студентів, що навчаються за напрямом "Комп'ютерні науки" та "Комп'ютерна інженерія", а також для фахівців у галузі розроблення КІТ стосовно прикладних задач штучного інтелекту.

519.6(07) З-40

Заяць, В. М. Числові методи аналізу лінійних та нелінійних систем / В. М. Заяць, Б. В. Дурняк, І. М. Яворський ; МОН України, УАД, Львів, держ. ін-т новітніх технологій та управління ім. В. Чорновола. – Львів : Вид-во УАД 2009. – 236 с.

Номер ISBN: 966-8450-00-0 Ціна 85 грн.

Заяць, В.М. Функційне програмування / В. М. Заяць / ID: МВ-26456, Львів: Видавництво: Бескид Біт, 2002, 160 с. Номер ISBN: 966-8450-00-0 Ціна 45 грн.

У навчальному посібнику висвітлені основи функційного програмування, принципи та найбільш ефективні методи і алгоритми побудови функційних програм.

(0362) 63-57-31. Заяць, В.М. Дискретні моделі коливних систем для аналізу їх динаміки [Текст] : монографія / В. М. Заяць ; рец.: О. В. Кириленко, Г. Ф. Кривуля, А. Г. Руткас, М. С. Сявавко, І. М. Яворський. — Львів : Укр. акад. друкарства, 2011. — 284 с. — ISBN 978-966-322-231-8. Ціна 95 грн.

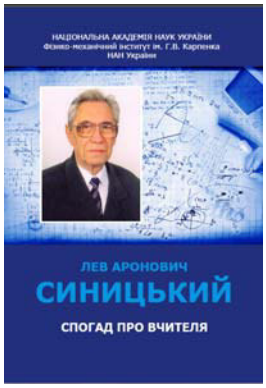
У монографії наведено основні підходи до побудови дискретних моделей коливних систем із високою добротністю і тривалими перехідними процесами, розроблено методи аналізу їх динаміки, встановлено необхідні та достатні умови стійкості та побудовано області синхронізації можливих динамічних режимів при дії зовнішнього збурення.

Запропоновано методики оцінки похибки обчислень під час проведення комп'ютерного моделювання і втрат потужності генераторних схем при зміні режиму роботи.

Розроблено методи пришвидшеного пошуку усталених режимів у системах коливної природи з тривалими перехідними процесами та високою добротністю.

Показано можливість аналізу коливних систем, що описуються диференційними рівняннями високих порядків до аналіз у набору моделей другого порядку за певних умов. Основні наукові результати, викладені в монографії, апробовано на реальних об'єктах коливної природи та системах зі складною динамікою.

Отримані результати мають теоретичне і прикладне значення для проектування коливних систем із бажаними за точністю і якістю характеристиками та достовірної ідентифікації об'єктів зі складною динамікою.



УДК 621.372.061
ББК Ч 484 (4УКР-4Льв-2л)
З 408. Заяць В.М. Лев Аронович Синицький. Спогад про вчителя / Автор–упорядник: В. М. Заяць, д.т.н, професор, академік АН прикладної радіоелектроніки.– Київ: Видавництво НАН України, 2013.№ ISBN 978-966-02-6713-– 172 с. Ціна 65 грн.

У виданні відображено основні етапи життя, наукову, педагогічну та громадянську позицію заслуженого професора Львівського НаціональногоУніверситету ім. Івана Франка, заслуженого діяча науки і техніки України, професора, доктора технічних наук в царині теоретичної електрорадіотехніка, академіка АН Прикладної Радіоелектроніки Лева Ароновича Синицького.

УДК 681.142.2; 004.432.42 (075.8)

ББК 32.9732 З-411

З-411. Заяць В.М., Заяць М.М. Логічне і функціональне програмування: Навчальний посібник. Гриф надано МОН України – Кам'янець-Подільський : ФОП Гордукова І. Є., 2016. – 400 с. ISBN 978-617-7381-38-8. Вага 700 грам. Ціна 120 грн.

У підручнику висвітлено теоретичні основи і програмні засоби логічного та функціонального програмування, найефективніші методи і алгоритми побудови програм мовою логіки та в термінах функціоналів, принципи реалізації найчастіше вживаних логічних та функціональних прикладних програм.

Призначений для студентів, які навчаються за спеціальностями галузі знань 12 "Інформаційні технології", а також може бути корисним для фахівців у галузі розроблення та застосування декларативних мов програмування стосовно прикладних задач штучного інтелекту. © Заяць В.М., Заяць М.М., 2016

ББК 32.817; 32.841; 32.9732
З – 412. УДК 681.142.2 ; 004.432.42 (075.8)
Заяць В. М.
З -412 Методи і засоби комп'ютерних інформаційних технологій в прикладних застосуваннях (МЗКІТ ПЗ) : підручник / В. М. Заяць. – Львів : НВФ «Українські технології», ISBN 978-966-345-315-6, 2017. – 260 с.
Вага 500 грам. Ціна 90 грн.

Викладено основні поняття, методи та засоби розроблення комп'ютерних інформаційних технологій (КІТ) засобами комп'ютерної техніки, що має широке прикладне застосування.

Описано відомі та запропоновано нові підходи до оцінки кількості інформації. Значну увагу звернуто на методи аналізу широкого класу сигналів як основних носіїв інформації та розглянуто підходи до їх ефективного опрацювання. Наведено комп'ютерну технологію автоматизованого проектування складних об'єктів у середовищі Pspice та визначено особливості створення систем штучного інтелекту як основного стимулу для розроблення КІТ.

Сформовано вимоги до виконання курсових проектів та лабораторних робіт. Наведено зразки виконаних курсових проектів. У додатку наведено основні визначення.

Призначений для студентів, що навчаються за напрямом «Комп'ютерні науки», «Комп'ютерна інженерія», «Програмна інженерія», а також може бути корисним для фахівців у галузі розроблення КІТ стосовно прикладних задач штучного інтелекту та інформаційних технологій.

УДК 681.5
ББК 32 965
К 89

«Комп'ютерне моделювання та програмне забезпечення інформаційних систем і технологій»: збірник наукових праць (тези доповідей і вибрані статті) III Всеукраїнської науково-практичної конф., м. Рівне, 28 вересня – 30 вересня 2017 р. – Рівне: Національний університет водного господарства та природокористування, 2017. – 234 с.
ISBN 978-966-345-318-7

НАЦІОНАЛЬНЫЙ УНИВЕРСИТЕТ ВОДНОГО ХОЗЯЙСТВА И ПРИРОДОИСПОЛЬЗОВАНИЯ (Украина, Ровно)
МЕЖДУНАРОДНАЯ АКАДЕМИЯ НАУК ПРИКЛАДНОЙ РАДИОЭЛЕКТРОНИКИ (Украина, Харьков)
УНИВЕРСИТЕТ НАУКИ и ТЕХНОЛОГИЙ (Польша, Бидгощ)

III Всеукраинская научно-практическая конференция
«КОМПЬЮТЕРНОЕ МОДЕЛИРОВАНИЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ И ТЕХНОЛОГИЙ» (КМПО-2017),
посвященная памяти профессора, доктора технических наук,
заслуженного деятеля науки и техники Украины, заслуженного профессора
ЛНУ им. И. Франко Синицкого Льва Ароновича

28–30 сентября 2017, г. Ровно, Украина

NATIONAL UNIVERSITY of WATER AND ENVIRONMENTAL ENGINEERING (Ukraine, Rivne)
INTERNATIONAL ACADEMY OF SCIENCES APPLIED ELECTRONICS (Ukraine, Kharkow)
UNIVERSITY of TEHNOLOGY and SCIENCE (Poland, Bydgozszc)

Triad National Scientific Conference
«COMPUTER SIMULATION AND SOFTWARE OF INFORMATION SYSTEMS AND TECHNOLOGY» (CSS-2017),
dedicated to the memory of Professor, D.Sc,
Honored Scientist of Ukraine LNU I. Franko Professor Emeritus Lev A. Sinitsky

September 28–30, 2017, Rivne, Ukraina

MIĘDZYNARODOWA AKADEMIA NAUK RADIOELEKTRONIKI STOSOWANEJ
NARODOWY UNIWERSYTET INŻYNIERII OCHRONY ŚRODOWISKA I WODY
UNIWERSYTET TECHNOLOGICZNO-PRYRODNICZY (Poland, Bydgoszcz)

III KRAJOWA KONFERENCJA NAUKOWA
«KOMPUTEROWA SYMULACJA, OPROGRAMOWANIE SYSTEMÓW INFORMACYJNYCH I TECHNOLOGII» (KSO-2017)
poświęcone pamięci profesora, dsc. Synytskiego Leva Aronovycha

28–30 września 2017, Równe, Ukraina

Організаційний комітет конференції:

Заяць Василь Михайлович (співголова) – д.т.н., професор, президент Львівського відділення Міжнародної академії наук прикладної радіо- електроніки, академік МАН ПРЕ (Україна, НУВГП, м. Рівне);

Тадеєв Петро Олександрович (співголова) – д.пед.н., професор, директор Інституту АКОТ (Україна, НУВГП, м. Рівне);

Круліковський Борис Борисович (заступник голови) – к.ф.-м.н., доцент, завідувач кафедри ОТ (Україна, НУВГП, м. Рівне);

Зубик Людмила Володимирівна (вчений секретар) – к.пед.н., ст. викл. (Україна, НУВГП, м. Рівне);

Василець Святослав Володимирович – д.т.н., доцент (Україна, НУВГП, м. Рівне);

Древецький Володимир Володимирович – д.т.н., професор (Україна, НУВГП, м. Рівне);

Мартинюк Петро Миколайович – д.т.н., доцент (Україна, НУВГП, м. Рівне);

Сафоник Андрій Петрович – д.т.н., доцент (Україна, НУВГП, м. Рівне);

Тулашвілі Юрій Йосопович – д.т.н., професор (Україна, НУВГП, м. Рівне);

Турбал Юрій Васильович – д.т.н., доцент (Україна, НУВГП, м. Рівне);

Яворський Ігор Миколайович – д.ф.-м.н., професор, академік МАН ПРЕ (Україна, ФМІ НАНУ ім. Г. В. Карпенка, м. Львів).

Секретаріат конференції:

Назарук Віталій – к.т.н., старший викладач кафедри обчислювальної техніки Національного університету водного господарства та природокористування (керівник секретаріату);

Zakrzewski Zbigniew – dr inż., ад'юнкт Університету Природничо-Технологічного, м. Бидгощ, Польща (за згодою);

Заяць Антоніна – системний програміст з тестування якості програмних продуктів фірми SoftServe, м. Львів (за згодою);

Момоток Галина – старший інженер кафедри обчислювальної техніки Національного університету водного господарства та природокористування;

Шатний Сергій – старший викладач кафедри обчислювальної техніки Національного університету водного господарства та природокористування.

Науковий координатор конференції д.т.н., професор Заяць В. М.

III Всеукраїнська науково-практична конференція

КОМП'ЮТЕРНЕ МОДЕЛЮВАННЯ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ І ТЕХНОЛОГІЙ КМПЗ – 2017

ЗБІРНИК НАУКОВИХ ПРАЦЬ
(*ТЕЗИ ДОПОВІДЕЙ І ВИБРАНІ СТАТТІ*)
Рівне, 28 вересня – 30 вересня 2017 р.
Науковий редактор і відповідальний за випуск В. М. Заяць

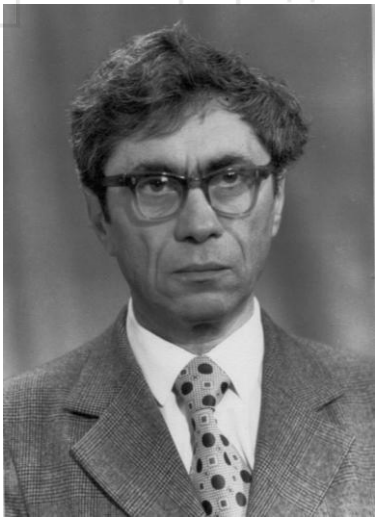
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ
МІЖНАРОДНА АКАДЕМІЯ НАУК ПРИКЛАДНОЇ РАДІОЕЛЕКТРОНІКИ

III Всеукраїнська науково-практична конференція

«КОМП'ЮТЕРНЕ МОДЕЛЮВАННЯ ТА ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ІНФОРМАЦІЙНИХ СИСТЕМ І ТЕХНОЛОГІЙ» (КМПЗ-2017)

присвячена пам'яті професора, доктора технічних наук, заслуженого діяча науки і техніки України, заслуженого професора ЛНУ імені Івана Франка Синицького Лева Ароновича

за підтримки: ТзОВ «Техніка для бізнесу»



Рівне
28 вересня – 30 вересня 2017 р.

Навчальне видання

Заяць Василь Михайлович
Заяць Марія Михайлівна

ЛОГІЧНЕ І ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ. СИСТЕМНИЙ ПІДХІД

Підручник

Друкується в авторській редакції

Технічний редактор *Галина Сімчук*
Коректор *Антоніна Заяць*
Комп'ютерна верстка *Марія Заяць*

Підписано до друку 28.04.2017 р. Формат 60×84 $^{1}/_{16}$.
Ум.-друк. арк. 24,5. Обл.-вид. арк. 27,3.
Тираж 500 прим. Зам. № 5347.

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

Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру
видавців, виготівників і розповсюджувачів видавничої продукції
РВ № 31 від 26.04.2005 р.