МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування
імені адмірала Макарова

В. К. ПАРТАС, О. В. ГАЙДАЄНКО

Функціональне та логічне програмування

Методичні вказівки для студентів напрямку "Комп'ютерні науки"

Рекомендовано Методичною радою НУК

Миколаїв 2009

УДК 681.3.06

Партас В.К., Гайдаєнко О.В. Функціональне та логічне програмування: Методичні вказівки для студентів напрямку "Комп'ютерні науки". – Миколаїв: НУК, 2009 – 60 с.

Кафедра інформаційних управляючих систем і технологій

Методичні вказівки містять завдання та теоретичні відомості, необхідні для виконання лабораторних робіт з курсу "Функціональне та логічне програмування". Вони можуть бути корисними також при роботі над завданнями курсових робіт, самостійному вивченні методів декларативного програмування.

Призначені для студентів усіх форм навчання напрямку "Комп'ютерні науки".

Рецензент канд. техн. наук, доцент Т.Г. Григорян

Згідно з наказом ректора НУК №8 від 09.01.2008 методичні вказівки публікуються в авторській редакції і відповідальність за їх редагування несе автор.

Лабораторная робота № 1

ОБРОБКА СПИСКІВ У МОВІ LISP

Ціль роботи: ознайомлення зі структурою списків, одержання практичних навичок з перетворення й конструювання списків

Завдання: написати λ-вираз для перетворення списку (a b c d e), який складається з п'яти елементів, до виду, що відповідає варіанту (табл. 1.1.).

Таблиця 1.1. **Варіанти завдань**

1	`((a . b) c (d . e))`	11	`(d (a . e) c . b)`
2	`(b a (c d) . e)`	12	`(((d . a) . b) e c)`
3	`((c b a) (e . d))`	13	`(((d . a) . b) (e c))`
4	`((d . e) b a c)`	14	`((d . a) b (e c))`
5	`(((b a) c) d . e)`	15	`((b (c (d (a . e)))))`
6	`((b a) c d . e)`	16	`((b . c) ((d . a) e))`
7	`((b a) c (d . e))`	17	`((b . c) (d a) . e)`
8	`(a (b . c) (d e))`	18	`(a e (d b c))`
9	`((d e) a b . c)`	19	`((a e) (d b).c)`
10	`((d e) a (b . c))`	20	`(((d b) . a) c.e)`

ТЕОРЕТИЧНІ ВІДОМОСТІ

Атоми. Основу мови LISP становлять символьні вирази, які називаються **S-виразами** й утворять область визначення для функціональних

програм. S-вирази являють собою послідовність деяких об'єктів, об'єднаних у єдину структуру [5]. Прикладами можуть служити наступні вирази:

```
(ПАВЛО СТАСЮК 33 РОКИ)
((МАША 21) (ВАСЯ 24) (ПЕТРО 1))
```

Перший S-вираз складається із простих об'єктів, названих атомами. Другий вираз складається із трьох складених об'єктів, кожен з яких складається із двох атомів.

Атом вважається найпростішим S-виразом, з якого складаються всі інші структури.

У мові Lisp розрізняють атоми двох типів: **символьні** й **числові**.

Символьні атоми можуть складатися з букв і цифр, а також будь-яких спеціальних знаків[1], у тому числі й знака "пробіл". При використанні деяких спеціальних знаків (питальний і знак оклику, тильда, пробіл, закрита й відкрита дужки, зворотна похила риса, вертикальна риса) варто обмежувати символ із двох сторін вертикальною рисою (|, bar) або перед кожним спеціальним знаком ставиться зворотна коса риса (\, back slash).

Символьні атоми мають більш широке поняття, чим змінна. Символьний атом у загальному випадку позначає будь-який об'єкт, сутність. У мові Lisp символами представляють числа, функції, структури або інші символи. Символ може існувати й використовуватися в програмі, незважаючи на те, що він не має значення. Приклади символів представлені в табл. 1.2.

Таблиця 1.2. **Приклади символів у мові Lisp**

Символ	Пояснення
ABC	Звичайний символ, що складається із трьох прописних букв
|5432g6|	Символ обмежений вертикальною парою із двох сторін, тому що він починається із цифри і Lisp може розділити його на два атоми
G54326	Звичайний символ, тому що починається з букви
+	Символ складається з одного спеціального знака
|~?@\\|(% – це теж символ !!|	Символ, який складається з послідовності спеціальних знаків, пробілів і букв. Обмежений із двох сторін знаком "вертикальна риса"

Якщо атом представлений тільки послідовністю цифр, то це числовий атом, або просте число (табл. 1.3.). В основному правила представ-

[1] У мові Lisp поняття "знак" замінює поняття "символ" в інших мовах програмування у зв'язку с тим, що у мові Lisp слово "символ" використовується як назва символьного атому.

лення чисел не відрізняються від інших мов програмування. У мові Lisp додатково застосовуються дробі – числа, представлені у вигляді двох чисел, розділених похилою рисою. Приклади чисел наведені нижче.

Таблиця 1.3. **Приклади числових атомів**

Число	Пояснення
15	Ціле число
15.0	Дійсне число
–3.28E+5	Від'ємне дійсне число, що представлене мантисою й порядком
4/7	Дробове число

Списки. *Поняття списку.* Під списком розуміється послідовність, елементами якого є атоми й списки. Списки знаходяться у круглих дужках, а елементи списку розділяються пробілами. Приклади списків представлені в табл. 1.4.

Таблиця 1.4. **Приклади списків у мові Lisp**

Список	Пояснення
`(a b c)`	Список, що складається із трьох елементів – символьних атомів
`((ab 2) (Foo 3) 2)`	Список містить три елемента, перші два з них – списки
`((a (b (c (d)))))`	Список, що складається з одного елемента – багаторівневий список
`()`	Список не містить жодного елемента. Такий список називається пустим

У мові Lisp можна побудувати список без елементів – пустий список. Такий список позначається атомом NIL.

Таким чином, виходячи з вище сказаного, список являє собою ієрархічну структуру даних. При цьому відкриваючі й закриваючі дужки повинні бути в строгій відповідності для точної вказівки довжини списків. Необхідно відмітити, що у мові Lisp список використовується як для зберігання даних, також і для представлення програм [7].

Внутрішнє представлення списків. У пам'яті список являє собою бінарну структуру, яка називається осередком списку або cons-осередком [7]. Кожне із двох полів структури містить покажчик (pointer), що може посилатися на іншу таку ж структуру або на атом. Графічно структуру списку можна представити у вигляді прямокутника, розділеного на два поля (рис.1.1). Покажчики представляються стрілками. Перше поле вказує на частину списку, названою головою і являє собою перший еле-

мент списку. Друге поле вказує на частину списку, яка називається хвостом і являє собою всі інші, крім першого, елементи списку.

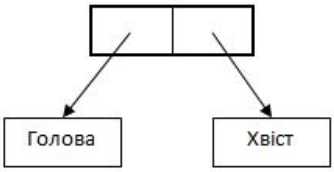


Рис. 1.1. Структура списку

Структура списку з трьох елементів `(a (b c) d)` представлена на рис. 1.2.

Очевидно, що хвостом списку `(c)` або `(d)` є пустий список (або NIL).

Крапкова пара. Якщо хвостом списку є атом, тоді такий список називається крапковою парою й представляється у вигляді `(H . T)`, де *H* – голова, *T* – хвіст списку [7, 8].

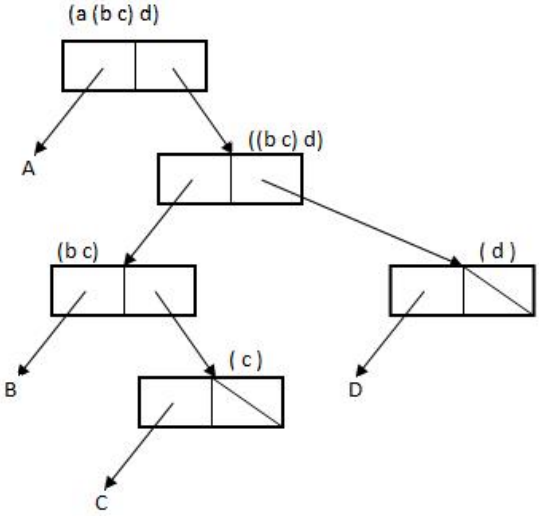


Рис. 1.2. Представлення списку в пам'яті

У мові Lisp крапкова нотація використовується для узагальненого подання будь-якого списку. Так, список `(a b c d e)` у крапковій нотації представляється у вигляді

```
(a . (b . (c . (d . nil))))
```

Для більш компактного представлення списків прийняті наступні правила скорочення:

– послідовність ". NIL" (крапка NIL) можна пропускати;

– послідовність знаків ". (" (крапка – відкрита дужка) і відповідну закриту дужку можна скоротити.

Функції. *Поняття функції.* У функціональному програмуванні функція має такий же зміст як і в математиці – відображення об'єктів з області визначення на об'єкти з області значень функції [5]. Наприклад, вираз

```
z = f (x, y)
```

означає, що пари значень **x** і **y** відображаються на єдине значення **z**.

Величини **x**, **y** називаються аргументами функції, **z** – значення, що повертає функція **f**. Виклик функції означає застосування визначення функції до конкретних значень аргументів.

У мові Lisp виклик функції записується в префіксній нотації, так само, як прийнято в математиці [6, 7]. При цьому використовується однакова форма запису у вигляді списку (табл. 1.5). Арифметичні вирази також записуються в префіксній нотації, тому що вони представляються як виклик відповідних функцій.

Таблиця 1.5. **Приклади виклику функцій у мові Lisp**

Вираз	Пояснення
`(f x y)`	Виклик функції **F** з аргументами **X** і **Y**
`(+ x y)`	Обчислення суми величин **X** і **Y**
`(+ (* x x) (* 2 x y) (* y y) )`	Обчислення квадрата суми величин **X** і **Y**

Блокування виклику функції. Через однакову нотацію подання виклику функцій і даних у вигляді списку в мові Lisp використається спеціальна функція QUOTE, що блокує обчислення [5, 7, 8] виразу й представляє його як дані. Наприклад, виклик функції

```
(quote (+ 5 x))
```

повертає список (+ 5 x), і, таким чином, фактично блокується обчислення суми числа 5 і величини x. Для скорочення запису виклик функції quote можна замінити знаком "'" (апостроф). Тоді попередній виклик функції буде виглядати в такий спосіб

```
'(+ 5 x)
```

Функція `quote` використовується також стосовно символьних атомів. Це дає можливість використати символ таким, яким він є, а не його значення.

Функції селектори. Селектори дозволяють одержати одну з частин списку. Функція CAR використається для одержання голови списку, а функція CDR – його хвоста. Ці функції мають один аргумент – список. У табл. 1.6. наведені приклади застосування функцій селекторів.

Таблиця 1.6. **Приклади застосування функцій селекторів**

Виклик функції	Результат, що повертає	Пояснення
`(car '(a b c))`	`A`	Повертає голову списку – атом **a**
`(car '((a b) c))`	`(a b)`	Повертає голову списку – підсписок **(a b)**
`(car '(a.b))`	`A`	Повертає голову крапкової пари – атом **a**
`(cdr '(a b c))`	`(b c)`	Повертає хвіст списку – підсписок **(b c)**
`(cdr '((a b) c))`	`(c)`	Повертає хвіст списку – підсписок **(c)**
`(cdr '((a b c)))`	`nil`	Повертає nil, тому що список містить тільки один елемент (голову **(a b c))**
`(cdr '(a . b))`	`B`	Повертає хвіст крапкової пари – атом **b**

Для одержання будь-якого елемента списку можна використати вкладений виклик функцій селекторів. Наприклад, для одержання третього елемента списку `(a b c d e)`, варто виконати наступну послідовність дій: отримати хвіст списку `(b c d e)`; від отриманого списку одержати хвіст `(c d e)`; одержати голову списку **c**.

Ці дії можна записати одним виразом

```
(car (cdr (cdr '(a b c d e))))
```

Для скорочення подібних записів у мові Lisp зарезервований ряд функцій, що передбачають вибірку елементів списку в межах перших п'яти елементів. Імена цих функцій відповідають діям які виконуються і складаються за наступним шаблоном

C<букви>R,

де <букви> – послідовність букв A і/або D, залежно від послідовності дій. В табл. 1.7. наведені приклади подібних функцій.

Таблиця 1.7. **Приклади використання додаткових селекторів**

Назва функції	Дії, що виконуються	Вираз	Результат
`CADR`	Одержати другий елемент списку (голова хвоста)	`(cadr '(a b c))`	`B`

Продовж. табл. 1.7

Назва функції	Дії, що виконуються	Вираз	Результат
CDDAR	Одержати хвіст хвоста голови списку. Передбачається, що першим елементом списку є список.	(cddar '((1 2 3) (a b c) d))	(3)
CDDDDR	Одержати список, починаючи з п'ятого елемента списку	(cddddr '(a b c d e))	(e)

Конструктор списку. Для створення нового списку використовується функція CONS, що має два аргументи. Функція конструює новий список, у якому перший аргумент є головою, а хвостом – другий аргумент. Інакше кажучи, функція CONS створює cons-осередок і встановлює в її полях покажчики на перший і другий аргументи відповідно. У табл. 1.8. наведені приклади використання конструктора списку.

Таблиця 1.8. **Приклади застосування конструктору списків**

Вираз	Результат	Пояснення
(cons 'a '(b c))	(a b c)	Створення нового списку, у якому атом A стає головою, а список (b c) – хвостом списку.
(cons '(a b) nil)	((a b))	Створення списку з одного елемента – список (a b).
(cons (cdr '(3 8)) 'a)	((8) . a)	Першим аргументом є виклик функції CDR, що повертає список (8). Другим аргументом є атом A. Функція CONS створює крапкову пару, головою якої є список (8), а хвостом – атом A

МЕТОДИЧНІ ВКАЗІВКИ

Основні принципи конструювання списків. При виконанні завдання необхідно використовувати лише базові функції обробки списків: селектори CAR і CDR, додаткові селектори та конструктор CONS. Таке обмеження потрібно для більш глибокого засвоєння способів обробки списків, що необхідно для успішного виконання інших робіт.

Слід звернути увагу на те, що при створенні списку конструктор додає голову до хвоста, тобто створення списку починається з останнього елементу і поступово додаються нові елементи в голову списку. Наприклад, необхідно з атомів A, B і списку `(C D)` створити список `(B (C D) A)`.

Процес створення списку представимо з наступних кроків:

- додавання атому A до пустого списку – створення списку `(A)`;
- додавання списку `(C D)` до списку `(A)` – створення списку `((C D) A)`;
- додавання атому B до списку `((C D) A)` – створення списку `(B (C D) A)`.

При виконанні завдання процес конструювання нового списку супроводжується вибором атомів з вхідного списку. Для вибору атомів слід використовувати наступні функції – селектори:

`CAR`	перший атом
`CADR`	другий атом
`CADDR`	третій атом
`CADDDR`	четвертий атом
`CDDDDR`	хвіст, починаючи з п'ятого атому.

Таким чином, для одержання п'ятого елементу списку необхідно отримати хвіст, починаючи з п'ятого елементу, а потім одержати голову списку.

Приклад виконання завдання. Перетворити список `(a b c d e)` у список `((b c d) (e . a))`.

Рішення. Список, який потрібно створити, складається з двох елементів – список ((b c d) і список (e . a). Виклик конструктора у даному випадку має вигляд:

`(CONS <вираз1> <вираз2>)`,

де:

`<вираз1>` – вираз для створення списку `(b c d)` – голова списку;

`<вираз2>` – вираз для створення списку `((e . a))` – хвіст списку.

Припускаючи, що **X** – вхідний список, розглянемо кроки процесу конструювання списку `((e . a))`.

- Отримання атому A – `(car X)`;
- Отримання атому E – `(car (cddddr X))`;
- Створення крапкової пари – `(cons (car (cddddr X)) (car X))`;

• Створення списку `((e . a))` –
`(cons (cons (car (cddddr X)) (car X)) nil).`

Запишемо вираз для створення списку `(b c d)`:

```
(cons (cadr X) (cons (caddr X) (cons (cadddr X) nil)))
```

Остаточно запишемо лямбда-виклик для перетворення списку:

```
(                ; початкова відкрита дужка лямбда-
виклику
(lambda (X)      ;початок визначення лямбда-виразу
(cons           ;конструктор списку ((b c d) (e . a))
                ;конструктор списку (b c d)
(cons (cadr X) (cons (caddr X) (cons (cadddr X)
nil)))
                ; конструктор списку ((e . a))
(cons (cons (car (cddddr X)) (car X)) nil)
))              ; кінцева дужка лямбда-виразу
'(a b c d e)    ; фактичний параметр
)               ; кінцева дужка лямбда-виклику
```

Контрольні запитання

- Визначення списку.
- Визначення символьного та числового атомів.
- Визначення лямбда-виразу.
- Визначення лямбда-виклику.
- Чим відрізняються формальні та фактичні параметри функції?
- Функції – селектори.
- Функції конструювання списків.

Лабораторна робота № 2

ВИЗНАЧЕННЯ ФУНКЦІЇ ТА РЕКУРСІЯ

Цілі роботи: одержання теоретичних знань та практичних навичок з програмуванням на основі функцій; одержання практичних навичок щодо розробки рекурсивних функцій.

Завдання: Скласти рекурсивну функцію для обробки списку з будь-якої кількості атомів, які складені з будь-яких знаків. Якщо у результаті

обробки з атому видалені всі знаки, замінити його на атом <ПУСТО>.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Таблиця 2.1. **Варіанти завдань**

№ п/п	Спосіб обробки
1	Упорядкувати цифри у кожному атомі списку (a53r2 → a23r5)
2	Розрахувати суму цифр у всьому списку (сума цифр списку ‘(a35f ab28 t1g) рівна 19)
3	У кожному атомі списку латинські літери змінити на знак + (плюс), літери кирилиці – на знак – (мінус).
4	Кожний атом списку будь-якої вкладеності замінити на кількість літер в атомі (a2e54 der5e) → (2 4)
5	У кожному атомі списку цифри розмістити у зворотному порядку (a53r2 → a23r5)
6	Розрахувати добуток цифр у всьому списку (добуток цифр списку ‘(a35f ab28 t1g) рівно 240)
7	Замінити кожну цифру у атомах списку на перші три літери їй назви. (a53r2 → адватригпят)
8	В кожному атомі списку замінити літери кирилиці замінити на знаки ~K~.
9	У всіх атомах списку літери англійського алфавіту перетворити у прописні.
10	У всіх атомах списку поміняти регістр літер англійського алфавіту на протилежний.
11	У всіх атомів списку видалити знак із зазначеним номером.
12	Кожний атом скласти з літер, знаку підкреслення та суми цифр (a1b5c → abc_6)
13	Кожний атом списку скласти з упорядкованих цифр, знаку підкреслення та літер
14	Кожний атом скласти з цифр, знаку підкреслення та літер англійського алфавіту.
15	У кожному списку замінити літери англійського алфавіту на відповідний номер у алфавіті (аф6Гs4 → 1ф6Г194)
16	Зі всіх атомів списку видалити зазначений знак
17	Розрахувати суму цифр у атомах вкладених підсписків
18	У всіх атомах списку видалити однакові знаки (знаки, що присутні в атомі більш одного разу).
19	Кожний атом класти з суми цифр, знаку підкреслення та кодів літер.
20	Кожний атом скласти з кількості цифр, знаку підкреслення, кількості літер, знаку підкреслення та кількості останніх знаків. (sa3+$4d4-r → 3_3_3)

Продовж. табл. 2.1

№ п/п	Спосіб обробки
21	Знаки кожного атома списку розташувати у вигляді послідовності цифр, а потім літер. Інші знаки видалити. (sa3+$4d4-r → 344sad)
22	Скласти список зі зазначеної кількості атомів, що складені зі зазначеної кількості знаків з використанням зазначеного знака.
23	У кожному атомі видалити одиночні знаки
24	У кожному атомі видалити цифри і літери, залишити тільки спеціальні знаки

Визначення функції. Визначення нової функції в мові Lisp виконується за допомогою функції DEFUN, яка має наступний формат:

```
(DEFUN <ім'я> <лямбда-список> <тіло функції>)
```

Як можна побачити, визначення функції подібне визначенням лямбда-виразу, за винятком того, що визначення функції передбачає обов'язкове завдання імені, яке використовується у подальшому для виклику функції.

Лямбда-вираз є контекстною функцією і знищується після виклику. У відмінність від цього функція DEFUN зв'язує символ (ім'я функції) з лямбда-виразом і зберігає цей зв'язок на весь сеанс роботи інтерпретатору. Для перевірки існування зв'язку символу з визначенням функції у мові Lisp служить предикатна функція FBOUNDP, а для отримання самого лямбда-виразу – функція SYMBOL-FUNCTION [7].

Наприклад, розглянемо визначення функції для розрахунку квадрату різниці двох величин:

```
(defun s2 (x y) (+ (- (* x x) (* 2 x y)) (* y y))),
```

де: `S2` – символ, що представляє ім'я функції, який функція DEFUN зв'язує з лямбда-виразом `(+ (- (* x x) (* 2 x y)) (* y y))`; `(x y)` – лямбда-список або список формальних параметрів функції.

Застосування функції `(fboundp 's2)` повертає значення `T` (істина), а виклик функції `(symbol-function 's2)` дозволяє отримати список

```
(+ (- (* x x) (* 2 x y)) (* y y)).
```

Для розрахунку квадрату різниці чисел 2 і 3 слід записати виклик функції S2 у вигляді `(s2 2 3)`.

Функції–предикати. Предикат – це функція, що визначає наявність деяких властивостей аргументу і повертає логічне значення NIL (хибне), або T (істина) [7].

У мові Lisp базовими предикатами є функції ATOM і EQ.

Функція ATOM перевіряє, якщо аргумент є атомом. В табл. 2.2. наведені низка прикладів застосування функції. Слід звернути увагу на те, що предикат ATOM сприймає пустий список як атом NIL.

Таблиця 2.2. **Приклади застосування базових предикатів ATOM и EQ**

Вираз	Результат	Пояснення
(atom ‘abc)	T	ABC є символьним атомом
(atom ‘(a b c))	NIL	(A B C) є списком
(atom nil)	T	NIL є атомом
(atom ‘())	T	Пустий список може бути представлений як атом NIL, тому функція повертає T
(atom ‘(nil))	NIL	Аргумент є список, що містить один елемент
(eq ‘a ‘b)	NIL	Аргументи – різні символи
(eq ‘a (car ‘(a b c))	T	Другий аргумент – виклик функції CAR, яка повертає голову списку – атом A
(eq nil (atom ‘(a b)))	T	Другий аргумент повертає значення NIL

Предикат EQ порівнює два символьних атома і повертає T, якщо вони ідентичні. Предикат можна використовувати тільки для символів та констант T и NIL. Функція не перевіряє логічну рівність аргументів, а тільки їх представлення у пам'яті, тому для чисел і списків можна отримати неадекватний результат. Для порівняння таких об'єктів використовуються інші функції.

Крім функцій ATOM і EQ у мові Lisp передбачено ще ряд предикатів, що широко використовуються при композиції програм. Найбільш відомі функції-предикати представлені в табл. 2.3. Застосування предикатів сприяє розробці більш зрозумілих та читабельних програм і відповідає принципам функціонального програмування [7, 8].

Таблиця 2.3. **Функції–предикати у мові Lisp**

Найменування	Призначення
LISTP	Перевіряє, чи є аргумент списком
CONSP	Перевірка структури списку (cons-осередку)

Продовж. табл. 2.3

Найменування	Призначення
NULL	Перевірка пустого списку
NUMBERP	Перевіряє, чи є аргумент числовим атомом
PLUSP	Перевірка числового атома на додатне значення
MINUSP	Перевірка числового атома на від'ємне значення
ZEROP	Перевірка числового атома на нуль
BOUNDP	Перевіряє, чи існує зв'язок символу з деяким значенням
FBOUNDP	Перевірка зв'язку символу з лямбда-виразом

Псевдофункції. У функціональному програмуванні функція повинна мати властивість математичної функціональності – виконання обчислень і повернення значення. Однак в програмуванні часто застосовуються функції, що виконують додаткові дії, крім обчислення виразу, тобто мають побічний ефект. Такі функції називаються псевдофункціями [7, 8].

У мові Lisp застосовується ряд псевдофункцій, що потрібні для написання практичних програм. Найбільш широко використовуються наступні функції: `SET`, `SETQ`, `SETF` – функції, що встановлюють зв'язок символу з деяким значенням. Така дія аналогічна операції присвоєння. Функції повертають значення, що зв'язано, а у якості побічного ефекту змінюють внутрішній стан символу; `READ` – функція читання з вхідного потоку. Повертає значення, що прочитано, а у якості побічного ефекту виконує операцію читання; `PRINT`, `PRIN1`, `PRINC`, `WRITE`, `FORMAT` – функції запису у вихідний потік. Повертає значення, що передано в потік, а у якості побічного ефекту виконує перетворення даних і вивід.

Додаткові матеріали про ці функції можна знайти в [7].

Рекурсивні функції. Застосування рекурсивних функцій у повній мірі відповідає функціональному стилю програмування, а точніше, головному принципу – композиція програм з викликів функцій без накопичення у змінних проміжних результатів [6]. Основне призначення рекурсивних функцій – виконання циклічних операцій.

Функція є рекурсивною, якщо її визначення містить виклик самої функції. Якщо деяка гілка програми містить один рекурсивний виклик, така рекурсія називається простою. При двох і більш рекурсивних викликах в одної гілці програми говорять про паралельну рекурсію. Якщо деяка функція F містить виклик другої функції G, а визначення функції G містить виклик функції F, то така організація функції називається взаємною рекурсією. Якщо рекурсивний виклик містить у якості фактичних

параметрів рекурсивний виклик, то це – рекурсія більш високого порядку [7].

Будь-яка рекурсивна функція містить щонайменше дві гілки: рекурсивну і термінальну. Термінальна гілка повертає значення, що заздалегідь відоме при деяких умовах. Рекурсивна гілка містить рекурсивний виклик функцій з фактичними параметрами, значення яких приведе у подальшому до термінальної гілки. Це є обов'язковою умовою завершення рекурсивного процесу і, відповідно, рекурсивної функції.

У якості прикладу розглянемо функцію розрахунку довжини списку. На підставі того, що існує лише можливість отримання доступу до голови (першого елементу) і хвосту списку, кількість елементів можна полічити як суму

$$L(s) = 1 + L(s_t), \qquad (2.1)$$

де: S – список; S_t – хвіст списку S; $L(s)$ – довжина списку S; $L(s_t)$ – довжина хвоста списку S, яка обчислюється по формулі 2.1.

Для функції $L(s)$ можна однозначно казати, що, якщо S – пустий список, то $L(s) = 0$.

Таким чином, попередні міркування можна записати на мові Lisp у наступному вигляді:

```
(defun LENLIST (x)     ; визначення функції (x – список)
    (if (NULL x)       ; якщо список пустий
          0            ; кількість елементів рівно(дорівнює)
                       ; (це термінальна гілка функції)
                       ; інакше здійснюється рекурсивний
                       ; виклик функції
          (+ 1 (LENLIST (cdr x)))
                       ; LENLIST для хвоста списку, до
                       ; результату
                       ; якого додається 1
     )                 ; заключна дужка функції IF
)                      ; заключна дужка визначення функції
```

МЕТОДИЧНІ ВКАЗІВКИ

Загальні вказівки щодо виконання завдання. Для рішення завдання відповідно до варіанту необхідно розробити рекурсивні функції обробки списку. У якості фактичного параметру може бути використаний

будь-який однорівневий список з довільної кількості атомів, кожний з яких може складатися з будь-яких припустимих знаків.

Виконання завдання варто почати з декомпозиції всієї задачі на більш дрібні підзадачі, які згодом будуть втілені в окремі прості функції. Обов'язково варто передбачити розробку функцій–предикатів, які можна використати при реалізації умов рекурсивного виклику в більш складних функціях.

Для роботи із символьними атомами їх варто перетворити в список знаків, а після відповідної обробки виконати зворотне перетворення в символ. Для цього можна застосувати функції, представлені нижче.

Розробка функцій повинна виконуватися в строгій відповідності із принципами функціонального програмування. Так, у розроблених функціях у даній лабораторній роботі повинні бути відсутні локальні та глобальні змінні, функції зв'язування, функції циклів. Визначення функцій повинні містити лише рекурсивні виклики, або виклики інших функцій.

Функції розгалужених операцій. Умовна пропозиція COND.

Функція COND є основним засобом розгалуження обчислень у мові Lisp. Функція має наступний формат:

$$(\mathrm{COND}\ (p_1\ a_1)\ (p_2\ a_2)\ \ldots\ (p_i\ a_i)\ \ldots(p_n\ a_n)\ )$$

де: p_i – будь-яка предикатна форма; A_i – форма, яка обчислюється у разі, якщо предикат p_i дорівнює T (істина) і повертається у якості результату функції COND.

Якщо предикат p_i дорівнює NIL (хибне), відповідна форма a_i не обчислюється і виконується перехід до наступного предикату. Якщо всі N предикатів хибні, функція COND повертає NIL.

Умовна пропозиція IF

Функція IF є спрощеною формою функції COND, й використовується у випадках коли існує лише одна альтернатива. Функція має наступний формат:

$$(\mathrm{IF}\ p\ a_1\ a_2)$$

де: p – предикатна форма; a_1 – форма, яка обчислюється у разі, якщо предикат p дорівнює T (істина) і повертається у якості результату функції IF; a_2 – форма, яка обчислюється у разі, якщо предикат p дорівнює NIL і повертається у якості результата функції IF.

Функції для роботи з ім'ям символу. Для отримання доступу до кожного знаку символа атома використовуються наступні функції перетворювання (табл. 2.4.):

Таблиця 2.4. **Функції перетворення даних**

Функція	Призначення	Пояснення
(string <symbol>)	перетворювання символа в рядок	symbol – символьний атом
(coerce <arg> <type>)	універсальна функція перетворювання	arg – об'єкт, що перетворюється type – тип, до якого здійснюється перетворення
(intern <string>)	створення символа з рядка	string – рядок
(char-code <mark>	визначення ASCII-коду знака	mark – знак
(code-char <int>)	визначення знака по ASCII-коду	int – ціле число

Таблиця 2.5. **Приклади застосування функцій перетворення даних**

Применение функции	Результат
(string 'data43b)	"DATA43B"
(string '|daTA43b|)	"daTA43b"
(coerce "daTA43b" 'list)	(#\d #\a #\T #\A #\4 #\3 #\b)
(coerce '(#\d #\a #\T #\A #\b) 'string)	"daTAb"
(intern "daTAb")	|daTAb|
(char-code #\3)	51
(code-char 97)	#\a

Приклад виконання завдання

Завдання: у кожному атому списку залишити лише парні цифри. Якщо у атомі не залишаються знаки, тоді слід замінити словом ПУСТО.

Рішення: розглянемо рішення задачі взагалі. Припустимо, що маємо функцію CONVERT, яка виконує перетворення атому відповідно до завдання. Тоді, для обробки всього списку X, запишемо рекурсивну функцію MAIN, яка перетворює голову списку, а потім створює новий список за допомогою конструктора CONS. У якості фактичних параметрів використаємо перетворену голову списку й рекурсивний виклик для перетворення хвоста:

```
(cons (convert (car X)) (main (cdr X)))   (2.1.)
```

Вираз (2.1.) використаємо як рекурсивну гілку функції. Термінальну гілку створюємо з умов, що перетворення пустого списку є пустий список. Таким чином, функцію MAIN запишемо у вигляді:

```
(defun main (X)
    (if ((null X)      ; якщо список X пустий
  nil                  ; функція повертає пустий
                       список
  (cons (convert (car X)) (main (cdr X)))
                       ; інакше створюється
                       ; новий список
    )
)
```

Дія функції CONVERT представимо з наступних кроків: перетворення атому до списку знаків; видалення зі списку всіх знаків, крім парних цифр; перетворення списку знаків, що залишились, до атому.

Перший крок можна реалізувати на основі функцій перетворення (табл. 2.1). Відповідна функція AtomToList має наступний вигляд:

```
(defun AtomToList (X) (coerce (string X) 'list)).
```

Аналогічно реалізуємо третій крок, однак при цьому необхідно повертати атом ПУСТО у разі, якщо список знаків пустий. Напишемо функцію ListToAtom, яка здійснює ці дії :

```
(defun ListToAtom (X)
(if (null x) 'ПУСТО
(intern (coerce X 'string))).
```

Реалізація другого кроку більш складна. По-перше, необхідна предикатна функція для визначення знаку, що є парною цифрою. Очевидно, що парну цифру можна визначити через ASCII код знаку. Тоді відповідну функцію напишемо таким чином:

```
(defun IsEven (x)
(and
(< 47 (char-code x) 58)   ; перевірка, чи є знак
                          ; цифрою
(evenp (char-code x))     ; перевірка, чи є код пар
                          ; ним числом
))
```

Далі розглянемо, яким чином можна перетворити список знаків, залишивши тільки парні цифри: якщо список пустий, тоді результатом перетворення – пустий список; якщо список не пустий, тоді розглянемо голову списку; якщо голова списку – парна цифра, тоді перетворюємо

хвіст і створюємо новий список, включаючи голову до перетвореного списку. Інакше, функція повинна повернути перетворений хвіст.

Напишемо функцію `ConvertList`, що перетворює список знаків відповідно вищевказаного:

```
(defun ConvertList (Lst)
(cond
   ; список пустий, функція повертає пустий
   ; список (термінальна гілка)
   ((null Lst) nil)
   ; якщо голова списку – парна цифра, створюємо
   ; новий список з голови списку та хвоста, який
   ; перетворюється за допомогою рекурсивного вик-
лику
   ((IsEven (car Lst)) (cons (car Lst)
                             (ConvertList (cdr Lst))))
   ; якщо голова списку не парна цифра –
   ; функція повертає лише перетворений хвіст
   (T (ConvertList (cdr Lst)))
   ))
```

Нарешті, запишемо функцію `Convert`, що складається з викликів вже розроблених функцій:

```
   (defun Convert (X)
       (ListToAtom (ConvertList (AtomToList X))))
```

Контрольні запитання

- Визначення функції.
- Визначення рекурсивної функції.
- Види рекурсивних функції.
- Псевдофункції.
- Функції – предикати.
- Умовні пропозиції у мові Lisp.

Лабораторна робота № 3

ФУНКЦІОНАЛИ

Цілі роботи: вивчення теоретичних основ застосування функцій з функціональним аргументом; одержання практичних навичок за програмуванням функцій більш високого порядку.

Завдання: На основі використання функціоналів розробити функцію відповідно варіанту завдання (табл. 3.1.).

Таблиця 3.1. **Варіанти завдань**

№ варіанту	Зміст завдання
1	Розробити функцію виду: (f '(atom null plusp) '(1 2 3)) → (T NIL T).
2	Розробити функцію виду: (f '(+ max min) '(1 2 3)) → (6 3 1).
3	Створити фільтр для літер кирилиці, латинських літер й цифр.
4	Обчислювати суми стовпців й рядків прямокутної матриці. Результат представити у вигляді списку сум.
5	З двох списків: (1 2 3 4 …) і (a b c d …) отримати один список виду: (a 1 b 2 c 3 ….).
6	Створити одиничну матрицю зазначеного розміру.
7	Розробити функцію для отримання об'єднання двох множин.
8	Розробити функцію для отримання перетину двох множин.
9	Розробити функцію для отримання різниці двох множин.
10	Видалити з атомів списку всі цифри.
11	Розробити функцію для табулювання функції двох аргументів.
12	Розташувати крапкові пари списку у порядку зростання добутку CAR и CDR частин.
13	Видалити зі списку атоми, що містять цифри.
14	В багаторівневому списку залишити атоми, що містять цифри.
15	Визначити рядок матриці, який містить елементи з максимальним значенням їх суми. Результат представити у вигляді (Номер_Рядка . Сума).
16	Визначити кількість елементів багаторівневого списку.
17	Видалити зі списку зазначений атом.
18	Скласти безперервну лінію з відрізків, які представлені кінцевими точками у вигляді ($(x_1\ y_1)\ (x_2\ y_2)$).
19	Перетворити інфіксну операцію виду (2 + 2) до префіксної нотації і обчислити значення виразу.
20	Скласти декартовий добуток двох множин.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Функції більш високого порядку. У мові Lisp і, взагалі в функціональному програмуванні, на основі єдиного представлення даних та програм, у якості фактичного аргументу функції можна застосовувати визначення функції. Такий аргумент називають функціональним, а функцію, що має функціональний аргумент – функціоналом [6, 7].

Функціонали застосовуються в тих випадках, коли необхідна різноманітна обробка одних і тих же даних. Розглянемо дуже простий приклад. Функції `F1` й `F2` обробляють числовий список наступним чином:

```
(defun F1 (X A)
(cond ((null X) NIL)
     (t (cons (- (car x) A) (f1 (cdr X) A) ) )
))
(defun F2 (X A)
(cond ((null X) NIL)
     (t (cons (cons (car x) A) (f1 (cdr X) A) ) )
))
```

Застосування цих функцій з однаковими фактичними параметрами дають наступні результати:

```
(F1 '(6 13 2) 2) → (4 11 0)
(F2 '(6 13 2) 2) → ((6 . 2) (13 . 2) (2 . 2))
```

Відмінність функцій `F1` і `F2` полягає у використанні різних функцій обробки голови списку в рекурсивної гільці функції. Якщо потрібну функцію передавати у якості фактичного параметру, тоді можна написати єдину функцію обробки числового списку:

```
(defun FC (FUN X A)
(cond ((null X) NIL)
   (t (cons (funcall FUN (car x) A) (FC (cdr X) A) ) )
))
```

Функція `FUNCALL` у мові Lisp застосовується для виклику функціонального аргументу.

Виклик функції `FC` має вигляд:

```
(FC '- '(6 13 2) 2) → (4 11 0)
(FC 'CONS '(6 13 2) 2) → ((6 . 2) (13 . 2) (2 . 2))
```

Таким чином, функція FC має один функціональний аргумент, тому ця функція є функціоналом, або функцією більш високого порядку.

У якості функціонального аргументу може бути використано: символьне ім'я, яке було визначено раніше (системна функція або функція, що визначена користувачем); лямбда-вираз; лексичне замикання.

Аплікативні функціонали. У мові Lisp передбачені два функціонала, що виконують виклик інших функцій, тобто застосовують функціональний аргумент до його параметрів. Такі функціонали називають функціонали, що застосовують, або аплікативні функціонали (`applicative functional`) [7].

Функціонал `APPLY` має наступний формат:

`(APPLY <Fn> <lst>),`

де: `Fn` – функціональний параметр; `lst` – список параметрів функції `Fn`.

Функціонал викликає функцію `Fn` з фактичними параметрами зі списку `lst` і повертає результат виконання функції `Fn`.

Функціонал `FUNCALL` відміняється від функціонала `APPLY` тим, що параметри функції Fn перелічені безпосередньо після функціонального аргументу:

$$(\texttt{FUNCALL <Fn>} \ \texttt{<}p_1\texttt{>} \ \texttt{<}p_2\texttt{>} \ \ldots \ \texttt{<}p_N\texttt{>}),$$

де: `N` – кількість параметрів функції `Fn`; `pi, i = 1, N` – параметри функції `Fn`.

Відображаючи функціонали. Відображаючи функціонали є функціями, що відображають деяким чином список на результат. Тому ці функціонали ще називають `MAP`-функціями [7]. Виклик функціоналів має наступний загальний формат:

$$(\texttt{MAPx <Fn>} \ \texttt{<}p_1\texttt{>} \ \texttt{<}p_2\texttt{>} \ \ldots \ \texttt{<}p_N\texttt{>}),$$

де: літера ***x*** у складі назви функції повинна бути замінена відповідними літерами імені конкретного функціонала. Літерами MAP починається ім'я всіх функціоналів які відображаються; `Fn` – функціональний параметр; `pi, i = 1, N` – списки; `N` – необхідна кількість параметрів функції `Fn`.

Функціонал `MAPCAR` повторює обчислення функції `Fn` для кожного елемента списків p_i. Результати обчислень збираються в єдиний список за допомогою функції `LIST`. Приклади з табл. 3.2 демонструють роботу функціонала. У першому рядку функція CONS створює крапкові пари з відповідних елементів списків `(a b c)` і `(1 2 3)`. Приклад з другого рядка демонструє як лямбда-вираз створює списки з відповідних елементів тих же списків.

Таблиця 3.2. **Приклади застосування функціоналу MAPCAR**

№ п/п	Виклик функції	Результат
1	`(mapcar 'cons '(a b c) '(1 2 3))`	`((a . 1) (b . 2) (c . 3))`
2	`(mapcar '(lambda (x y) (cons (y (cons x nil))) '(a b c) '(1 2 3))`	`((1 a) (2 b) (3 c))`

Функціонал MAPLIST повторює обчислення функції Fn для хвостових частин списків pi, починаючи зі самого списку. Результати обчислень збираються в єдиний список за допомогою функції LIST. В табл. 3.3 наведені приклади застосування функціоналу. У першому рядку функція LENGTH розраховує довжину списку на кожному кроці обчислень. Результат демонструє, що функціонал обробляє послідовно хвости списку. У другому рядку лямбда вираз створює крапкові пари з голови першого списку та голови хвоста другого списку. Коректний результат отримано за рахунок того, що другий параметр містить на один атом більше.

Таблиця 3.3. **Приклади застосування функціоналу MAPLIST**

№ п/п	Виклик функції	Результат
1	`(maplist 'length '(a b c))`	`(3 2 1)`
2	`(maplist '(lambda (x y) (cons (car x) (cadr y))) '(a b) '(0 1 2))`	`((A . 1) (B . 2))`

Функціонали MAPCAN та MAPCON аналогічні відповідно функціоналам MAPCAR та MAPLIST. Відмінність полягає в тому, що вони не створюють список результатів за допомогою функції LIST, а поєднують результати в єдиний список. При цьому використовується функція NCONC, тому бажано отримати результати у вигляді списків. Приклади застосування функціоналів наведені в табл. 3.4. У першому рядку на кожному кроці обчислень функція LIST створює списки з відповідних атомів фактичних параметрів: (a 1), (b 2), (c 3). Функціонал MAPCAN об'єднує ці списки в єдиний список. У другому прикладі виконується перевірка наявності кожного атома першого списку у відповідному хвості другого списку. Результати перевірки повертаються у вигляді списку з одного атому 1 або 0.

Необхідно звернути увагу на те, що функціонали MAPCAN та MAPCON знищують пусті списки. Таким чином, якщо функціональний аргумент повертає NIL, довжина списку менше кількості отриманих результатів,

тому ці функціонали використовують у якості фільтрів. Приклад у третьому рядку табл. 3.4 демонструє яким чином функціонал MAPCAN залишає у списку тільки числові атоми.

Таблиця 3.4. **Приклади застосування функціоналів MAPCAN, MAPCON**

№ п/п	Виклик функції	Результат
1	`(mapcan 'list '(a b c) '(1 2 3))`	`(a 1 b 2 c 3)`
2	`(mapcon '(lambda (x y) (if (member (car x) y) (list 1) (list 0)) '(a b c) '(c b a))`	`(1 1 0)`
3	`(mapcan '(lambda (x) (if (numberp x) (cons x nil) nil)) '(a 2 5 b 1 c))`	`(2 5 1)`

Лексичні замикання. На час обчислення функції або лямбда–виразу створюється так званий обчислювальний контекст, який знищується після завершення функції [6]. Іноді, корисно зберегти стан обчислювального контексту для його використання у потрібний час. Іншими словами, необхідно зберегти стан та зв'язки деяких змінних. У мові Lisp застосовується функція FUNCTION, яка блокує стан функції і зберігає стан обчислювального контексту. Таке блокування називається лексичним замиканням (lexical closure), тому що замикаються (зберігаються) зв'язки вільних змінних [7, 8]. Лексичне замикання може бути використане як функціональний параметр для завершення обчислень.

Існують два способи створення лексичного замикання: замикання на основі використання вільних змінних; параметричні замикання.

У першому випадку після створення лексичного замикання зберігається стан вільної зміни у контексті і нові значення зміни вже не впливають на подальші обчислення. Наприклад:

```
(setq Y 10) ; вільна зміна отримає значення 10
(setq S (function (lambda (x) (* x y)))) ; створено
; лексичне замикання S, у якому збережено стан вільної змінної Y.
```

Застосування лексичного замикання:

```
(funcall S 3) → 30
(funcall S 5) → 50
```

Нове значення вільної зміни не змінюватиме обчислювальний контекст:

```
(setq Y 5)
(funcall S 5) → 50
```

Параметричне замикання виконує ініціалізацію та зберігання контексту обчислень одностайно, що достатньо зручно при створенні генераторів будь-яких числових рядів. Наприклад, необхідно отримати список з одиниць та нулів, що чергуються. Напишемо функцію для створення лексичного замикання, яке зберігає початкове значення у зміні X:

```
(defun SC (X)
(function (lambda nil (setq X (if (= X 0) 1 0)))))
```

Рекурсивна функція, що генерує список:

```
(defun GenList (Fa N) ; Fa – функціональний аргумент
(if (= N 0) nil
(cons (funcall Fa) (GenList Fa (- N 1)))))
```

При кожному рекурсивному виклику функції генерується нове число в залежності від значення, що збережено у контексті обчислення. Таким чином, після виклику функції:

```
(GenList (SC 1) 5)
```

отримаємо результат: `(0 1 0 1 0)`.

МЕТОДИЧНІ ВКАЗІВКИ

Загальні рекомендації. Виконання завдання передбачає обов'язкове застосування функціоналів.

Використання функціоналів корисно у наступних випадках: коли потрібно виконати циклічні операції над кожним елементом списку, застосовують функціонали, які відображають; коли задача передбачає попередню підготовку даних для обчислення, застосовують аплікативні функціонали.

Таким чином, при виконанні завдання спочатку необхідно визначити функцію для обробки одного елементу списку, а потім застосувати цю функцію відносно всього списку за допомогою функціонала.

При розробці "фільтрів" (видалення будь-яких атомів зі списку тощо) необхідно розробити спочатку відповідну предикатну функцію. У подальшому слід використати предикатну функцію у складі лямбда-виразу у якості функціонального аргументу для функціоналів `MAPCAN` або `MAPCON` (див. приклад в табл. 3.4).

Приклад виконання завдання

Завдання: полічити кількість числових атомів у списку.

Рішення: Полічити кількість числових атомів можна наступним чином: створити список, в якому записується одиниця, якщо атом числовий, і нуль – якщо атом є символьним; обчислити суму елементів отриманого списку

Для визначення числового атому можна застосувати предикатну функцію NUMBERP. Тоді, для перетворення атому запишемо лямбда-вираз:

```
(lambda (X) (if (numberp X) 1 0))
```

і застосуємо його для кожного атому списку за допомогою функціонала MAPCAR. Для обчислення суми використаємо аплікативний функціонал APPLY з функціональним аргументом "+":

```
(defun CalcSum (Lst)
(apply '+ (mapcar '(lambda (X) (if (numberp X) 1 0)) Lst))
)
```

Очевидно, що виклик функції

```
(CalcSum '(a 45 7 b d 9))
```

повертає результат 3.

Контрольні запитання

- Визначення функціонального аргументу.
- Визначення функціонала.
- Аплікативні функціонали.
- Функціонали, що відображають.
- Визначення лексичного замикання.

Лабораторная робота № 4

ТИПИ ДАНИХ

Цілі роботи: ознайомлення з типами даних у мові Lisp; отримання практичних навичок роботи з типами даних.

Завдання: Розробити функцію для обробки даних відповідно варіанту (табл. 4.1). Використати функції мови Lisp, які призначені для відповідного типу даних. Застосувати лише рекурсію і/або функціонали для виконання циклічних операцій.

Таблиця 4.1

№ варіанту	Зміст завдання
1	У рядку змінити регістр за вибором: прописні, строкові або як у реченні.
2	Змінити місцями найбільший і найменший елементи двовимірної матриці
3	У рядку залишити лише по одному пробілу між словами.
4	Зі списку книг отримати ті, що видані раніше зазначеного року.
5	Полічити у рядку кількість слів, що починаються із зазначеної літери.
6	Упорядкувати рядки двовимірної матриці за зменшенням максимального елементу рядка.
7	У рядку залишити лише слова, що містять задану комбінацію літер
8	У списку книг знайти всі книги зазначеного автора
9	Знайти довжину самого короткого слова у рядку
10	Транспонувати двовимірну матрицю
11	У рядку видалити всі знаки, що не є літерою або цифрою.
12	Створити для масиву чисел відповідний бітовий вектор для відображення знаку кожного елементу: 0 - від'ємне число, 1 - додатне число або нуль
13	У рядку видалити останню літеру кожного слова.
14	Зі списку книг отримати унікальний список авторів
15	У кінці кожного слова рядка додати символ.
16	Полічити кількість слів у рядку, що містять зазначену літеру.
17	Визначити наявність у рядку зазначеної комбінації слів.
18	Перетворити список чисел до вектору і розташувати числа у порядку: від'ємні числа, нулі, додатні числа
19	Перетворити рядок до списку, кожний атом якого представляє слово.
20	Виконати сортування стовпців двовимірної матриці за зменшенням суми елементів.
21	Реалізувати сортування масиву способом бульбашки.
22	Зі списку книг отримати унікальний список видавництв
23	У рядку полічити загальну кількість знаків, кількість літер та кількість слів.
24	У списку книг знайти всі книги, назва якої починається зі зазначеної літери

ТЕОРЕТИЧНІ ВІДОМОСТІ

Загальні поняття про типи даних. Підклас об'єктів даних, що виділяються представленням та засобами обробки, називають типом даних. Визначення типу даних може бути явним або неявним. У функціональних мовах програмування, в тому числі у мові Lisp, переважно застосовано неявний спосіб визначення типу, тобто тип даних визначається значенням даних [5, 7].

Визначення типів даних базується на абстракції даних – відділення властивостей даних від об'єктів даних. Абстрактний тип – це тип даних, що визначений операціями чи функціями, які застосовуються відносно типу даних. Наприклад, список – абстрактний тип даних, що реалізований за допомогою cons-осередків. Базові функції (`CAR, CDR, CONS, ATOM` та інші) створюють множину операцій, що можна застосовувати відносно списків, незалежно від складу списків.

На основі списків у мові Lisp розроблені інші типи даних. Функції для роботи зі списками складають базовий механізм реалізації додаткових типів даних.

Основні типи даних. Мова Lisp є без типовою мовою програмування. Це означає, що повною мірою реалізовано абстракція даних і типів. Тип змінної визначається динамічно за типом даних, з якими встановлено зв'язок.

Основні типи даних у мові Lisp є списки (`list`), числа (`number`), символи (`symbol`), які використовувались при розробці мови. Інші типи реалізовані пізніше як підтипи відносно основних. Необхідно відмітити, що введено більш загальний тип `SEQUENCE` (послідовність), а тип `LIST` є підтипом послідовності. Для перевірки типу об'єкту використовується предикатна функція `TYPEP`:

```
(TYPEP <об'єкт> <тип>)
```

Послідовності. Послідовність – це узагальнений тип та представляє деяку множину даних. Підтипами послідовності є список, вектор, рядок. Функції для створення та обробки послідовності можна використовувати для підтипів. Однак, для кожного з підтипів розроблені більш ефективні засоби, які рекомендовано до застосування.

Для створення послідовності використовується функція:

```
(make-sequence <тип> <кількість>
&key : initial-element <значення>),
```

де: <тип> – тип послідовності, може прийняти значення: list, vector, string, а також їхні підтипи; <кількість> – кількість екземплярів об'єктів даного типу; <значення> – початкове значення кожного об'єкту послідовності. У випадку відсутності цього параметру використовується значення за замовчуванням. В таблиці 4.2. наведені приклади створення послідовностей.

Таблиця 4.2.**Приклади створення послідовностей**

Вираз	Результат
(make-sequence 'list 3 :initial-element 0)	(0 0 0)
(make-sequence 'vector 4)	#(nil nil nil nil)
(make-sequence 'string 5 :initial-element #\a)	"aaaaa"
(make-sequence 'bit-vector 4 :initial-element 1)	#*1111

Для отримання елементу послідовності служить функція ELT:

(ELT <послідовність> n)

де: n – номер елементу послідовності, починаючи з нуля.

Зміна значення елементу виконується за допомогою узагальненої функції присвоєння SETF.

В таблиці 4.3. наведені деякі корисні функції обробки послідовностей. За більш повною інформацією звертайтесь до джерел [7].

Таблиця 4.3. **Функції обробки послідовностей**

Функція або функціонал	Призначення
subseq	Отримати частину послідовності
map	Аналогічно функціоналу MAPCAR
every	Перевірка властивостей кожного елементу
remove(-if-not)	Віддалення елементів послідовності
substitute(-if-not)	Зміна частини послідовності
find(-if-not)	Пошук елементу
count(-if-not)	Розрахунок кількості за умовою

Масиви. У відмінності від списків масиви мають зручний механізм індексування доступу до будь-якого елементу. Всі елементи масиву відносяться до одного типу. Дані масиву чітко структуровані за розмірністю, яка задається при визначенні масиву.

Для створення масиву у мові Lisp передбачена функція:

(MAKE-ARRAY (n_1 n_2 … n_k) <режими>),

де: n_1 n_2 … n_k – кількість елементів за кожної розмірністю; k – розмірність масиву; <режими> – додаткові параметри, що використовуються при ініціалізації.

Наприклад, виклик функції:

```
(make-array '(2 1 2)   :element-type 'integer
                       :initial-element 1)
```

повертає трьохвимірний масив, всі елементи якого – одиниці. Масив представляється як багаторівневий список:

#3A(((1 1)) ((1 1))).

Знаки перед списком указують на те, що список створений як масив.

Доступ до елементів масиву здійснюється за допомогою функції:

(AREF <масив> n_1 n_2 … n_k),

де: n_1 n_2 … n_k – індекси елементу масиву

Зміна значення елементу масиву виконується узагальненої функцією присвоювання SETF. Наведемо приклад створення двовимірного масиву у середовищі Common Lisp та зміна значення елементу третього рядку другого стовпця:

```
>(setq DM (make-array '(4 3) :initial-element 0))
#2A( (0 0 0) (0 0 0) (0 0 0) (0 0 0) )
>(setf (aref DM 2 1) 7)
7
>DM
#2A( (0 0 0) (0 0 0) (0 7 0) (0 0 0) )
```

Одновимірний масив відноситься до окремого типу – вектор (VECTOR). Вектор може бути створений як послідовність або як масив.

Рядки та знаки. Для роботи з рядком існує набір функцій, що відносяться до цього типу даних. Одночасно, рядки наслідують дії та властивості, які застосовуються для послідовностей.

Рядок складається з послідовності знаків й обмежується з двох боків лапками. Знак теж є тип даних і відображається у вигляді #***x***, де ***x*** – будь-який, у даному випадку, друкований або не друкований символ.

Для доступу до знаку рядка використовується функція CHAR:

(CHAR <рядок> N),

де N – індекс знаку, починаючи з нуля.

Рядки можна порівнювати за допомогою спеціальних функцій, дія яких зрозуміло з назви: STRING=, STRING>, STRING>=, STRING<, STRING<=.

Структури. Структури служать для об'єднання даних будь-якого типу у логічну одиницю. Списки у мові Lisp також мають таку можливість, однак для обробки даних потрібні більш ефективні засоби.

Для використання структури спочатку потрібно їй визначити наступним чином:

(DEFSTRUCT <символ> <p1> <p2> …),

Де: <символ> – тип структури, що визначається.

<p1> <p2> … – позначення полів структури.

Макрос DEFSTRUCT у якості побічного ефекту генерує декілька функцій:

– функцію MAKE-<символ> для створення об'єкту структури;
– функцію COPY-<символ> для копіювання об'єкту структури;
– предикатну функцію <символ>-P для визначення, чи є об'єкт структурою <символ>;
– функції доступу до полів структури у вигляді <символ>-<pi>.

Наведемо приклад визначення структури, що описує замовлення на друкування фотографій. Замовлення представимо у вигляді структури з наступних полів:

– ім'я замовника (NAME) – рядок;
– дата замовлення (ORDERDATE) – структура, що складається з полів: день, місяць, рік;
– список файлів для друку (FILES) – список;
– вартість (PRICE) – число.

Розглянемо процес створення структур у середовищі Common Lisp.

Створення структури DATE:

```
>(defstruct DATE day month year)
DATE
```

Створення структури замовлення ORDER

```
>(defstruct ORDER name orderdate files price)
ORDER
```

Створення об'єкту замовлення:

```
>(setq ord1 (make-order :name "Стасюк" :orderdate
(make-date :day 12 :month 5 :year 2009) :files
'("f1.jpg" "f2.jpg" "f3.png" "f4.png") :price 45.0))
#S(ORDER :NAME "Стасюк" :ORDERDATE #S(ORDERDATE :DAY
12 :MONTH 5 :YEAR 2009) :FILES ("f1.jpg" "f2.jpg"
"f3.png" "f4.png") :PRICE 45.0)
```

Зміна вартості замовлення:

```
>(setf (order-price ord1) 50.0)
50.0
```

Перевірка значення поля PRICE:

```
>(order-price ord1)
50.0
```

Перетворення типів. Типи, що є похідними від одного типу, можна перетворити з одного типу у другий. Для цього застосовується функція COERCE. Приклади застосування функції наведені у методичних вказівках до лабораторної роботи № 2.

МЕТОДИЧНІ ВКАЗІВКИ

При виконанні завдання необхідно обов'язково використати функції, що передбачені для відповідного типу даних. Це потрібно з метою глибокого вивчення засобів обробки даних певного типу.

Для закріплення попередніх тем слід застосувати лише рекурсію або функціонали для циклічних операцій.

Необхідно звернути увагу, що функції для базових типів можна застосувати для похідних типів. В таблиці 4.3 показане застосування функції `ELT` для отримання елементу даних різних типів. Відповідні функції отримання елементу за типом даних більш ефективні, тому необхідно віддавати їм перевагу.

Таблиця 4.3. **Функції отримання елементу за типом даних**

Вираз	Тип	Відповідна функція типу даних
`(elt '(a b c) 1)`	Список (list)	NTH
`(elt '#(a b c) 1)`	Вектор (vector)	AREF
`(elt "abcde" 1)`	Рядок (string)	CHAR
`(elt #*010101 1)`	Бітовий вектор (bit-vector)	AREF

Контрольні запитання

- Визначення типу "послідовність". Основні функції обробки послідовності.
- Визначення типу "рядок". Функції обробки рядка.
- Визначення типу "масив". Функції обробки масиву.
- Функція перетворення даних з одного типу в другий.
- Асоціативні списки.
- Визначення структури. Побічні ефекти при створенні структури.

Лабораторна робота № 5

СКЛАДАННЯ ПРАВИЛА НА МОВІ PROLOG

Цілі роботи: ознайомлення з основами мові Prolog; вивчення правил уніфікації змінних; одержання практичних навичок складання найпростіших правил на мові Prolog.

Завдання: скласти правило для визначення предикату відповідно варіанту (табл. 5.1.) на основі фактів: *чоловік (ім'я), жінка (ім'я), батьки (ім'я_батька, ім'я_дитини).*

Таблиця 5.1. **Варіанти завдань**

№	Правила	№	Правила
1	Двоюрідний брат	14	Братова (дружина брата)
2	Дідусь	15	Зовиця (сестра чоловіка)
3	Дівер (брат чоловіка)	16	Шурин (брат дружини)
4	Онук	17	Своячка (сестра дружини)
5	Свекруха	18	Зять
6	Дядько	19	Свояк (чоловік сестри дружини)
7	Тітка	20	Сват
8	Племінник	21	Тесть
9	Племінниця	22	Сестра
10	Троюрідний племінник	23	Невістка
11	Бабуся	24	Онука
12	Теща	25	Двоюрідна сестра
13	Свекор	26	Брат

ТЕОРЕТИЧНІ ВІДОМОСТІ

Мова логічного програмування Prolog. Prolog – це найбільш відома мова логічного програмування, яка використовується для вирішення задач, в яких діють об'єкти та відношення між цими об'єктами. Програмування на мові Prolog відноситься до декларативного стилю програмування і має спільні властивості з функціональним програмуванням [3].

Теоретичною основою логічного програмування є логіка предикатів – один з розділів алгебри логіки. Логіка предикатів займається дослідженням відношень між об'єктами та доказами їхнього існування [4]. Такі відношення називаються предикатами і записуються у вигляді

$P(t_1, t_2, t_3, \ldots, t_n)$,

де: P – позначення предикату $t_1, t_2, t_3, \ldots, t_n$ – об'єкти чи терми.

Предикати можна об'єднати за допомогою логічних операцій у формули, які складають фрази (висловлювання). Фраза, що містить тільки один позитивний літерал (логічна формула), називається фразою Хорна і має вигляд:

$$C \vee \neg E \vee \neg F \vee \neg G \qquad (5.1.)$$

Фраза 5.1 означає, що C істина тоді і тільки тоді, коли E, F і G одностайно істинні. Звичайно, 5.1 можна переписати у вигляді імплікації зі зворотною стрілкою:

$$C \leftarrow E, F, G \qquad (5.2.)$$

Така фраза є основою побудови висловлювань програми на логічної мові програмування Prolog.

Склад та структура програми. Програма на мові Prolog складається з висловлювань трьох типів: факти, правила та питання. Твердження складається з двох частин: ліва частина – висновок, права частина – умова, які розділяються знаком ":-" (двокрапка і дефіс) [1, 4].

Факт – це твердження, що є безумовною істиною, який складається з висновку за яким не виходить ніяких умов, наприклад:

```
лікар (василь).
```

Це можна прочитати так:

```
Василь – лікар*).
```

Сукупність фактів складають деякий світ, у межах якого дійсні твердження програми.

На прикладі родинних відносин (рис. 5.1.) введемо відношення батьки (parent) між двома об'єктами X і Y, яке встановлює той факт, що X є батьком чи мати для Y:

```
parent (X, Y).
```

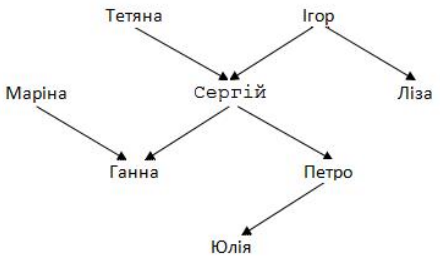


Рис. 5.1. Схема родинних відношень

Тоді родинні зв'язки, які зображені на рис. 5.1. запишемо у вигляді списку фактів:

```
parent (тетяна, ігор).
parent (ігор , сергій).
parent (ігор, ліза).
parent (сергій , ганна).
parent (сергій, петро).
parent (марина, ганна).
parent (петро, юлія).
```

При написанні фактів потрібно дотримуватись наступних правил: імена всіх відносин та об'єктів з маленької літери; спочатку записується ім'я відносин, потім в круглих дужках через кому об'єкти; в кінці ставиться крапка.

Після написання будь-якої програми та її завантаження середовище Prolog може відповісти на запитання.

Запитання – це твердження, що містить лише умову без висновку. Запитання складається з одній або декількома цілей [2]. Якщо всі цілі досягненні, Prolog дає позитивну відповідь – Так (Yes), у протилежному випадку – негативну відповідь Ні (No). Якщо у параметрах предикатів існують зміни, тоді, при позитивній відповіді, зміни конкретизуються, тобто приймають конкретне значення (див. нижче). В таблиці 5.2. наведені приклади запитань у контексті вищевказаної програми родинних відношень.

Таблиця 5.2. **Приклади запитань**

Запитання	Відповідь
`? parent(сергій , ганна).`	`Yes`
`? parent(сергій , ліза).`	`No`
`? parent (сергій ,X).`	`X = ганна;` `X = петро;`
`? parent (X, ліза).`	`X = сергій`

Правила – це *залежні відношення* і дозволяють Prolog робити виведення, *порівнюючи* одну частину інформації з іншою [1, 2].

Правила Prolog складаються з обох частин твердження: висновку та умови. Іноді висновок називають головою, умови – тілом правила.

Голова – це *факт*, який буде *вірним*, якщо деяка кількість умов виконається. Тіло – це *множина умов*, котрі повинні бути *істинними*, щоб Prolog міг довести, що голова правила *вірна* [2].

Наприклад, введемо відношення child "дитина", зворотне до parent "батьки" і запишемо у вигляді *правила*:

```
child(Y, X):-parent (X, Y).
```

Правило читається так: для всіх X та Y, Y дитиною X, якщо X – батько (мати) Y.

Правило доповнює базу даних програми новою сутністю child. Зараз можна запитати:

```
?-child(ліза, ігор).
```

В базі даних програми немає фактів `child`, але є правило, яке залежить від існуючих фактів parent. У даному випадку Prolog підставляє умовну частину правила замість запитання:

```
parent (ігор, ліза),
```

після чого ініціює процедуру пошуку в базі даних.

Основи мови Prolog. Програма на мові Prolog складається з речень. Речення бувають трьох видів: факти, правила, запитання. Всі речення складаються з термів [1, 2, 3].

Терм – це або *константа*, або *змінна*, або *вживання функції*. *Вживання функції* записується як символьне позначення функції за яким в дужках знаходиться список аргументів. Кожен аргумент теж є термом.

$f\ (t_1, t_2, \ldots, t_n)$,

де: f – позначення функції; $t_1, t_2, \ldots, t_n$ – терми.

Терм записується як послідовність *літер*. Літери поділяються на чотири категорії: прописні букви, строкові букви, цифри, спеціальні символи такі як: `+ - * / \ < > ^ : . ? & ? _ @ # $.`

Константами є поіменні конкретні об'єкти або конкретні відношення. Існує два види констант: *атоми* та *числа*.

Атомом називається конкретний об'єкт, який починається з малої літери. Якщо атом береться в одиночні лапки, то його ім'я може складатися з будь-яких літер. Наприклад:

```
->
=====>
міс_Джонс
key5689
'Джордж-сміт'
```

Числа використовуються для представлення числових даних, що дозволяє виконувати над ними арифметичні операції. Цілі числа складаються тільки з цифр, без десятинної коми.

Змінні складаються з літер, цифр та символів підкреслення та починаються з прописної літери або символу підкреслення. Наприклад:

```
A
_33
```

Структура – це єдиний іменований об'єкт, який складається з сукупності інших об'єктів, які називаються параметрами. Параметри записуються в дужках через кому після імені структури. Наприклад, представимо структуру дата, яка складається з трьох параметрів: день, місяць, рік:

```
дата(22, травня, 2000)
```

Ім'я структури називається функтором, кількість параметрів – арністю. Prolog дозволяє застосування структур з однаковим функтором і різною арністю.

Відповідь на запитання і пошук рішень. Відповідь на запитання здійснюється на основі фактів і правил, з яких складається програма. Необхідно звернути увагу на те, що повнота відповіді залежить від повноти складу бази даних програми (фактів), що описують "світ" програми [4].

Основний механізм, на якому базується пошук рішень, є співставлення термів, і називається *уніфікацією*.

Уніфікація може бути успішною або ні. Якщо уніфікація успішна, тоді ціль досягнуто. Успіх уніфікації залежить від складу термів, які співставленні. Крім того, якщо терм є зміною або у складі терму входить зміна, в результаті уніфікації зміна може бути конкретизована (отримати значення), якщо уніфікація успішна. Prolog керується наступними правилами [3, 4]:

- Якщо уніфікуються дві константи, тоді уніфікація є успішною, якщо обидві константи однакові.
- Якщо уніфікуються зміна з константою або структурою, уніфікація є успішною, а зміна конкретизується значенням константи або структури.
- Якщо уніфікуються дві вільні зміни, у подальшому вони сприймаються як одна, тобто, якщо одна із змін конкретизується, друга зміна теж конкретизується тим же значенням.
- Якщо уніфікуються дві структури, тоді уніфікація є успішною, якщо функтори і арність структур однакові, а відповідні параметри структур уніфікуються за вищевказаними правилами.

- Уніфікація завжди успішна, якщо терм зіставляється з анонімною зміною (підкреслення).

Запитання ініціює процес послідовної уніфікації цілей запитання з фактами та правилами бази даних. Якщо уніфікація для поточної цілі пройшла успішно, виконується перехід до наступної цілі. Співставлення з правилом заміняє ціль запитання на тіло правила і процес продовжується для кожної цілі. Якщо всі цілі досягнуті, рішення знайдено.

МЕТОДИЧНІ ВКАЗІВКИ

Етапи виконання роботи. Виконання завдання складається з чотирьох етапів: складання схеми зв'язків між об'єктами; формулювання і запис правила з використанням квантору загальності [4]; запис правила на мові Prolog; формулювання запитань та отримання відповідей у середовищі Prolog.

При виконанні другого етапу достатньо звертати увагу лише на декларативний сенс правила. У відмінності від цього, на наступному етапі більш уваги слід звернути на процедурний сенс правила для підвищення ефективності пошуку рішень.

Приклад виконання завдання.

Задача: скласти правило для визначення предикату "донька" на основі фактів . *жінка (ім'я), батьки (ім'я_батька, ім'я_дитини).*

Рішення: предикат "донька" позначимо `daughter(X, Y)`, де `X` – ім'я доньки, `Y` – імя одного з батьків (мамо або тато). Базу фактів можна скласти на основі схеми родинних відношень, яка представлена на рис. 5.1. Необхідно додати ще факти, які визначать жінок:

```
woman(тетяна).
woman(ліза).
woman(ганна).
woman(марина).
woman(юлія).
```

Предикат можна сформулювати наступним чином: *для всіх X і Y, X є донькою Y, якщо X є жінкою, і Y є батьком X.*

Тоді, правило на мові Prolog має вигляд:

```
daughter(X,Y)  :-  woman(X),  parent(Y,X).
```

Після цього можна поставити запитання (табл. 5.3.).

Таблиця 5.3. **Приклади запитань**

Запитання	Пояснення	Відповідь
`? daughter(ганна,марина).`	Чи є Ганна донькою Марини?	Yes
`? daughter(ганна, X).`	Хто є батьком (мати або тато) Ганни?	X = марина; X = сергій
`? daughter(X,марина).`	Хто є донькою Марини?	X = ганна
`? daughter(_,ліза).`	Чи є у Лізи донька?	No

Контрольні запитання

- Визначення поняття предикату.
- Визначення понять: факт, правило, запитання.
- Основні визначення мови Prolog: терм, атом, константа, змінна, структура.
- Правила уніфікації термів.
- Поясніть, як Prolog-система дає відповідь на запитання.

Лабораторна робота № 6

УПРАВЛІННЯ ПРОГРАМОЮ

Цілі роботи: ознайомлення з предикатами введення-виведення у мові Prolog; вивчення та одержання практичних навичок з формуванням та змінні бази даних програми.

Завдання: Використовуючи завдання з лабораторної роботи №5 скласти правила: для отримання першого варіанту рішення; для отримання всіх можливих рішень.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Механізм повернення. При відповіді на запитання Prolog – система розглядає всі факти і правила як множина аксіом, запитання – як теорему. Потім Prolog намагається доказати цю теорему, тобто показати, що теорема логічно виводиться з аксіом [4].

Якщо запитання складається з декількох цілей, які зв'язані кон'юнкцією, Prolog послідовно, зліва направо, виконує пошук рішення (до успіш-

ної уніфікації) для кожної з них. При цьому, якщо ціль містить зміну, вона конкретизується. При досягненні поточної цілі Prolog встановлює маркер і переходить до пошуку рішення для цілі, яка знаходиться праворуч. Зміни, яки були конкретизовані, рахуються як константи. Такий процес продовжується до останньої цілі твердження, після чого Prolog система виводить значення конкретизованих змін.

Якщо будь-яка ціль не досягнуто, тоді Prolog намагається знайти друге рішення. Для цього виконується повернення до попередньої цілі (ліворуч), відміняється конкретизація змін (якщо вона була), і, починаючи з останнього маркера, продовжується пошук рішення. Якщо пошук закінчується невдачею, знову виконується повернення до цілі, яка знаходиться ліворуч. Такий процес називається зворотним трасуванням (back tracking) або механізмом повернення.

Виклик механізму повернення. Якщо досягнуто остання ціль твердження, однак потенційно в базі даних можна знайти ще рішення, тобто не всі факти пройдені, система звертається до користувача за вказівкою, продовжити пошук чи ні. У рази необхідності отримання всіх рішень слід запустити механізм повернення після отримання кожного рішення. Для цього у мові Prolog існує предикат ***fail***, який завжди є хибним, тобто закінчується невдачею. Використання ***fail*** у складі тіла твердження ініціює перебір рішень для цілей, які розташовані ліворуч.

Для схеми родинних зв'язків (рис. 5.1) напишемо правило для отримання всіх дітей об'єкту X:

```
allchild(X) :- parent(X,Y), write (Y), nl, fail.
```

Очевидно, що на запитання:

```
?- allchild(сергій).
```

отримаємо відповідь:

```
ганна
петро
No
```

В процесі прямого трасування перший випадок успішної уніфікації відповідає факту `parent (сергій , ганна)` і зміна ***Y*** конкретизується значенням `. ганна`, яке виводиться предикатом ***write***. Наступна ціль ***fail*** завжди невдячна, тому ініціюється механізм повернення: зміна ***Y*** стає неконкретизованою, система продовжує пошук наступного рішення. Остання відповідь ***No*** означає, що база даних вичерпана і рішень більш немає.

Для управлінням механізмом повернення також використовуються предикати ***repeat***. Спеціальний вбудований предикат ***repeat*** завжди погоджується з базою даних і завжди може бути передоказаний при поверненні. Предикат ***repeat*** – ціль, яка завжди успішна, її особлива якість складається з того, що вона не детермінована, тому кожного разу, як до неї доходить перебір, вона породжує нову гілку обчислень.

В якості прикладу наведемо цільове твердження *менше_Y(X, Y)* яке запрошує число більше заданого:

```
менше _Y(X, Y) :- repeat,                  % організація циклу
        nl,
        write ('? X < '), write (Y),       % вивід запиту
        read (X),                          % зчитування
        X < Y.                             % перевірка відношення
```

Відтинання – спосіб обмеження пошуку рішень. Механізм повернення забезпечує отримання всіх рішень, які можливі відповідно бази даних. Однак, при рішенні задач, іноді потрібне лише одне рішення за визначеними умовами. Завдання полягає в тому, що при отриманні рішення система повинна його "закріпити" і відмовитись від пошуку альтернатив. Для обмеження пошуку у мові Prolog існує предикат "відтинання", який має позначку "!". Предикат використовується як ціль твердження і завжди є вдалим.

В процесі доказу твердження ціль "відтинання" змушує систему прийняти ті альтернативні рішення, які були обрані раніше при досягненні попередніх цілей [2]. Нехай існує деяке правило:

```
C :- P, Q, R, !, S, T, U.

C :- V.

A :- B, C, D.

?- A.
```

"Відтинання" вплине на обчислення мети **C** у такий спосіб. Перебір буде можливий у списку цілей **P, Q, R**; однак, як тільки ціль "відтинання" буде досягнута, всі альтернативні рішення для цього списку вилучаються з розгляду. Альтернативне речення **C:- V**. також не буде враховуватися. Проте перебір буде можливий у списку цілей **S, T, U** з урахуванням тих рішень, які були отримані при досягненні цілей **P, Q, R**.

Таким чином, "відтинання" вплине тільки на мету **C** і воно буде "невидимо" з мети **A**. Автоматичний перебір однаково буде відбуватися в списку цілей **B, C, D**, поза залежністю від наявності "відтинання" в реченні, що використовується для досягнення **C**.

МЕТОДИЧНІ ВКАЗІВКИ

Для виконання завдання потрібно розширити базу фактів, яка використовувалась у лабораторній роботі № 5.

Перша частина завдання може бути вирішена за допомогою предикату "відтинання", якій обмежує пошук у базі даних програми. Другу частину завдання необхідно виконати у двох варіантах з використанням вбудованих предикатів *fail* і *repeat*.

Приклади виконання завдання приведені вище.

Контрольні запитання

- Призначення механізму повернення.
- Управління механізмом повернення.
- Призначення предикату "відтинання".
- Пояснити, як "відтинання" впливає на пошук рішень.

Лабораторна робота № 7

СПИСКИ

Цілі роботи: ознайомлення з складними структурами даних у мові Prolog; одержання практичних навичок роботи зі списками та структурами при написанні правил на мові Prolog.

Завдання: Створити базу даних програми з фактів, які описують результати сесії студентів у вигляді:

```
student(Group, Fam, [D1/M1, D2/M2, D3/M3, D4/M4]).
```

де: `D` – назва дисципліни, `M` – відповідна оцінка, `Group` – навчальна група, `Fam` – прізвище студента. Отримати список об'єктів даних відповідно завдання (табл. 7.1.) та вивести його у табличній формі.

Таблиця 7.1. **Варіанти завдань**

№ варіанту	Зміст завдання
1	Студенти, що одержали тільки відмінні оцінки
2	Студенти, що одержали відмінну оцінку з вказаної дисципліні
3	Студенти, прізвище яких починається з зазначеної літери
4	Студенти, які отримують стипендію за результатами сесії (правила встановити самостійно)
5	Студенти, що отримали хоча б одну незадовільну оцінку
6	Студенти, що отримали більш двох незадовільних оцінок
7	Список студентів, які навчаються в одній групі і одержали лише позитивні оцінки
8	Список студентів, упорядкований за середній бал
9	Студенти, які навчаються у вказаній групі і не отримують стипендію.
10	Студенти, які навчаються на одній спеціальності і одержали незадовільну оцінку
11	Студенти, які навчаються на одній спеціальності і одержали позитивні оцінки
12	Список груп і кількості відмінних оцінок
13	Список груп і кількість студентів, що отримали позитивні оцінки
14	Список груп і кількість студентів, що мають середній бал вище вказаного
15	Список студентів і розмір стипендії за результатами сесії

ТЕОРЕТИЧНІ ВІДОМОСТІ

Визначення списку у мові Prolog. Список – це послідовність з будь-якої кількості елементів. Кожний елемент може бути будь-яким термом, припустимим у Prolog, в т.ч. списком [1].

Список має структуру бінарного дерева. Для позначки списку використовується функтор «`./2`» (крапка) з двома аргументами:

```
(Head, Tail),
```

де: `Head` – голова списку, у якості якого може бути будь-який терм;
`Tail` – хвіст списку, у якості якого може бути тільки список.

Таким чином, послідовність з чотирьох констант `a, b, c, d`, що складають список, можна представити у вигляді структури:

```
.(a, . (b, . (c, . (d, [])))),
```

де `[]` – пустий список.

Для більш зручної позначки списку застосовується інша форма запису – у квадратних дужках перелічуються елементи через кому:

```
[a, b, c, d].
```

Якщо потрібно виділити хвіст списку, використовується символ "|" (вертикальна риса), який розділяє один або декілька елементів списку від хвоста:

`[a | T]`	список містить голову і хвіст **T** з будь-якої кількості елементів
`[a, b, c | T]`	список містить три константи у початку і хвіст **T** з будь-якої кількості елементів.

Обробка списків. Структура списків дає можливість застосування рекурсії для обробки списків, тому що дії, які виконуються стосовно голови можна повторити для хвоста списку. Одне з правил повинна бути термінальною, тобто виконується конкретизація зміни заздалегідь передбаченим значенням. Альтернативні правила повинні мати у складі умовної частини рекурсивну ціль. Розглянемо простий приклад – отримання останнього елементу списку.

Для рішення напишемо предикат, який має формат:

```
last_el(L, X).
```

де `L` – список, `X` – результат (останній елемент, якщо він є).

Необхідно зазначити, що задача має два термінальних випадки: якщо список пустий, результат може бути будь що, тобто зміна X залишається неконкретизованою: last_el([], _).; якщо список містить лише один елемент, він буде результатом: last_el([X], X). У інших випадках потрібно відкинути голову і знайти останній елемент тільки в хвості списку, тобто здійснювати рекурсивний виклик правила:

```
last_el([H|T], X) :- last_el(T, X).
```

У мові Prolog передбачено ряд предикатів для обробки списків, який представлений в таблиці 7.2.

Таблиця 7.2. **Вбудовані предикати для обробки списків**

Предикат	Призначення
`is_list(<term>)`	Перевірка, чи є `<term>` списком.
`append(<list1>,<list2>, <list3>)`	Обєднання списків `<list1>` і `<list2>` в результуючий список `<list3>`
`member(<elem>, <list>)`	Перевірка входження терму `<elem>` в список `<list>`

Продовж. табл. 7.2

Предикат	Призначення
delete(<list1>, <elem>, <list2>)	Отримання списку <list2> у результаті видалення терму <elem> зі списку (<list1>
reverce(<list1>,<list2>)	Інвертування списку <list1> в список <list2>
flatten(<list1>,<list2>)	Розташування всіх елементів багаторівневого списку <list1> в простий список <list2>
length(<list>, <len>)	Отримання кількості елементів <len> списку <list>

МЕТОДИЧНІ ВКАЗІВКИ

Збереження рішень у списках. Prolog система забезпечує отримання результатів рішень за допомогою механізму повернення. Однак кожне рішення необхідно використати одразу після отримання, інакше воно пропаде. Вбудовані предикати bagof, setof, findall дозволяють накопити рішення у списку для подальшої їх обробки.

Всі предикати мають формат: F(X, P, L).
де: F – один з вищеназваних функторів; X – одна або декілька змін, що уособлюють ті об'єкти, які потрібно знайти; P – предикат, у залежності від якого виконується пошук; L – список для накопичення результатів.

В таблиці 7.3. наведені приклади застосування предикатів для родинних відносин, зображених на рисунку 5.1. – необхідно отримати список всіх об'єктів.

Таблиця 7.3. **Предикати отримання списків об'єктів бази даних**

№	Застосування	Результат
1	bagof(X,parent(X,_),L)	L=[тетяна]; L=[ігор]; L=[ігор]; L=[сергій]; L=[сергій, марина]; L=[петро]
2	setof(X,parent(X,_),L)	L=[тетяна]; L=[ігор]; L=[ігор]; L=[сергій]; L=[марина, сергій]; L=[петро]

Продовж. табл. 7.3

№	Застосування	Результат
3	`findall(X,parent(X,_),L)`	`L=[тетяна, ігор, ігор, сергій, сергій, марина, петро]`

бази даних, які є батьками (`parent`). Видно, що результати, які отримані предикатами `bagof` і `setof`, залежать від значення другого аргументу предиката *parent*, а предикат `findall` формує повний список об'єктів. Предикат `setof` у відміну від `bagof` упорядковує список.

Створення і декомпозиція атомів. У мові Prolog існує предикат `name`, який встановлює взаємозв'язок між атомами та їх кодировкою в ASCII. Предикат має формат: `name(A, L)`,

де: A – атом, L – список кодів символів атому A.

`name` істинно, якщо символи атому відповідають кодам списку L.

Наприклад:

```
? name(a123, L).
L = [97, 47, 48, 49]  % зміна L конкретизується списком кодів
                      % атому a123
? name(A, [97, 47, 48, 49])
A = a123              % зміна A конкретизується константою a123.
```

Приклад виконання завдання

Задача: отримати список студентів, які здобули оцінку "добре" з дисципліни "математика" на основі фактів бази даних, що записані у вигляді:

```
student (4151, petrov, [physics/5, mathematics/4,
c++/5, oop/3]).

student(4152, veselkov, [physics/4, mathematics/3,
c++/4, oop/5]).
```

Рішення: для отримання списку напишемо правило з використанням вбудованого предикату setof, тому що одностайно список буде упорядкованим. Однак, у зв'язку з тим, що setof в процесі пошуку рішення враховує значення інших аргументів (див. табл. 7.2), потрібно скласти окремий предикат для отримання прізвища студента, що одержав оцінку "добре" з математики:

```
p_stud(X) :- student(_, X, [ _,mathematics/4 |_ ] ).
```

Правило містить в правій частині ціль для співставлення з фактами student/3. Зверніть увагу на те, що застосована анонімна зміна для тих аргументів, які у даному випадку не потрібні. Тоді правило для отримання списку запишемо у наступному вигляді:

```
good_math(L) :- setof (X, .p_stud(X), L).
```

Для виводу списку напишемо рекурсивне правило:

```
% термінальне правило, виведення загальної кількості студентів
print_list([],N):-write('Всього '),write(N),
                write(' студентів'), !.
% рекурсивне правило. Виведення голови списку...
   print_list([X|T], N):- write(N), write('.'),
                tab(2), write(X),nl,
% ... рекурсивний виклик для виведення хвосту з наступним номером рядка
            K is N + 1, print_list[T, K).
```

Напишемо заключне правило (предикат):

```
p :- good_math(L),print_list(L,1).
```

Контрольні запитання

- Визначення списку, правила запису списку.
- Рекурсивна обробка списків.
- Навести приклади процедур обробки списків.
- Обробка структур, способи вибору потрібних даних з структури.
- Вбудовані предикати накопичення рішень у списках.

Лабораторна робота № 8

ВВЕДЕННЯ-ВИВЕДЕННЯ І РОБОТА З БАЗОЮ

Цілі роботи: ознайомлення з предикатами введення-виведення у мові Prolog; вивчення та одержання практичних навичок з формуванням та змінні бази даних програми.

Завдання: завантажити з файлу список фактів, виконати дії відповідно завдання (табл. 8.1.), а по закінченні сеансу роботи зберегти стан бази даних в файл.

Таблиця 8.1. **Варіанти завдань**

№ варіанту	Зміст завдання
1	Ввести будь яке число і знайти найближче від нього число, яке кратне числу 13. Запам'ятати знайдене число в базі даних програми, виключаючи дублювання
2	Створити українсько-англійський двосторонній словник. При введенні англійського слова відповісти відповідним українським і навпаки. Якщо слово відсутні в словнику, процедура повинна запитати переклад і запам'ятати його.
3	Процедура при першому зверненні запитує ім'я користувача, при другому – числової інтервал, у подальшому – запит числа, яке зберігається, якщо належить інтервалу. За командою створюється терм, який містить ім'я і список чисел, які були введені, а потім все починається знову
4	Виріб складається з трьох деталей a, b і c. Якщо на склад надійшли всі деталі, вони забираються для складання виробу. Деталі з'являються на склад випадково. За командою показати стан складу деталей і готових виробів.
5	Введення і коригування інформації про робітників підприємства: прізвище, день народження, посада, оклад. Одночасно формувати список посад і відповідний оклад.
6	Довідкова відповідає на запитання про розташування об'єктів міста – назва вулиці і номер дома. Якщо інформація відсутня, необхідно запитати адресу. Одночасно формувати список вулиць міста.
7	У автопарку за 5000 км пробігу всі автомобілі проходять технічний огляд. Щодня реєструється денний пробіг. Спрогнозувати, які автомобілі повинні наступного дня пройти технічний огляд. За командою відмітити проходження техогляду.
8	Ввести цифру і сформувати всі можливі двозначні числа з цифрами, які існують в базі даних програми. Запам'ятати введену цифру і список чисел. Якщо цифра введена повторно, замінити існуючий список чисел.
9	Сформувати базу фактів елемент (X) за правилом, яке залежить від номеру події введення: перші три факти заносяться безумовно, далі непарні заносяться у початок, парні – в кінець бази.
10	Ввести будь-який терм та вивести на екран всі терми бази даних, з якими зіставляється введений терм. Якщо подібні терми не знайдені, введений терм додати до бази.
11	Створити програму для вибору персоналу за критеріями, які розраховуються за результатами відповідей на питання анкети.

Продовж. табл. 8.1

<table>
<tr><th>№ варіанту</th><th>Зміст завдання</th></tr>
<tr><td>12</td><td>Програма з початку запрошує ім’я, потім переходить до введення термів об’єкт(X). Всі наступні терми програма додає до бази тільки у тому випадку, якщо тип об’єкту X співпадає з типом першого об’єкту. Введення термів закінчується при введені терму end. За командою вивести всіх користувачів зі списком об’єктів.</td></tr>
<tr><td>13</td><td>Програма управляється меню:
1.Введення числа
2.Введення правила.
3.Вихід.
Правило визначає числа, допустимі для введення. Правило можна змінити будь-коли і зберігається для наступного сеансу.</td></tr>
<tr><td>14</td><td>Ввести предикат виду p(Term), де Term – будь-який терм. Якщо всі аргументи терму конкретизовані, терм додається до бази, якщо ні – виконується спроба конкретизувати їх. Якщо конкретизація неможлива, програма пропонує змінити предикат.</td></tr>
<tr><td>15</td><td>Ввести правило, за якою з бази віддаляються тимчасово факти, що задовольняють правилу. За командою відновити базу.</td></tr>
<tr><td>16</td><td>Ввести число і додати його до бази у випадку, якщо число знаходиться в інтервалі між мінімальним та максимальним числами у базі. Дублювання чисел не дозволяється. Програма закінчується, якщо введені всі числа у заданому інтервалі.</td></tr>
<tr><td>17</td><td>Ввести будь-яке слово, створити паліндром і додати до бази, виключаючи дублювання.</td></tr>
<tr><td>18</td><td>Реалізувати гру «слова». Програма повинна «навчатися грати» за кожним сеансом.</td></tr>
<tr><td>19</td><td>Побудувати автомат F(X,Y), де X вибирається з черги, Y – зі стеку. Черга і стек можуть бути поповнені у будь-який час.
<table>
<tr><th>X</th><th>Y</th><th>Стан</th></tr>
<tr><td>1</td><td>1</td><td>a</td></tr>
<tr><td>1</td><td>0</td><td>b</td></tr>
<tr><td>0</td><td>1</td><td>c</td></tr>
<tr><td>0</td><td>0</td><td>d</td></tr>
</table></td></tr>
<tr><td>20</td><td>Створити программу для лікаря, яка може визначити імовірний діагноз за вказаними симптомами в умовах неповної визначеності.</td></tr>
</table>

ТЕОРЕТИЧНІ ВІДОМОСТІ

Предикати введення-виведення. Prolog-програма може читати дані з вхідного потоку і записувати в вихідний потік. У якості вхідного та вихідного потоку може бути файл або термінал (клавіатура і монітор відповідно). Дані, які введені з клавіатури і дані, які виведені на монітор, розглядаються як потік, якій має назву `user`. У будь який час існує один поточний вхідний і один вихідний потік. За замовчування поточним потоком є термінал, однак його можна змінити на файл, а після закінчення роботи з файлом відновити потік `user`.

Для зміни вхідного потоку використовується предикат

```
see(<імя файлу>).
```

Якщо вказаний файл існує, ціль see успішна, поточним вхідним потоком становиться файл і всі наступні операції читання здійснюються з файлу. Для відновлення потоку user необхідно застосувати предикат see(user) або предикат seen без параметрів.

Таким чином, для читання даних з будь-якого файлу потрібно виконати наступні дії:

```
…, see(<файл>), <читання з файлу>, see(user), …
```

Аналогічно, для запису даних використовується наступна послідовність цілей:

```
…, tell(<>), <запис в файл>, tell(user), …
```

Основні предикати введення-виведення представлені в таблиці 8.2. Всі предикати поділені на такі групи [1, 2]: предикати введення-виведення термів; предикати читання та запису символів; допоміжні предикати.

Таблиця 8.2. **Предикати введення-виведення**

Предикати введення-виведення термів	
`read(X)`	Введення терму з поточного вхідного потоку
`write(X)`	Виведення терму у поточний вихідний потік
`writeq(X)`	Виведення у лапках величини, які мають бути виведені таким чином
`display(X)`	виводить терм X в стандартній дужковій префіксній формі запису
Предикати введення-виведення символів	
`get(X)`	Читання символу, пропускаючи недруковані символи
`get0(X)`	Читання будь-якого символу
`skip(X)`	Читання символів доти, доки не знайдеться символ, вказаний у дужках

Продовж. табл. 8.2

	Предикати введення-виведення символів
`put(X)`	Запис у вихідний потік символу з кодом X. (X може бути цілочисловим виразом)
	Допоміжні предикати
`tab(X)`	У вихідний потік виводиться X пробілів, X може бути цілочисловим виразом
`nl`	Перехід на новий рядок

Предикати для роботи з базою даних програми. У процесі виконання програми зміст бази даних може бути змінений. Для цього існують предикати (табл. 8.3.), які дозволяють додавати або вилучати твердження із бази даних.

Таблиця 8.3. **Предикати для роботи з базою даних**

Назва	Призначення
assert(X) assertz(X)	розміщує нове твердження X в кінці бази даних, або додає його до подібних за арністю та функтором тверджень за умови їх присутності
asserta(X)	розміщує нове твердження X у початок бази даних
retract(X)	вилучає задане твердження X із бази даних
abolish(P/A)	вилучає всі подібні твердження з заданими функтором P і арністю A

МЕТОДИЧНІ ВКАЗІВКИ

Динамічні предикати. Всі факти бази даних за умовчання статичні і Prolog система не дозволяє змінити їй стан [2]. Для додавання або вилучення предикатів з бази даних необхідно відповідні предикати об'явити як динамічні за допомогою вбудованого предикату `dynamic (P/A)`, де `P` – імя предикату, що може бути змінений, `A` – його арність.

Особливості роботи з файлами. Використання будь-якого предикату читання з файлу припускає, що файл існує. Щоб запобігти виникненню помилки у разі, якщо файл не існує, слід застосувати предикат `exist_file(F)`, де `F` – ім'я файлу. Предикат є вдалим, якщо файл `F` існує.

При читанні даних з файлу іноді необхідна перевірка кінця файлу. У мові Prolog існує атом `end_of_file`, що дає можливість виконати

перевірку кінця файлу після кожного читання. Фрагмент програми, представлений нижче, демонструє подібну перевірку:

```
p :- read(X), checkread(X).
checkread(end_of_file) :- write('Find end of file.'),
!.
checkread(X) :- <Обробка терму X>.
```

Перевірка конкретизації змінної. У програмі іноді важливо визначити стан конкретизації змінної виконання подальших дій [2, 4]. Предикат var(X) є вдалим, якщо X – неконкретизована змінна. Предикат nonvar(X) є вдалим, якщо X – конкретизована змінна. Приклад застосування цих предикатів представлений нижче.

Приклад виконання завдання. Задача: скласти програму–довідник розкладу руху автобусів. Програма повинна додавати до бази нові маршрути, якщо запит неможливо задовольнити.

```
% Факти представлені предикатом ruh(S,L),
% де S – назва міста, L – список часу відправлення автобуса
                :-dynamic ruh/2, ruhtmp/2.
% У разі виклику з невизначеною назвою міста – pruh(_) – виведення
стану розкладу.
              pruh(X):-var(X),listing(ruh/2),!.
% У разі виклику з визначеною назвою міста, який присутній у роз-
кладі -
% виведення списку часу відправлення за допомогою процедури
writeruh
              pruh(X):-ruh(X,L),writeruh(X,L),!.
% Якщо місто не визначено у розкладі:
% – введення часу відправлення ( gettime )
% – отримання часу відправлення у вигляду списку, який є отсор-
тованим ( setof )
% – додавання нового факту до бази assert(ruh(X,L))
% – видалення з бази тимчасових предикатів ruhtmp, які були ство-
рені у процедурі gettime
        pruh(X):- gettime(X),setof(T,ru(X,T),L),
      assert(ruh(X,L)),abolish(ruhtmp/2),pruh(X).
% використовується виключно для організації циклу введення часу
відправлення.
```

% створює в бази тимчасові факти ***ruhtmp(Місто,Час_видправлення).***

```
gettime(S):-write(S),nl,gettime1(S).
```

% реалізація циклу введення часу відправлення спільно з процедурою obr

```
gettime1(S):-write('Time  (hh:mm)
    :'),nl,read(X),obr(X,S).
```

% введення атому end – завершує цикл введення.

```
obr(end,_):-true.
obr(X,S):-assert(ruhtmp(S,X)),gettime1(S).
```

% друкування списку часу відправлення за допомогою рекурсії

```
writeruh(X,L):-nl,write(X),nl,writelist(L).
writelist([]):-!.
writelist([X|L]):-write(X),nl,writelist(L).
```

Контрольні запитання

- Предикати запису у вихідний потік.
- Предикати зчитування з вхідного потоку.
- Вбудовані предикати роботи з файлом.
- Предикати додавання термів до бази даних програми.
- Предикати видалення термів з бази даних програми.
- Вбудовані предикати перевірки конкретизації змінних.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. *Братко И.* Программирование на языке Пролог для искусственного интеллекта: Пер. с англ. – М.: Мир, 1990. – 560 с.

2. *Братко И*. Алгоритмы искусственного интеллекта на языке PROLOG, 3-е издание. : Пер. с англ. – М. : Издательский дом "Вильямс", 2004. – 640 с.

3. *Глибовець М.М., Кравченко І.В., Олецький О.В., Терещенко В.М.* Програмування в Пролозі: Навч. посібник. – К.: Київський університет, 1998. – 110 с.

4. *Малпас Дж*. Реляционний язык Пролог и его применение: Пер. с англ. – М.: Наука. Гл. ред. физ.-мат. лит., 1990. – 464 с.

5. *Филд А., Харисон П.* Функциональное программирование: Пер. с англ. – М: Мир, 1993, 637 с., ил.

6. *Хендерсон П.* Функциональное программирование. Применение и реализация: Пер. с англ. – М.: Мир, 1983. – 349 с.

7. *Хювенен Э., Сеппянен Й*. Мир Лиспа. В 2-ч т. Т.1: Введение в язык Лисп и функциональное программирование: Пер. с финск. – М.: Мир, 1990. – 447 с.

8. *Устенко С.А*. Функціональне та логічне програмування. Частина 1. Функціональне програмування: Навч. посібник. – Миколаїв: УДМТУ, 1998. – 60 с.

9. *Устенко С.А*. Функціональне та логічне програмування. Частина 2. Логічне програмування: Навч. посібник. – Миколаїв: УДМТУ, 1998. – 49 с.

ЗМІСТ

Лабораторная робота № 1 3
ОБРОБКА СПИСКІВ У МОВІ LISP 3
ТЕОРЕТИЧНІ ВІДОМОСТІ 3
МЕТОДИЧНІ ВКАЗІВКИ 9
Контрольні запитання 11
Лабораторна робота № 2 11
ВИЗНАЧЕННЯ ФУНКЦІЇ ТА РЕКУРСІЯ 11
ТЕОРЕТИЧНІ ВІДОМОСТІ 13
МЕТОДИЧНІ ВКАЗІВКИ 16
Лабораторна робота № 3 21
ФУНКЦІОНАЛИ 21
ТЕОРЕТИЧНІ ВІДОМОСТІ 22
МЕТОДИЧНІ ВКАЗІВКИ 26
Контрольні запитання 27
Лабораторная робота № 4 27
ТИПИ ДАНИХ 27
ТЕОРЕТИЧНІ ВІДОМОСТІ 29
МЕТОДИЧНІ ВКАЗІВКИ 33
Контрольні запитання 33
Лабораторна робота № 5 34
СКЛАДАННЯ ПРАВИЛА НА МОВІ PROLOG 34
ТЕОРЕТИЧНІ ВІДОМОСТІ 34
МЕТОДИЧНІ ВКАЗІВКИ 39
Контрольні запитання 40
Лабораторна робота № 6 40
УПРАВЛІННЯ ПРОГРАМОЮ 40
ТЕОРЕТИЧНІ ВІДОМОСТІ 40
МЕТОДИЧНІ ВКАЗІВКИ 43
Контрольні запитання 43
Лабораторна робота № 7 43
СПИСКИ 43
ТЕОРЕТИЧНІ ВІДОМОСТІ 44
МЕТОДИЧНІ ВКАЗІВКИ 46
Контрольні запитання 48

Лабораторна робота № 8 48
ВВЕДЕННЯ-ВИВЕДЕННЯ І РОБОТА З БАЗОЮ 48
ТЕОРЕТИЧНІ ВІДОМОСТІ 51
МЕТОДИЧНІ ВКАЗІВКИ 52
Контрольні запитання 54
Список використаної літератури 55

Навчальне видання

ПАРТАС Віктор Кирилович
ГАЙДАЄНКО Оксана Володимирівна

Функціональне та логічне програмування
Методичні вказівки для студентів напрямку "Комп'ютерні науки"

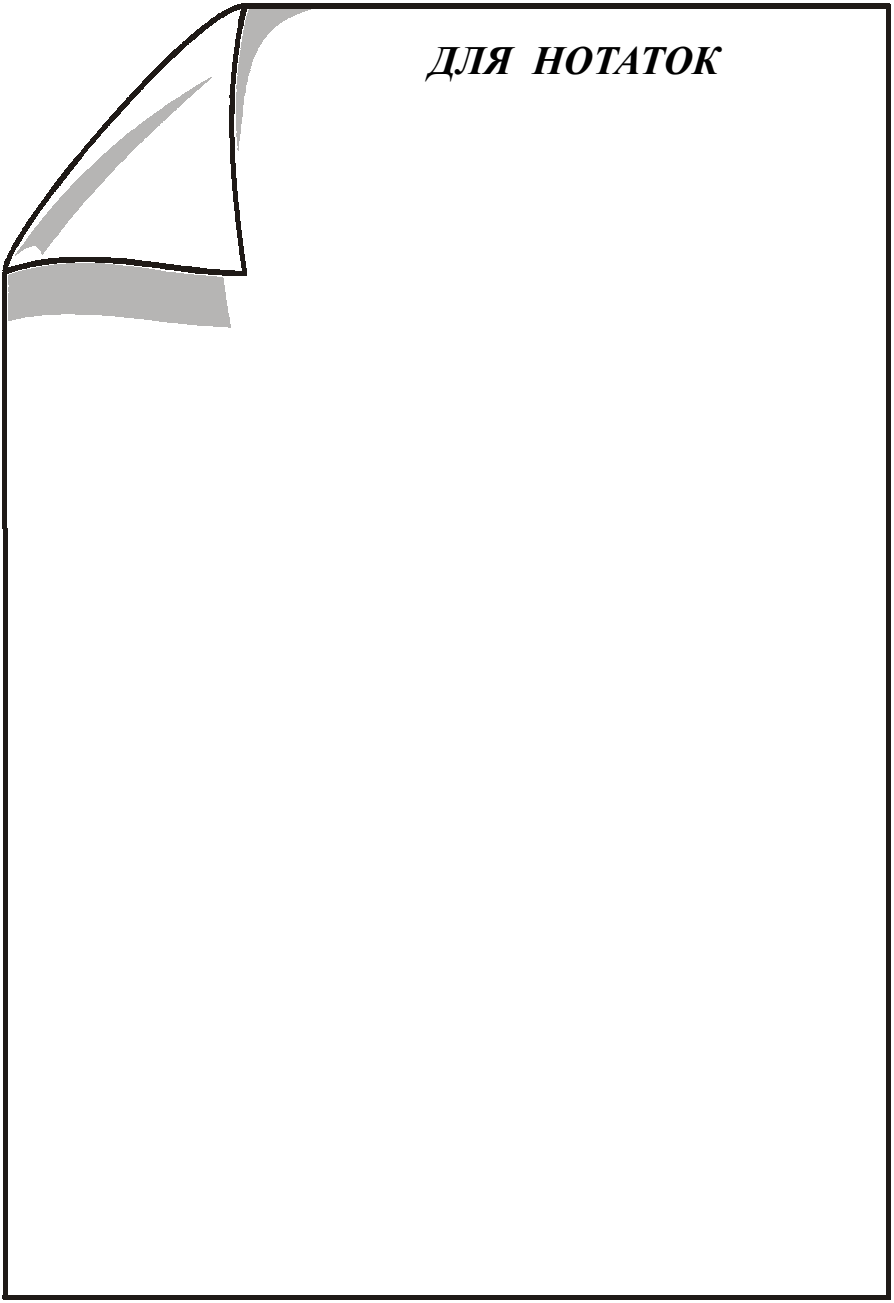
(*українською мовою*)

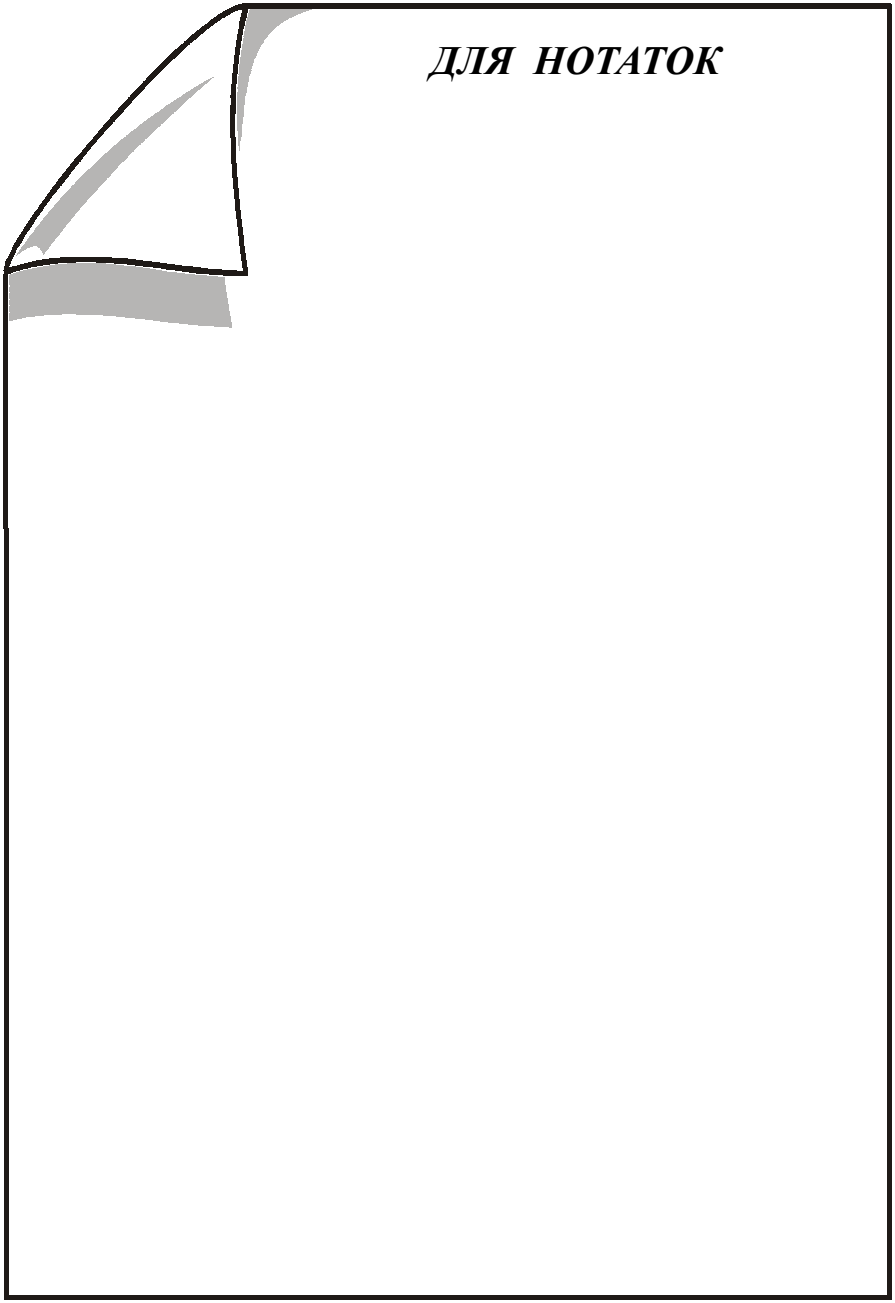
Комп'ютерна верстка *В.Г. Мазанко*
Коректор *М.О. Паненко*

Свідоцтво про внесення суб'єкта видавничої справи до Державного
реєстру видавців, виготівників і розповсюджувачів видавничої продукції
ДК № 2506 від 25.05.2006 р.

Підписано до друку 10.11.09. Папір офсетний. Формат 60×84/16.
Друк офсетний. Гарнітура "Таймс". Ум. друк. арк. 3,4. Обл.-вид. арк. 3,6.
Тираж 100 прим. Вид. № 43. Зам. № 322. Ціна договірна.

Видавець і виготівник Національний університет кораблебудування,
54002, м. Миколаїв, вул. Скороходова, 5


ДЛЯ НОТАТОК


ДЛЯ НОТАТОК