Національний педагогічний університет імені М.П. Драгоманова

І.С. Мінтій

Схематичне програмування

Початки програмування: функціональний підхід

Науковий редактор
дійсний член НАПН України
М.І. Жалдак

Київ
2010

УДК 004.42 (075.8)
ББК 32.97
М-62

Мінтій І. С.
М-62 Схематичне програмування (початки програмування: функціональний підхід) / За ред. академіка НАПН України М. І. Жалдака. – К. : НПУ імені М. П. Драгоманова, 2010. – 147 с.

Посібник призначений для вивчення розділу «Вступ до програмування» курсу інформатики в процесі формування у студентів педагогічних університетів компетентності в програмуванні на основі функціонального підходу.

В якості засобів формування компетентності в програмуванні обрані мова програмування Scheme та середовище програмування DrRacket. Посібник містить необхідний теоретичний матеріал, що супроводжується лабораторними роботами, індивідуальними завданнями та проектами із застосування моделей та методів математичної інформатики. У додатках наведено опис локалізованої версії DrRacket та розглянуто її додаткові можливості.

Для студентів молодших курсів інформатичних спеціальностей, вчителів інформатики та учнів профільних класів.

Рецензенти:

Ю. В. Триус, доктор педагогічних наук, професор, професор кафедри комп'ютерних технологій Черкаського державного технологічного університету;

С. А. Раков, доктор педагогічних наук, професор, професор кафедри інформатики Харківського національного педагогічного університету імені Г. С. Сковороди

Друкується згідно з рішенням ученої ради Національного педагогічного університету імені М. П. Драгоманова (протокол № 12 від 26 червня 2009 р.)

ЗМІСТ

Вступ

Scheme – невеликий і елегантний діалект мови Lisp – інтерпретована функціональна мова програмування високого рівня. Перший варіант мови Scheme був створений Дж. Дж. Сассманом та Г. Стілом в Массачусетському технологічному інституті у 1975 р. як «простий та конкретний експериментальний інструмент для досліджень в галузі семантики та стилю програмування» [8]. Найбільш інтенсивна розробка ядра мови відбувалась у період з 1975 по 1980 рр.

Серед переваг Scheme можна назвати як такі, що характеризують її як функціональну мову програмування, так і її власні:

– відкладені обчислення;

– стислість і простота;

– модульність;

– використання функцій як значень;

– чистота;

– невеликий розмір ядра мови;

– мобільність програм;

– можливість використання в діалоговому режимі;

– зручність для розв’язування математичних задач;

– підтримка багатьох парадигм (функціональної, об’єктно-орієнтованої та імперативної).

Інтерпретатор Scheme та стандартні бібліотеки доступні на всіх основних платформах, тобто сам інтерпретатор є мобільним.

Вказані переваги стали причиною застосування мови Scheme як у вступних курсах інформатики, так і при розгляді загальних принципів програмування у провідних ВНЗ світу (за матеріалами журналу Times [10]): Гарвардський університет, Сполучені Штати Америки (США); Єльський університет, США; Чиказький університет, США; Массачусетський технологічний університет, США; Каліфорнійський технологічний університет, США; Токійський технологічний інститут, Японія; Гонконгзький університет науки та технологій, Гонконг; університет Торонто, Канада; Національний університет Сінгапуру, Сінгапур та ін. На рівні середніх навчальних закладів Scheme також активно впроваджується, витісняючи, таким чином, традиційні мови програмування Basic, Pascal та C [7].

Саме тому при розробці розділу «Вступ до програмування» курсу інформатики для студентів молодших курсів інформатичних спеціальностей вищих педагогічних навчальних закладів нами було обрано мову програмування Scheme.

Домовленості щодо шрифтів та позначень:

– звичайний, Courier – виокремлені тексти визначень функцій, ключові слова Scheme – (`(define ім'я значення)`);

– напівжирний, Courier – виокремлені результати обчислень у вікні інтерпретатора (**`10`**);

– знаком > починається введення користувача у вікні інтерпретатора (`> (define сторона 4)`);

– напівжирний, в рамці – виокремлені назви кнопок та пунктів меню DrRacket (**Виконати**).

Вибір варіанту індивідуальних завдань

Для визначення варіанту завдань (в індивідуальних завданнях) необхідно обрати в першому стовбці таблиці із завданням свій номер в списку групи і виконати завдання з другого стовбця.

1. Основи синтаксису Scheme

1.1. Опис мови

Програми, описані мовою Scheme, складаються з ключових слів, змінних, символів (чисел, знаків, рядків, векторів, списків та ін.), пропусків (пробіл, табуляція, новий рядок) та коментарів.

Сукупність ключових слів, змінних та символів називається *ідентифікаторами*. Ідентифікатори можуть складатися з таких знаків:

– букв;
– цифр 0 .. 9;
– знаків ? ! . + – * / < = > : $ % ^ & _ ~ @.

Ідентифікатор не може складатися лише з цифр.

Наприклад:

`сума`, `квадрат`, `A`, `Б`, `FN`, `F_N`, `3f`, `!a` – ідентифікатори.

111, .3, F N – не ідентифікатори.

Ідентифікатори повинні бути розділені пропусками, круглими дужками, подвійними лапками або знаком «;», який позначає коментар. *Коментар* – частина програми, яка пояснює роботу програми та не впливає на її виконання (початок коментаря – «;», кінець – кінця рядка:

```
(+ 2 3 4); це коментар
```

Обмеження стосовно довжини ідентифікатора визначаються реалізацією мови. Зазвичай програмісти використовують стільки символів, скільки необхідно. Але довгі ідентифікатори не замінюють коментарів, та й до того ж ускладнюють програму і її стає складно читати.

1.2. Запис виразів у Scheme

У Scheme використовується *префіксна нотація* (прямий польський запис). Тобто, запис 3 + 4 в префіксній нотації матиме вигляд `(+ 3 4)`. Scheme, як і більшість функціональних мов програмування, працює зі списками, що записуються в круглих дужках. Крайній зліва елемент списку – функція, інші елементи – аргументи функції.

В математиці ми звикли, що функція записується у вигляді $y=f(x)$, в деяких випадках, наприклад, коли x – простий вираз, дужки опускаються, а інколи замість дужок використовуються інші знаки. Наприклад: $y=\sin x$, $y=x+3$ і $y=\sin(x+3)$, y=|x| та $y=[x]$. Вико-

ристання префіксної форми запису функцій надає можливість уніфікувати запис функцій та їх аргументів. Інші переваги префіксної нотації:

– вираз може містити більше, ніж 2 аргументи: `(+ 2 3 4)`; `(* 5 6 9)` (таким чином, префіксна нотація є більш компактною, ніж інфіксна, у якій знак записується між аргументами);

– при обчисленні виразів не потрібно враховувати пріоритет операцій – функція разом з аргументами заключена в дужки: `(* (+ 3 4) (/ 2 7))`;

– порядок обчислень визначається групуванням дужок;

– вона природно розширюється, в результаті чого отримуються вкладені списки (композиції функцій), елементами яких також є списки:

```
(+ (/ 4
        (- 6 3 4))
     (* 4 5 2))
```

Дужки використовуються для визначення того, що має бути обчислено. В багатьох мовах програмування зайві дужки не впливають на виконання програми, інша справа Scheme: зайві дужки можуть призвести або ж до помилки або ж до неправильного результату. Так, спробуйте обчислити `((+ 2 3 4))`.

Запитання і завдання

1. Які з наведених виразів є ідентифікаторами:

– замінити;
– +додати;
– сума квадратів;
– квадрат-суми;
– .крапка;
– -знайти від'ємні числа;
– модуль;
– 4слова.

2. Наведіть приклади виразів, що є ідентифікаторами і навпаки.

3. Використовуючи префіксну нотацію, запишіть наступні вирази:

а) $\dfrac{6+3+(9-12*(4-3/4))}{2-(3+6)(4+5)}$; б) $2-5*\dfrac{7+14+17-6}{2-6*12+10}$.

1.3. Визначення змінних та функцій

У Scheme немає відмінностей між змінними і функціями: функції є об'єктами першого порядку, тобто їх теж можна передавати

іншим функціям в якості аргументів.

1.3.1. Вираз `define`

Для створення змінних використовують форму `define`:

```
    (define ім'я значення)
> ; визначення змінної a зі значенням 10
     (define a 10)
> a
10
```

Визначення `define` зв'язує ім'я зі значенням, тоді визначена змінна – зв'язана.

1.3.2. Вираз `set!`

Надання змінній нового значення виконується за допомогою `set!`:

```
> ; надання змінній a значення 3
>    (set! a 3)
> a
3
```

За допомогою `set!` можна змінити значення змінної тільки у випадку, якщо вона вже була визначена. Так, спробуйте обчислити:

```
> (set! x 4).
```

Який результат отримано? Чому?

Для створення (визначення) функцій використовують форму `lambda`:

```
    (lambda (формальні_параметри) тіло)
> ; створення функції, яка повертає куб введеного числа
    (lambda (x) (* x x x))
#<procedure>
> ; застосування функції до аргументу 3
    ((lambda (x) (* x x x)) 3)
27
```

Якщо ми хочемо назвати функцію, яку описує `lambda`-вираз, необхідно використати форму `define`, що зв'язує між собою ім'я функції та її `lambda`-вираз:

```
    (define ім'я
        (lambda (формальні_параметри) тіло))
```

Даний запис можна скоротити, видаливши слово `lambda` та дужки початку й кінця списку. В такій формі `define` отримує в якості параметрів дві частини. Перша – вираз, що складається з

імені функції та списку її формальних параметрів, а друга – тіло функції:

```
(define (ім'я формальні_параметри)
    тіло)
```

Формальність параметрів означає, що при визначенні функції їх можна замінити на будь-які інші символи і це не відобразиться на дії функції.

Приклади:

```
> ; визначення функції квадрат,
  ; яка повертає квадрат уведеного числа x
  ; (x – формальний параметр)
    (define (квадрат x)
      (* x x))
> ; виклик функції квадрат з аргументом 3
     (квадрат 3)
9
> ; визначення функції квадрат,
  ; яка повертає квадрат уведеного числа a
  ; (a – формальний параметр)
    (define (квадрат a); в даному випадку
      (* a a))
> (квадрат 3)
9
```

Визначену функцію далі можна використовувати в інших функціях так само, як і будь-яку іншу стандартну функцію:

```
> ; визначення функції,
  ; що повертає суму квадратів своїх аргументів
    (define (сума-квадратів x y)
      (+ (квадрат x)
         (квадрат y)))
> (сума-квадратів 3 4)
25
```

За допомогою `define` створюють *глобальні змінні та функції*, які будуть визначені доти, доки не буде закінчено поточний сеанс роботи інтерпретатора.

Для створення *локальних змінних*, які будуть визначені тільки протягом виконання певної функції, можна скористатись формами `let` або `let*`.

Наприклад, функція, що розв'язує квадратне рівняння $ax^2+bx+c=0$, як аргументи буде приймати значення змінних a, b, c.

Корені рівняння шукаються за формулою $x_{1,2} = \frac{-b \pm \sqrt{D}}{2a}$, де D – дискримінант рівняння: $D = b^2 - 4ac$. Тому в процесі розв'язування квадратного рівняння бажано було б створити змінну дискримінант. Це можна зробити, створивши допоміжну функцію, яка знаходить дискримінант квадратного рівняння:

```
> (define (корені-рівняння a b c)
    (define дискримінант
        (- (квадрат b)
            (* 4 a c)))
; в даний момент зосередимо увагу
; саме на визначенні локальної змінної дискримінант
; і не розглядатимемо випадки, коли дискримінант менше 0
    (cons (/ (+ (- b) (sqrt дискримінант))
             (* 2 a))
; функція cons приймає два аргументи
; і повертає пару, створену з них
          (/ (- (- b) (sqrt дискримінант))
             (* 2 a))))
> (корені-рівняння 1 8 1)
(-0.12701665379258298 . -7.872983346207417)
```

Інший спосіб – створити функцію без імені для зв'язування локальних змінних як `lambda`-вираз. В цьому випадку тіло функції корені-рівняння є просто викликом цієї функції:

```
> (define (корені-рівняння a b c)
    ((lambda (дискримінант)
       (cons (/ (+ (* -1 b)
                   (sqrt дискримінант))
                (* 2 a))
             (/ (- (* -1 b)
                   (sqrt дискримінант))
                (* 2 a))))
     (- (квадрат b)
        (* 4 a c))))
```

1.3.3. Вираз `let`

Така конструкція виявивилась настільки корисною, що для неї є спеціальна форма `let`, яка робить її ще зручнішою. З використанням `let` функцію корені-рівняння можна записати так:

```
> (define (корені-рівняння a b c)
    (let ((дискримінант (- (квадрат b)
```

```
                          (* 4 a c))))
   (cons (/ (+ (* -1 b) (sqrt дискримінант))
            (* 2 a))
         (/ (- (* -1 b) (sqrt дискримінант))
            (* 2 a)))))
```

Загальна форма виразу `let`:

```
(let ((змінна1 вираз1)
      (змінна2 вираз2)
       ...
      (змінnan виразn))
    тіло)
```

Це можна розуміти так:

(Нехай ((змінна1 має значення вираз1)
(і змінна2 має значення вираз2)
...
(і змінnan має значення виразn))
в тілі)

Перша частина `let`-виразу – список пар вигляду ім'я–значення. Коли обчислюється `let`, кожне ім'я зв'язується зі значенням відповідного виразу, тобто створюються змінні; область їх дії (визначення) – тіло `let`.

Приклад 1:

```
> (let ((x 2) (y 10))
    (+ x y))
12
> ((lambda (x y) (+ x y)) 2 10)
12
```

Приклад 2:

```
> (define x 10)
> (+ (let ((x 5))
       (* x (+ x 2)))
     x)
45
```

Тіло `let` є виразом `(* x (+ x 2))`. Значення змінної `x` в тілі `let` – 5. Тобто, значенням `let`-виразу є 35. Тепер до цього значення додається значення змінної `x`. Ця змінна `x` не має нічого спільного зі змінною `x`, яка зв'язана в `let`-виразі. Випадково вони носять одне й те саме ім'я, але мають різні області визначення. Значення змінної `x` за межами `let`-виразу – 10. Тобто, шуканим значенням всього виразу є (+ 35 10) = 45.

Приклад 3:

```
> (define x 10)
> (let ((x 5)
        (y (* x 2)))
    (+ x y))
25
```

Змінна `x` зв'язана зі значенням 5, а змінна `y` має значення 20, а не 10, тому що `x` для виразу `y` обчислюється за межами `let`-виразу, де глобальна змінна `x` має значення 10.

`let` використовується для створення лише локальних змінних, а `define` –локальних і глобальних.

1.3.4. Вираз `let*`

Запис виразу `let*` ідентичний `let`:

```
(let* ((змінна1 вираз1)
       (змінна2 вираз2)
       ...
       (змінnan виразn))
    тіло)
```

Але результат, однак, інакший. `let*` еквівалентний послідовності вкладених `let`-виразів:

```
(let ((змінна1 вираз1))
    (let (змінна2 вираз2))
        (let...
            (let змінnan виразn))
                тіло) … )))
```

Значення кожної наступної змінної в `let*` може мати змінні від попереднього зв'язування. Запишемо наведений вище приклад 3 з використанням `let*`:

```
>(define x 10)
>(let* ((x 5)
        (y (* x 2)))
    (+ x y))
15
```

Значення `let*`-виразу 15, оскільки при обчисленні значення змінної `y` значення змінної `x` – 5.

1.4. Уведення/виведення

За допомогою операцій уведення/виведення можна здійснювати обмін даними між програмою та зовнішніми пристроями. В системі введення/виведення Scheme узагальнений пристрій уве-

дення/виведення називається потоком (портом). Стандартним зовнішнім пристроєм є термінал. На терміналі можна надрукувати певні відомості. Можна ввести дані з терміналу, надрукувавши їх на клавіатурі.

Потоки бувають бінарні (двійкові) та текстові.

Бінарний потік – це послідовність байтів, що взаємно однозначно відповідає байтам на зовнішньому пристрої, перетворення символів не відбувається. Кількість байтів, що пишуться (читаються), і тих, що зберігаються на зовнішньому пристрої, однакова (в кінці бінарного потоку може додаватись визначена додатком кількість нульових байтів для заповнення вільного місця в блоці пам'яті незначущими даними, щоб вони точно заповнили сектор на диску).

Текстовий потік – це послідовність символів. Текстовий потік утворюється з рядків, кожен з яких закінчується символом нового рядка. В кінці останнього рядка цей символ не є обов'язковим. В текстових потоках немає однозначної відповідності між символами, що записуються (читаються), і символами, які зберігаються на зовнішньому пристрої.

Для введення/виведення даних з/у певний файл необхідно відкрити його для запису/читання та зв'язати з ним потік введення/виведення. Після завершення роботи з файлом його потрібно закрити.

Наступний приклад демонструє, як після запуску функції на термінал виводиться повідомлення «Введіть число, для якого необхідно обчислити x^3+3x+4: ». Далі виконання програми зупиняється і система переходить в режим очікування введення з терміналу значення змінної `x`. Після завершення введення (натиснення клавіші вводу), програма видає результат на термінал.

```
> (define (обчислити)
    (printf "Уведіть число, для якого необхідно обчисли-
ти x^3+3x+4: ")
    (let ((x (read)))
      (+ (* x x x)
         (* 3 x)
         4)))
> (обчислити)
Уведіть число, для якого необхідно обчислити x^3+3x+4: 3
40
```

Якщо користувач випадково ввів не одне, а декілька значень, інші значення будуть зчитані при наступому виклику функції read:

```
> (обчислити)
Уведіть число, для якого необхідно обчислити x^3+3x+4: 3 4
2
40
> (обчислити)
Уведіть число, для якого необхідно обчислити x^3+3x+4: 80
> (обчислити)
Уведіть число, для якого необхідно обчислити x^3+3x+4: 18
```

Уведення даних з файлу:

```
; відкриємо файл test.txt для читання
; та зв'яжемо його з потоком f
> (define f (open-input-file "test.txt"))
```

Вміст файлу test.txt: 2 3.

```
> (define (обчислити)
; в якості аргумента функції read вкажемо ім'я потоку f
    (let ((x (read f)))
      (printf "Число, для якого необхідно обчислити
x^3+3x+4: ~v\nРезультат:" x)
      (+ (* x x x)
         (* 3 x)
         4)))
> (обчислити)
Число, для якого необхідно обчислити x^3+3x+4: 2
Результат:18
> (обчислити)
Число, для якого необхідно обчислити x^3+3x+4: 3
Результат:40
> (обчислити)
Число, для якого необхідно обчислити x^3+3x+4:
Помилка: очікувався тип «число», отримано <eof>.
; закриємо потік уведення f
> (close-input-port f)
```

Функція read повертає в якості результату eof, якщо досягнуто кінець файлу.

Виведення даних у файл:

```
; відкриємо файл test2.txt для запису
; і зв'яжемо його з потоком f
> (define f (open-output-file "test2.txt" #:mode 'text
#:exists 'replace))
```

```
> (define (обчислити)
    (printf "Введіть число, для якого необхідно обчисли-
ти x^3+3x+4 ")
    (let ((x (read)))
; в якості аргументу функції fprintf
; вкажемо ім'я потоку f
      (fprintf f "Результат - ~v\n"
               (+ (* x x x)
                  (* 3 x)
                  4))))
> (обчислити)
Уведіть число, для якого необхідно обчислити x^3+3x+4 2
```

Вміст файлу test2.txt:

Результат – 18

```
> (обчислити)
Уведіть число, для якого необхідно обчислити x^3+3x+4 3
> (обчислити)
```

Вміст файлу test2.txt:

Результат – 18

Результат – 40

```
; закриємо потік виведення f
> (close-output-port f)
```

Функції введення/виведення:

Функція	Дія
`(read [in])`	вилучає з потоку введення одне дане та повертає його в якості результату

`(read-char [in])`	вилучає з потоку введення перший знак та повертає його.
	Наприклад:
	`; введення з файлу test1.txt`
	`; вміст файлу: HELLO`
	`> (define f (open-input-file "test1.txt"))`
	`> (read-char f)`
	`#\H`
	`> (read-char f)`
	`#\e`
	`> (read-char f)`
	`#\l`
	`> (close-input-port f)`

Функція	Дія

	`; введення з терміналу`
	`> (read-char)`
	`7`
	`#\7`

`(read-byte [in])`	повертає код першого знаку з потоку введення.
	Наприклад:
	`> (read-byte)`
	`1`
	`49`

`(read-line [in mode])`	повертає перший рядок з потоку введення; знаки зчитуються, доки не зустрінеться кінець рядка або кінець файлу.
	Наприклад:
	`> (read-line)`
	`1 2 3 4 5 6 7 8 9 10`
	`"1 2 3 4 5 6 7 8 9 10"`

`(read-string n [in])`	повертає рядок з `n` знаків із потоку введення.
	Наприклад:
	`> (read-string 10)`
	`1 2 3 4 5 6 7 8 9 10`
	`"1 2 3 4 5 "`

`(read-string! str [in start-pos end-pos])`	рядку `str` надається значення вилученого з потоку введення рядка; необов'язковий аргумент – потік введення та позиція початку `start-pos` та кінця `end-pos` зчитування знаків.
	Наприклад:
	`> (define рядок (make-string 5))`
	`> (read-string! рядок)`
	`12345`
	`5`
	`> (write рядок)`
	`"12345"`

`(peek-char [in amt])`	знак зчитується, проте не вилучається з потоку введення, і при наступному виклику `peek-char` може бути прочитаний знову. Наприклад: `> (peek-char)`

Функція	Дія
	1234 **#\1** > (peek-char) #**\1**
(open-input-file path [#:mode mode-flag])	відкриває файл за адресою path для читання; аргумент mode-flag визначає тип файлу: 'binary – бінарний; 'text – текстовий; за замовчуванням, значення mode-flag – 'binary
(close-input-port in)	закриває порт виведення in
(write datum [out])	виводить дане datum в порт виведення out. Наприклад: > (write 'hi) **hi** > (write "hi") **"hi"** > (write 4) **4**
(display datum [out])	виводить дане datum в порт виведення out. Наприклад: > (display 'hi) **hi** > (display "hi") **hi** > (display 4) **4**
(print datum [out])	виводить дане datum в порт виведення out. Наприклад: > (display 'hi) **hi** > (display "hi") **hi** > (display 4) **4**
(printf form v ...)	здійснює форматоване виведення на термінал, де form – рядок, що складається з елементів двох видів, перший – знаки, що бу-

Функція	Дія

	дуть виведені, другий – це специфікатори перетворення, що визначають спосіб виведення аргументів vs:
	~n або ~% або \n – перехід на новий рядок;
	~a або ~A – наступний аргумент vs виводиться як аргумент функції display;
	~s або ~S – наступний аргумент vs виводиться як аргумент функції write;
	~v або ~V – наступний аргумент vs виводиться як аргумент функції print;
	~c або ~C – виведення знаку – наступного аргументу vs; якщо наступний аргумент vs – не знак, результат – помилка;
	~b або ~B – наступний аргумент vs виводиться як двійкове число; якщо наступний аргумент vs – не ціле число, результат – помилка;
	~o або ~O – наступний аргумент vs виводиться як вісімкове число; якщо наступний аргумент vs – не ціле число, результат – помилка;
	~x або ~X – наступний аргумент vs виводиться як шістнадцяткове число; якщо наступний аргумент vs – не ціле число, результат – помилка;
	~~ – виведення знаку «~».
	Наприклад:
	`> (define x 1)`
	`> (define y 5)`
	`> (printf "Значення змінної x - ~v;~nзначення змінної y - ~v." x y)`
	Значення змінної x- 1;
	значення змінної y- 5.
	`> (printf "Значення змінної y – ~b" y)`
	Значення змінної x – 101

(newline [out])	перехід на новий рядок
(open-output-file path	відкриває файл за адресою path для запису;

Функція	Дія
[#:mode mode-flag] #:exists exists-flag])	аргумент exists-flag визначає як опрацьовувати файл, якщо він існує: 'error – повідомити про помилку; 'replace – видалити старий файл і створити новий; 'truncate – видалити старі дані; 'must-truncate – видалити старі дані, якщо файл існує; якщо файл не існує – повідомити про помилку; 'update – відкрити файл для запису без видалення його попереднього змісту. Запис почати з початку файлу; якщо файл не існує – повідомити про помилку; 'can-update – та ж, що й 'update, але якщо файл не існує – створити файл; 'append – додати дані в кінець файлу; за замовчуванням, значення exists-flag = 'error
(close-output-port out)	закриває потік введення out
(fprintf out form v ...)	дія аналогічна дії функції printf, за виключенням того, що виведення здійснюється в потік виведення out
(with-output-to-file path proc [#:mode mode-flag #:exists exists-flag])	відкриває файл за адресою path для запису; результатом функції є дія функції proc над файлом; закриває файл
(with-input-from-file path proc [#:mode mode-flag])	відкриває файл за адресою path для читання; результатом функції є дія функції proc над файлом; закриває файл. Наприклад: > (with-output-to-file "test4.txt" (lambda () (printf "Запис тексту в файл")))

Функція	Дія

	`> (with-input-from-file "test4.txt"`
	`  (lambda () (read-string 20)))`
	`"text in a file"`

`(port-file-identity port)`	повертає номер потоку, пов'язаного з відкритим файлом.
	Наприклад:
	`> (define f (open-output-file "new.txt"))`
	`> (port-file-identity f)`
	3647412659

`(eof-object? obj)`	повертає `#t`, якщо об'єкт – кінець файлу, інакше – `#f`

`(error msg v ...)`	створює повідомлення про помилку шляхом об'єднання рядка `msg` зі значеннями змінних `vs`.
	Наприклад:
	`> (define x 345)`
	`> (error "Код помилки" x)`
	`Код помилки 345`

Примітки: в [] дужках вказуються необов'язкові аргументи.

1.5. Запитання і завдання до пп. 1.1–1.4

1. Знайдіть значення виразів:
   - ((lambda (a b)
     (/ (+ (* 3 a)
     (* 4 b))
     (- a b)))
     2 3);
   - (let ((a 5)
     (b 6))
     (* (+ 5 a)
     (- 4 b))).
2. Знайдіть правильні вирази:
   - (define (функція 3))
   - (define (функція x) (/ (+ x 7.2)
   - (define (@кут a)
3. За допомогою форми lambda визначте let та let*.
4. Визначте функції:
   - $z(x) = 3x + 6$;
   - $f(x,y) = 2x^3 - 3y^3$;
   - $y(x) = \frac{(7+2x)^2}{3+4x}$;
   - $k(a,b,c) = ax^2 + bx + c$.

У разі необхідності визначте допоміжні функції.

5. Визначте функцію $f(x,y)=x(3-xy)+y(5+x)^3+(3-xy)(5+x)$.

За необхідності, скористайтесь локальними змінними:

6. Три опори R_1, R_2, R_3 з'єднані паралельно. Знайдіть опір з'єднання. Результат запишіть у файл «результат.txt».

1.6. Прості типи даних

Scheme є безтиповою мовою програмування у тому сенсі, що імена змінних, списків, функцій та інших об'єктів попередньо не закріплені за певними типами даних. Типи в цілому не пов'язані з іменами об'єктів даних, а супроводжують самі об'єкти. Таким чином, змінні можуть в різні моменти часу представляти різні об'єкти. В цьому розумінні Scheme – безтипова (typeless) мова. В Scheme визначення типу відбувається в процесі перебігу програми.

Динамічна перевірка типів та пізнє зв'язування (late binding) допускають різностороннє використання символів та гнучку модифікацію програми. Функції в Scheme можна визначати практично незалежно від типів даних, до яких вони відносяться.

Однак, все вищесказане не означає, що в Scheme зовсім немає різних типів даних.

Існують прості і похідні типи даних. Похідні типи утворюються з простих за певними правилами.

Прості типи даних:

1.6.1. Числовий тип (number)

В Scheme існує ієрархія числових типів даних (рис. 1):

– числа – number;

– комплексні числа – complex;

– дійсні числа – real;

– раціональні числа – rational;

– цілі числа – integer.

Так, наприклад, 5 є цілим числом, але воно також є й раціональним, дійсним та комплексним числом.

Число можна вказати точно і неточно. Число точне, якщо це ціле число, або отримане від цілого числа шляхом застосування тільки точних операцій. Інакше – число неточне. Виключення складають лише результати функції `inexact->exact`. Приклад застосування:

```
> (inexact->exact pi)
3 39854788871587/281474976710656
> (inexact->exact 1.0)
1
```

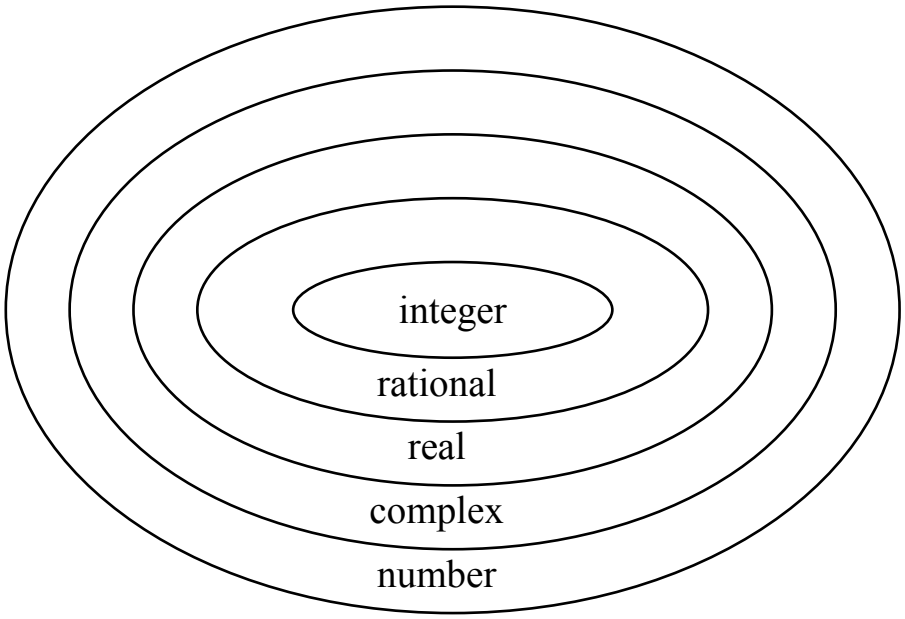


Рис. 1. Ієрархія числових типів

Число можу бути подане в двійковій, вісімковій, десятковій та шістнадцятковій системах числення. Для чисел, поданих в двійковій системі числення, необхідно додати префікс `#b`, у вісімковій – `#o`, в десятковій – `#d`, та – `#x` в шістнадцятковій.

Основа системи числення	**Префікс**	**Приклад**
2	`#b`	`> (+ #b1011 #b101)` **`16`**
8	`#o`	`> (+ #o72 #o54)` **`102`**
10	`#d`	`> (+ #d12 #d39)` **`51`**
16	`#x`	`> (+ #x72 #xE12)` **`3716`**

Для чисел, поданих в десятковій системі числення, префікс `#d` вказувати необов'язково.

В Scheme існують такі логічні функції для чисел (*логічна функція* – функція, результатом якої є «істина» (`#t`) або «хиба» (`#f`)):

Логічна функція	**Дія**	**Результат**: **`#t`**	**Результат: `#f`**
`(number? x)`	аргумент число?	`(number? 5)`	`(number? 'c)`
`(complex? x)`	аргумент ком-	`(complex?`	`(complex? 'b)`

Логічна функція	Дія	Результат: `#t`	Результат: `#f`
	плексне число?	5+3i)	
(real? x)	аргумент дійсне число?	(real? 5)	(rational? 6+4i)
(rational? x)	аргумент раціональне число?	(rational? 7/5)	(rational? 6+4i)
(integer? x)	аргумент ціле число?	(integer? 5)	(integer? 4.6)
(exact? x)	аргумент точне число?	(exact? (* 5 5))	(exact? 3.6)
(inexact? x)	аргумент неточне число?	(inexact? 5.1)	(inexact? 3)
(zero? x)	аргумент нуль?	(zero? 0)	(zero? 4)
(positive? x)	аргумент додатне число?	(positive? 4)	(positive? -9)
(negative? x)	аргумент від'ємне число?	(negative? -7)	(negative? 2)
(even? x)	аргумент парне число?	(even? 0)	(even? 3)
(odd? x)	аргумент непарне число?	(odd? 7)	(odd? 10)

Арифметичні операції можна виконувати з операторами +, –, *, /:

Вираз	Результат	Приклад
(+ x1 x2 … xn)	x1 + x2 + … + xn	> (+ 1 2 3 4 5) **15** > (+) **0**
(– x1 x2 … xn)	x1 – x2 – … – xn	> (– 1 2 3 4 5) **-13** > (– 3) **-3**
(* x1 x2 … xn)	x1 * x2 * … * xn	> (* 1 2 3 4 5) **120** > (*) **1**
(/ x1 x2 … xn)	x1 / x2 / … / xn	> (/ 1 2 3 4 5)

Вираз	Результат	Приклад
		1/120 > (/ 3) **1/3**

В Scheme також є різні математичні функції:

Функція	Математичний запис, коментар	Приклад
(abs x)	$\|x\|$	> (abs -3) **3**
(exp x)	e^x	> (exp 1) **2.828182845904571**
(expt x y)	x^y	> (expt 3 2) **9**
(log x)	ln x	> (log 3) **1.0986122886681098**
(sqr x)	x^2	> (sqr 14) **196**
(sqrt x)	$\sqrt{x}$	> (sqrt 14) **3.7416573867739413**
(cos x)	cos x	> (cos pi) **-1.0**
(sin x)	sin x	> (sin (/ pi 2)) **1.0**
(tan x)	tg x	> (tan 0) **0**
(acos x)	arccos x	> (acos 0) **1.5707963267948966**
(asin x)	arcsin x	> (asin 0) **0**
(atan x)	arctg x	> (atan 0) **0**
(denominator x)	знаменник x	> (denominator 5/6) **6**
(numerator x)	чисельник x	(numerator 11/6) **11**
(remainder x y)	остача від ділення x на y	> (remainder 11 6) **5**
(quotient x y)	ціла частина від ділення x на y	> (quotient 11 4) **2**
(gcd x1 x2 … xn)	НСД(x1, x2, …, xn)	> (gcd 276 84 76)

Функція	Математичний запис, коментар	Приклад
		4
(lcm x1 x2 … xn)	НСК(x1, x2, …, xn)	> (lcm 43 21 7) **903**
(max x1 x2 … xn)	найбільше число з набору x1, x2, …, xn	> (max 5 6 7 12 8) **12**
(min x1 x2 … xn)	найменше число з набору x1, x2, …, xn	> (min 5 6 7 12 8) **5**
(add1 x)	x + 1	> (add1 4) **5**
(sub1 x)	x – 1	> (sub1 4) **3**
(ceiling x)	найближче ціле число, більше або рівне даному; округлює x з надлишком	> (ceiling 3.3) **4.0** > (ceiling 3.7) **4.0** > (ceiling 4.0) **4.0**
(round x)	округлене x	> (round 3.3) **3.0** > (round 3.7) **4.0**
(truncate x)	відкидає дробову частину x	> (truncate 3.3) **3.0** > (truncate 3.7) **3.0** > (truncate 4.0) **4.0**
(random x)	випадкове ціле число з проміжку [0; x – 1]	> (random 6) **3**

1.6.2. Логічний тип (boolean)

Об’єкти логічного типу можуть мати значення істина (#t) або хиба (#f). Для перевірки, чи є об’єкт логічного типу існує логічна функція boolean?:

```
> (boolean? #t)                    > (boolean? 4)
#t                                 #f
```

Слід зазначити, що в Scheme всі стандартні значення, окрім #f, вважаються істинними в умовних виразах:

```
> (define a 5)
> (if a
        "не нуль"
        "нуль")
"не нуль"
```

Логічні функції рівності:

Логічна функція	Математичний запис, коментар	Приклад
(eq? v1 v2)	#t, якщо значення v1 і v2 рівні і якщо інше не передбачено для них eqv? Тип v1 і v2: числа, знаки, символи	> (eq? 'yes 'yes) **#t** > (eq? 'yes 'no) **#f** > (eq? (make-string 3 #\z) (make-string 3 #\z)) **#f**
(eqv? v1 v2)	#t, якщо значення v1 і v2 рівні і якщо інше не передбачено для них eqv? Тип v1 і v2: числа, знаки, символи, рядки, вектори	> (eqv? 'yes 'yes) **#t** > (eqv? 'yes 'no) **#f** > (eqv? (expt 2 100) (expt 2 100)) **#t** > (eqv? 2 2.0) **#f** > (eqv? (integer->char 955) (integer->char 955)) **#t** > (eqv? (make-string 3 #\z) (make-string 3 #\z)) **#f**
(equal? v1 v2)	#t, якщо v1 і v2 посилаються на один і той самий об'єкт	> (equal? 'yes 'yes) **#t** > (equal? 'yes 'no) **#f** > (equal? (expt 2

Логічна функція	Математичний запис, коментар	Приклад
		100) (expt 2 100)) **#t** > (equal? 2 2.0) **#f** > (equal? (make-string 3 #\z) (make-string 3 #\z)) **#t**

1.6.3. Тип знак (character)

Об'єкти типу знак – букви, цифри та інші знаки клавіатури комп'ютера, а також деякі керуючі знаки, такі як перехід на новий рядок та табуляція. Знаки позначаються префіксом «#\».

Більшість функцій для роботи зі знаками – логічні:

Логічна функція	Дія	#t	#f
(char? x)	аргумент знак?	(char? #\b) (char? #\space)	(char? 'b) (char? "a")
(char=? ch1 ch2)	знаки ch1 і ch2 однакові?	(char=? #\s #\s)	(char=? #\s #\S)
(char<? ch1 ch2)	знак ch1 менший за ch2?	(char<? #\4 #\s)	(char<? #\s #\S)
(char>? ch1 ch2)	знак ch1 більший за ch2?	(char>? #\i #\4)	(char>? #\б #\i)
(char<=? ch1 ch2)	знак ch1 не більший за ch2?	(char<? #\4 #\a)	(char<? #\a #\4)
(char>=? ch1 ch2)	знак ch1 не менший за ch2?	(char>=? #\б #\б)	(char>=? #\4 #\б)
(char-alphabetic? x)	знак буква?	(char-alphabetic? #\a)	(char-alphabetic? #\4)
(char-numeric? x)	знак цифра?	(char-numeric? #\4)	(char-numeric? #\a)
(char-upper-case? x)	знак велика буква?	(char-upper-case? #\A)	(char-upper-case? #\a)

Логічна функція	Дія	#t	#f
(char-lower-case? x)	знак маленька буква?	(char-lower-case? #\a)	(char-lower-case? #\A)

Порядок розміщення знаків в таблиці ASCII (англ. American Standard Code for Information Interchange — американський стандартний код для обміну інформацією) наступний: спершу – цифри, далі – великі, а потім малі букви англійського алфавіту. Далі слідують букви національного алфавіту (українського).

Функції для роботи зі знаками:

Функція	Дія	Приклад
(char->integer ch)	код знаку ch (у відповідності з таблицею кодів ASCII)	> (char->integer #\3) **51**
(integer->char n)	знак, код якого в таблиці ASCI, дорівнює n	> (char->integer #\a) **97**
(char-upcase ch)	знак ch верхнього регістру	> (char-upcase #\a) **#\A**
(char-downcase ch)	знак ch нижнього регістру	> (char-downcase #\B) **#\b**

1.6.4. Символьний тип (symbol)

Корисність об’єктів типу символ полягає в тому, що два символи рівні тоді і тільки тоді, якщо їх імена написані однаково.

Символи позначаються префіксом «'».

Функції для роботи з символами:

Функція	Дія	Приклад
(symbol? s)	аргумент символ?	> (symbol? 'символ) **#t** > (symbol? "boo") **#f**
(symbol->string s)	перетворення символу s в рядок	> (symbol->string 'hello) **"hello"**
(string->symbol str)	перетворення рядка str в символ	> (string->symbol "hello")

Функція	Дія	Приклад
		`hello`

Примітки: детальніше про тип `string` дивись у п. 2.1.

1.7. Запитання і завдання до пп. 1.1–1.6

1. Дано числа x, y, z. Знайдіть a і b, якщо

а) $a = \dfrac{\sqrt{|x-1|} - \sqrt[3]{y}}{1 + \dfrac{x^2}{2} + \dfrac{y^2}{4}}$, $b = x(arctgz + e^{-(x+3)})$;

б) $a = \dfrac{3 + e^{y-1}}{1 + x^2 \,|\, y - tgz \,|}$; $b = 1 + |\, y - x \,| + \dfrac{(y-x)^2}{2} + \dfrac{(y-x)^3}{3\sin x}$.

2. Обчисліть значення функції $U=f(x,y,z)$ при заданих значеннях x, y, z:

$$U = \frac{\dfrac{3 + e^{y-1}}{1 + x^2 \,|\, y - tgz \,|}}{1 + |\, y - x \,| + \dfrac{(y-x)^2}{2} + \dfrac{|\, y - x \,|^3}{3}}$$

1.8. Індивідуальна робота №1 до пп. 1.1–1.6

1. Визначте функцію $y(x)$ та продемонструйте її роботу. Попередньо знайдіть область визначення функції $y(x)$, для того, щоб при тестуванні роботи функції, не викликати її з x, що не належить до її області визначення.

№	Завдання
1.	$y = \dfrac{\sin(x^2 - 4) + 2}{\cos 3x}$
2.	$y = \sqrt{|\, 1 - x^2 / 4x - 4\ln x \,|}$
3.	$y = \dfrac{|\, x^2 - 4x \sin 6.2x \,|}{\sin 3.4x}$
4.	$y = \dfrac{5\ln(x + 6.2) + \sin 3x^7 - 4}{|\, tg\, 7.5x - 3 \,|}$
5.	$y = tg\, 5.7x + \cos 5.2x - \ln x^2$

№	Завдання
6.	$y = \sqrt{x^4 + 3x + \ln(5x + 4)}$
7.	$y = \lvert \cos x / 2.7 \rvert + \sin(1.2x + 2)$
8.	$y = \dfrac{1 + tg(3x^4 - 4.1 + 5)}{\cos 4.9x^5 + 3.7x}$
9.	$y = \sqrt{\lvert \sin 4.3x^5 - \cos x + 43 \rvert}$
10.	$y = \dfrac{tg\,7.6x + 19.2x^3}{\cos(x^4 - 5.3x)}$
11.	$y = 11.2\cos(2x + 1) + \lvert \sin 1.5 \rvert$
12.	$y = \dfrac{\sqrt{x^5 + 2 - \sqrt{x}}}{3x}$
13.	$y = \sin \lvert x - 9.5 \rvert + \cos^2 4.3x$
14.	$y = \dfrac{\cos(x^2 - 4)}{\ln(5 + \sin(x))}$
15.	$y = \dfrac{\sqrt{6x^2 - 2\ln x}}{\lvert 1 - x\sin x \rvert} + 3$
16.	$y = \ln(\dfrac{1 + \sin(x^2 + 4)}{x^5 - 2})$
17.	$y = \sqrt{\lvert \cos 9.2x^8 - \ln(x + 3) + 4 \rvert}$
18.	$y = \lvert 4.9x^6 + \sin(3.5x + 4) \rvert$
19.	$y = \sqrt{8.4x^3 - \ln(2x + 6.2)}$
20.	$y = \lvert x^7 + 7.1x\sin 3.4x^2 \rvert$
21.	$y = \dfrac{\sqrt{2x^3 - 4.2}}{\lvert \sin 4x^3 \rvert}$
22.	$y = \dfrac{\sin \lvert x^3 + 5tg3x \rvert}{\cos 7x^2}$

№	Завдання
23.	$y = \sqrt{\lvert \sin 3x^5 \rvert + \cos 5.3x}$
24.	$y = \lvert x^4 \ln(4 + \cos 4.1x) \rvert$
25.	$y = \sqrt{1 + \cos(4x^5 - 3x + 4)}$
26.	$y = 9.5tgx^3 + \ln(x - 9.4)$
27.	$y = \frac{tg7.8x + 5.2\cos 3x}{\sqrt{7.6x}}$
28.	$y = \sqrt{\frac{\sin(3.3x + 5)}{\ln x}}$
29.	$y = \frac{7 - \ln(5x^3 + 2.4 - 7)}{\cos 9.7x^7 - +7.1x}$
30.	$y = \sqrt{9 - \sin(7x^6 + 3x^2 - 2)}$

2. Трикутник задано координатами вершин: A (5, 4), B(i, i + 1), C(i + 7, i – 2), де i – номер варіанту. Визначити функцію, яка обчислює:

№	Завдання
1.	висоту, проведену до сторони AB, та периметр трикутника BCD, де D – середина сторони AB
2.	бісектрису кута ABC та медіану сторони AC
3.	площу трикутника ABC та бісектрису кута BCA
4.	медіану сторони AB та висоту, проведену до сторони AC
5.	периметр трикутника ABC та площу трикутника ABD, де D – середина сторони BC
6.	радіус кола, описаного навколо трикутника ABC та висоту, проведену до сторони AB
7.	площу трикутника ABD, де D – середина сторони AC, та медіану сторони BC
8.	радіус кола, вписаного в трикутник ABC та медіану сторони AC
9.	радіус кола, описаного навколо трикутника ABC, та площу трикутника ABC
10.	периметр трикутника BCD, де D – середина сторони AB, та

№	Завдання
	висоту, проведену до сторони AB
11.	радіус кола, вписаного в трикутник BCD, де D – середина сторони AC, та бісектрису кута ABC
12.	периметр трикутника ABC та бісектрису кута BCD, де D – середина сторони AC
13.	радіус кола, вписаного в трикутник ABC, та висоту, проведену до сторони AB
14.	периметр ABD, де D – середина сторони BC, та площу трикутника ABC
15.	медіану сторони BC та бісектрису кута ABC
16.	площу трикутника ABC та радіус кола, описаного навколо трикутника ABC
17.	периметр трикутника ABC та бісектрису кута ABD, де D – середина сторони BC
18.	медіану сторони AB та бісектрису кута ABC
19.	радіус кола, описаного навколо трикутника ABC, та бісектрису кута BCA
20.	бісектрису кута CAD, де D – середина сторони, та площу трикутника CAD
21.	висоту, проведену до сторони AC, та бісектрису кута CAD, де D – середина сторони AB
22.	радіуси вписаного та описаного кіл
23.	бісектрису кута BCD, де D – середина сторони AC, та медіану сторони BC
24.	периметр трикутника ABD, де D – середина сторони BC, та медіану сторони AC
25.	площу трикутника ABD, де D – середина сторони BC, та висоту, проведену до сторони AB
26.	радіус кола, описаного навколо трикутника ABC, та периметр трикутника ABC
27.	бісектрису кута CAB та радіус кола, вписаного в трикутник ABC
28.	периметр трикутника ABC та медіану сторони AC
29.	висоту, проведену до сторони AC, та медіану сторони BC
30.	медіану сторони AB та радіус кола, описаного навколо трикутника ABC

1.9. Керування виконанням програми

При виконанні програми може знадобитись здійснення певних перевірок та в залежності від їх результату виконання тих чи інших дій або ж виконання певних дій необхідну кількість разів чи доти, доки виконуються певні умови. В таких випадках використовують умовні або циклічні вирази.

1.9.1. Умовні вирази

1.9.1.1. Умовний вираз `if`

Розглянемо формулу для знаходження модуля числа:

$$|x| = \begin{cases} x, & x >= 0 \\ -x & x < 0 \end{cases}.$$

Для розв'язання подібних задач можна скористатись примітивним умовним виразом `if` (`if` є примітивним умовним виразом, оскільки, використовуючи його, можна реалізувати всі інші умовні вирази). Загальна форма:

```
(if умова вираз-так вираз-інакше)
```

Якщо значенням умови є «істина», значенням умовного виразу буде значення виразу-так, якщо «хиба» – виразу-інакше.

Функція обчислення модуля числа з використанням умовного виразу `if`:

```
> (define (модуль x)
    (if (>= x 0)
        x
        (- x)))
> (модуль -5)
5
```

Якщо ж у задачі наявні три або більше випадки, необхідно скористатися вкладеними умовними виразами. Так, розглянемо формулу для знаходження signum-функції (функції знаку) числа:

$$signx = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}.$$

```
> (define (знак x)
    (if (> x 0)
        1
        (if (= x 0)
            0
```

```
              -1)))
> (знак 3)
1
```

1.9.1.2. Умовний вираз `cond`

Якщо в задачі наявні три або більше випадки, для її розв'язання доцільніше скористатись похідним умовним виразом – розбором випадків `cond`. Скорочена форма:

```
(cond (умова1 тіло1)
      (умова2 тіло2)
       ...
      (умоваn тілоn))
```

Тіло може мати вигляд `вираз1 вираз2 … виразk`.

Форма складається з ключового слова `cond`, після якого записані взяті в дужки пари виразів `(умова тіло)`, які називаються гілками. Значення умовного виразу `cond` обчислюється так: спочатку перевіряється `умова1`, якщо вона має значення «істина» результат всього `cond`-виразу – значення останнього виразу в `тілі1`, інакше – перевіряється `умова2`. Так продовжується доти, доки не знайдеться умова, значенням якої буде «істина». В цьому випадку інтерпретатор повертає значення останнього виразу у відповідному тілі в якості значення всього виразу `cond`. Якщо жодна з умов не матиме значення «істина», значення умовного виразу не визначене.

З використанням `cond` функція обчислення знаку числа матиме вигляд:

```
> (define (знак x)
    (cond ((> x 0) 1)
          ((= x 0) 0)
          ((< x 0) -1)))
> (модуль -3)
-1
```

Повна форма виразу `cond`:

```
(cond (умова1 тіло1)
      (умова2 тіло2)
       ...
      (умоваn тілоn)
      (else тіло))
```

`else` – спеціальне ключове слово в заключній гілці `cond`. Якщо жодна з умов не матиме значення «істина», в якості результату буде повернуто значення останнього виразу, що стоїть в тілі саме

ції гілки.

Використовуючи повну форму умовного виразу `cond`, функція обчислення знаку числа матиме вигляд:

```
> (define (модуль x)
    (cond ((> x 0) 1)
          ((= x 0) 0)
          (else -1)))
> (модуль -3)
-1
```

1.9.1.3. Умовний вираз `case`

`case`-вираз, як і `cond`, є похідним умовним виразом. Його скорочена форма:

```
(case ключ гілка1 гілка2 ... гілкаn)
```

Повна форма `case`:

```
(case ключ гілка1 гілка2 ... гілкаn (else тіло))
```

Ключ може бути будь-яким виразом, а гілки повинні мати вигляд `(список_значень тіло)`, де тіло може мати вигляд `вираз1 вираз2 … виразk`.

Значення `case`-виразу обчислюється таким чином: ключ порівнюється зі списком_значень гілки1, якщо ключ не дорівнює жодному із значень, ключ порівнюється зі списком_значень гілки2 і т. д., доки не знайдеться рівне ключу значення, в цьому випадку значенням `case`-виразу буде значення відповідного тіла. Якщо ключ не співпав із жодним значенням, результат – значення тіла `else`.

Приклад 1. Визначити функцію, яка отримує номер дня тижня та повертає його назву.

```
> (define (день-тижня x)
    (case x
      ((1) 'понеділок)
      ((2) 'вівторок)
      ((3) 'середа)
      ((4) 'четвер)
      ((5) 'пятниця)
      ((6) 'субота)
      ((7) 'неділя)
      (else 'невідомо)))
> (день-тижня 13)
невідомо
```

Приклад 2. У деякого працівника повні робочі дні – понеділок, вівторок, четвер; середа і п’ятниця – неповні. Субота і неділя – ви-

хідні. Визначити функцію, яка за номером дня тижня, повертає його «робочу характеристику».

```
> (define (робочий-день x)
    (case x
      ((1 2 4) 'повний)
      ((3 5) 'неповний)
      ((6 7) 'вихідний)
      (else 'невідомо)))
> (робочий-день 6)
вихідний
```

1.9.1.4. Умовні вирази `when` і `unless`

Похідні умовні вирази `when` і `unless` зручно використовувати тоді, коли є тільки одна гілка в умовному виразі («коли» або «інакше»). Загальна форма `when`:

```
(when умова тіло)
```

Тіло може мати вигляд `вираз1 вираз2 … виразk`.

Якщо умова має значення «істина», обчислюються всі вирази. Значенням всього `when`-виразу буде значення останнього виразу, обчисленого в його тілі.

Загальна форма умовного виразу `unless`:

```
(unless умова тіло)
```

Тіло може мати вигляд `вираз1 вираз2 … виразk`.

Якщо умова має значення «хиба», обчислюються всі вирази. Значенням всього `unless`-виразу буде значення останнього виразу, обчисленого в його тілі.

1.9.1.5. Вирази логічної композиції

Крім знаків елементарних співвідношень («>», «<», «<=», «<=») та стандартних логічних функцій існують вирази логічної композиції, які дозволяють утворювати складені умови:

- `(and вираз1 вираз2 … виразn)`

Інтерпретатор обчислює значення виразів по одному, зліва направо. Якщо якийсь із виразів має значення «хиба», значення всього `and`-виразу – «хиба», інші вирази навіть не обчислюються. Якщо всі вирази мають значення «істина», значенням виразу `and` є значення останнього з них.

```
> (define a 3)
> (and (< 2 5) (number? a))
#t
> (define a 3)
> (define b 5)
```

```
> (and (= a 3) (< b a) (and (number? a) (number? b)))
#f
```

– (or вираз1 вираз2 … виразn)

Інтерпретатор обчислює значення виразів по одному, зліва направо. Якщо якийсь із виразів має значення «істина», воно повертається як результат виразу `or`, а інші вирази не обчислюються. Якщо значення всіх виразів «хиба», значенням виразу `or` також є «хиба».

```
> (define a 3)
> (or (< 2 5) (number? a))
#t
```

– (not вираз)

Значення виразу `not` – «істина», якщо значення виразу «хиба», і «хиба» – в протилежному випадку.

```
> (define a 5)
> (not (< a 10))
#f
```

1.9.2. Запитання і завдання до п. 1.9.1

1. Коли найдоцільніше використовувати умовний вираз `if`? `cond`? `case`? `when`? `unless`? Наведіть приклади задач та їх розв'язки, використовуючи всі можливі для даного випадку умовні вирази.

В наступних завданнях (2-8) обов'язково перевірте правильність вхідних даних та продемонструйте роботу функцій для всіх можливих випадків.

2. Визначте функції:

– (`max x y`), яка повертає більше з двох чисел;

– (`min x y`), яка повертає менше з двох чисел;

– $z = \begin{cases} x - y, & якщо \quad x > y \\ y - x + 1 & інакше \end{cases}$.

Протестуйте їх роботу з комплексними числами.

3. Визначте функцію, яка за номером місяця повертає відповідну місяцю назву пори року.

4. Використовуючи функцію `remainder`, визначте, чи є задане ціле число парним.

5. Визначте функцію, яка отримує ціле число n (n>99) та обчислює число сотень в ньому.

6. Визначте функцію, яка отримує число n (n≤100) та обчислює:

– кількість цифр в числі n;

– суму цифр числа n;
– останню цифру числа n;
– першу цифру числа n.

7. Визначте функцію (відрахування сума), яка за нарахованою заробітною платнею розраховує суму, яку робітник отримає «на руки». При цьому необхідно врахувати, що:
– відрахування в пенсійний фонд (ПФ) становлять 1%, якщо нарахована сума до 150 грн. і 2% – якщо більше;
– податок, що йде до фонду зайнятості (ФЗ), складає 0,5% від нарахованої суми. При цьому слід врахувати, що до 17 грн. (неоподатковуваний мінімум) податок не нараховується;
– прибутковий податок (ПП) нараховується згідно ставок (див. таблицю):

Місячний сукупний прибуток, що підлягає оподаткуванню	Ставки та розміри прибуткового податку
до 17 грн. (1 неоп. мінімум)	мінімум, що не підлягає оподаткуванню
18-85 грн. (від 1 до 5 неоп. мінімумів)	10% суми з прибутку, що перевищує розмір одного неоп. мінімуму
86-170 грн. (від 5 до 10 неоп. мінімумів)	6 грн. 80 коп. + 15% з суми, що перевищує 85 грн.
171-1020 (від 10 до 60 неоп. мінімумів)	19 грн. 55 коп. + 20% з суми, що перевищує 170 грн.
1021-1700 грн. (від 60 до 100 неоп. мінімумів)	189 грн. 55 коп. + 30% з суми, що перевищує 1020 грн.
1701 грн. та вище (більше 100 неоп. мінімумів)	393 грн. 55 + 40% з суми, що перевищує 1700 грн.

Таким чином, результат функції
`(відрахування сума) = сума – відрахування_ПФ – відрахування_ФЗ – ПП`

8. «Копійка гривню береже»

Визначте функцію `(сума копійки)`, яка отримує кількість копійок (ціле число з діапазону 1..100 000) та визначає ціле число гривень і залишок у копійках. При цьому слова "гривня" та "копійка" треба узгоджувати з числівниками. Наприклад: "1 гривня", але "73 гривні"; "1 копійка", але "25 копійок".

Виводити кількості гривень та копійок потрібно в окремих ря-

дках. Якщо кількість гривень чи копійок дорівнює нулю – відповідний рядок виводити не треба.

Аналіз задачі.

– Перевірте правильність вхідних даних – копійки мають бути таким цілим числом: 0 ≤ копійки ≤ 100 000, якщо ця умова не виконується – виведіть повідомлення про некоректність даних, якщо ж умова виконується – перейдіть до наступного пункту.

– Для отримання числа гривень знайдіть остачу від ділення копійок на 100.

– Знайдіть залежність між закінченням слова «гривня» та числом гривень (наприклад, для 1 ≤ гривня ≤ 31).

– Узагальніть отриманий результат для всіх можливих випадків.

– Який діапазон чисел гривні не описує дане правило?

– Який з умовних виразів є найбільш раціональним для даної задачі? Чому?

– Для створення локальних змінних скористайтесь конструкцією `let` або `let*`. Яка з них в даному випадку є доречнішою? Чому?

1.9.3. Індивідуальна робота №2 до п. 1.9.1

Визначте функції всіма можливими способами. Обґрунтуйте, чому для їх визначення не можна скористатись іншими умовними виразами. Перевірте правильність вхідних даних та продемонструйте роботу функцій хоча б для трьох випадків:

1. Визначте функцію X:

№	Завдання
1.	$X = \begin{cases} ab+1, якщо\ a > b \\ 25, якщо\ a = b \\ (a-5)/b, якщо\ a < b \end{cases}$
2.	$X = \begin{cases} 2a+b+1, якщо\ a > b \\ a-2, якщо\ a = b \\ (a-5)/b, якщо\ a < b \end{cases}$
3.	$X = \begin{cases} ab-3, якщо\ a > b \\ 2a^2, якщо\ a = b \\ (ab+1)/b, якщо\ a < b \end{cases}$

4.	$X = \begin{cases} b+31, якщо\ \ a > b \\ a-25, якщо\ \ a = b \\ (3a-5)/b, якщо\ \ a < b \end{cases}$
5.	$X = \begin{cases} a/b+5, якщо\ \ a > b \\ -5/b, якщо\ \ a = b \\ (a-b^2)/b, якщо\ \ a < b \end{cases}$
6.	$X = \begin{cases} b/a+61, якщо\ \ a > b \\ -5a^3, якщо\ \ a = b \\ (b-a)/b, якщо\ \ a < b \end{cases}$
7.	$X = \begin{cases} a/b+1, якщо\ \ a > b \\ a-1, якщо\ \ a = b \\ (ab-5)/a, якщо\ \ a < b \end{cases}$
8.	$X = \begin{cases} a/b-11a, якщо\ \ a > b \\ b-25, якщо\ \ a = b \\ (a^3-5)/a, якщо\ \ a < b \end{cases}$
9.	$X = \begin{cases} (3a-5), якщо\ \ a > b \\ 5ab-4, якщо\ \ a = b \\ (a^3+b)/a, якщо\ \ a < b \end{cases}$
10.	$X = \begin{cases} ab+21, якщо\ \ a > b \\ (3a/b+1), якщо\ \ a = b \\ 23, якщо\ \ a < b \end{cases}$
11.	$X = \begin{cases} a/b+31, якщо\ \ a > b \\ (b-25)/a, якщо\ \ a = b \\ 5a-1, якщо\ \ a < b \end{cases}$

12.	$X = \begin{cases} 5a + b + 1, якщо\ \ a > b \\ a + b^4 - 125, якщо\ \ a = b \\ (a - 5)/b, якщо\ \ a < b \end{cases}$
13.	$X = \begin{cases} b/a - 1, якщо\ \ a > b \\ 4ab + 2, якщо\ \ a = b \\ (a - 53)/b, якщо\ \ a < b \end{cases}$
14.	$X = \begin{cases} ab - 1, якщо\ \ a > b \\ b/(a + 6), якщо\ \ a = b \\ (a - 5)/b, якщо\ \ a < b \end{cases}$
15.	$X = \begin{cases} 2a/b + 1, якщо\ \ a > b \\ -445, якщо\ \ a = b \\ (b + 5)/a, якщо\ \ a < b \end{cases}$
16.	$X = \begin{cases} a/b + 1, якщо\ \ a > b \\ a + 25, якщо\ \ a = b \\ (ab - 2)/a, якщо\ \ a < b \end{cases}$
17.	$X = \begin{cases} b/a + 1, якщо\ \ a > b \\ a - 5, якщо\ \ a = b \\ (3b - 5a)/b, якщо\ \ a < b \end{cases}$
18.	$X = \begin{cases} a, якщо\ \ a > b \\ b^5 - 10, якщо\ \ a = b \\ (a - 5)/b, якщо\ \ a < b \end{cases}$
19.	$X = \begin{cases} (b + 7)^5, якщо\ \ a > b \\ (6 - a)/b, якщо\ \ a = b \\ a^2 + 3, якщо\ \ a < b \end{cases}$

20.	$X=\begin{cases}2ab/7+61, якщо\ \ a>b\\ 4b-5, якщо\ \ a=b\\ (b-a)/3, якщо\ \ a<b\end{cases}$
21.	$X=\begin{cases}4b+7, якщо\ \ a>b\\ a-5, якщо\ \ a=b\\ (b-a)/b, якщо\ \ a<b\end{cases}$
22.	$X=\begin{cases}5/a+6, якщо\ \ a>b\\ 7b-5, якщо\ \ a=b\\ (2-a)/b, якщо\ \ a<b\end{cases}$
23.	$X=\begin{cases}a/b+7, якщо\ \ a>b\\ a/b-5, якщо\ \ a=b\\ (4-a)/3, если\ \ a<b\end{cases}$
24.	$X=\begin{cases}b/a+3, якщо\ \ a>b\\ b-5, якщо\ \ a=b\\ (9b-a)/b, якщо\ \ a<b\end{cases}$
25.	$X=\begin{cases}ab-3, якщо\ \ a>b\\ a+2/b, якщо\ \ a=b\\ (1-a)^3, якщо\ \ a<b\end{cases}$
26.	$X=\begin{cases}6+b, якщо\ \ a>b\\ a-9, якщо\ \ a=b\\ (b-3)^4, якщо\ \ a<b\end{cases}$
27.	$X=\begin{cases}ab/5-1, якщо\ \ a>b\\ 2a-b, якщо\ \ a=b\\ (5b-a)/b, якщо\ \ a<b\end{cases}$

28.	$X = \begin{cases} 5/b, якщо\ a > b \\ 3a+6, якщо\ a = b \\ (a+2b)/3, якщо\ a < b \end{cases}$
29.	$X = \begin{cases} a+2, якщо\ a > b \\ b-5a, якщо\ a = b \\ b-a/(a+3), якщо\ a < b \end{cases}$
30.	$X = \begin{cases} a/b+1, якщо\ a > b \\ (b-2)^2, якщо\ a = b \\ (a-b)/a, якщо\ a < b \end{cases}$

2. Визначте функцію *y(x)*:

а) $y(x) = \begin{cases} y_{i+3}, & якщо\ x=1\ або\ x=2\ або\ x=3 \\ & інакше\ y_{i+4} \end{cases}$;

б) $y(x) = \begin{cases} y_{i-3}, & якщо\ x=1\ або\ x=2\ або\ x=3 \\ & інакше\ y_{i-4} \end{cases}$.

Примітки:

– якщо ваш варіант *i* <= 20 виконуйте завдання а), інакше – завдання б);

– функцію y_i використайте з завдання 1 індивідуальної роботи № 1.

3. Визначте функцію, яка отримує:

№	Завдання
1.	номер студента зі списку в журналі та повертає його домашню адресу і номер телефону
2.	марку моделі автомобіля та повертає об'єм її двигуна і максимальну швидкість
3.	номер потягу та повертає назву кінцевого пункту призначення і кількість зупинок до нього
4.	номер дня тижня та повертає його назву і кількість пар в цей день
5.	номер студента зі списку в журналі та повертає його прізвище й ім'я

№	Завдання
6.	номер маршрутного таксі та повертає довжину його маршруту і першу зупинку
7.	числовий код товару та повертає довідку про ціну і кількість товару на складі
8.	числовий код групи та повертає повну назву групи і кількість студентів в ній
9.	номер потягу та повертає час його відправлення і час прибуття на кінцеву зупинку
10.	ідентифікаційний код та повертає прізвище й ім’я його власника
11.	номер квартири в будинку та повертає кількість кімнат та мешканців у ній
12.	номер продавця певного магазину та повертає суму його продаж за останній місяць і нараховану заробітну платню
13.	номер студента зі списку в журналі та повертає номери його мобільного і домашнього телефонів
14.	номер мобільного телефону та повертає прізвище й ім’я його власника
15.	номер сторінки в журналі групи та повертає назву предмета, для якого відведена ця сторінка і прізвище викладача, який читає лекції з цього предмету
16.	номер спеціальності та повертає кількість поданих заявок на неї і кількість місць за держбюджетом
17.	номер палати в лікарні та повертає кількість хворих в ній і прізвище лікаря, закріпленого за цією палатою
18.	номер книжкової полиці в бібліотеці та повертає кількість книжок на цій полиці та наукову область, до якої вони відносяться
19.	номер паспорту та повертає прізвище й ім’я його власника
20.	номер автобусу та повертає кількість зупинок і довжину його маршруту
21.	номер телевізійного каналу та повертає його назву і вашу улюблену програму на цьому каналі
22.	номер кабінету в лікарні та повертає прізвище лікаря і кількість прийнятих за останній день хворих
23.	числовий код групи та повертає повну назву групи прізвище її куратора

№	Завдання
24.	номер місяця та повертає назву місяця українською й англійськими мовами
25.	код товару та повертає його ціну і розмір знижки на нього
26.	номер країни та повертає назву цієї країни і назву її столиці
27.	номер рейсу літака та повертає час його відправлення і країну – пункт призначення
28.	код товару та повертає його назву і цінову категорію, до якої він належить
29.	номер міста в Україні та повертає його назву і визначні пам'ятки цього міста
30.	магічне число людини та повертає характеристику людини за цим числом

Примітки: придумати і задати вхідні дані так, щоб вибір був з 4-6 альтернатив.

1.9.4. Циклічні вирази

Цикл – це процес виконання певного набору команд деяку кількість разів.

1.9.4.1. Рекурсивні функції

В Scheme цикли здебільшого реалізують за допомогою *рекурсивних функцій*. Функція називається рекурсивною, якщо вона явно чи неявно містить виклик самої себе.

Розглянемо функцію факторіал, що визначається рівнянням:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1.$

Це рівняння можна переписати в такому вигляді:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 = n \cdot (n - 1)!$

Враховуючи, що 1! = 1, отримуємо:

```
> (define (факторіал n)
    (if (= n 1)
        1
        (* n (факторіал (- n 1)))))
> (факторіал 3)
6
```

Обчислення факторіалу можна записати і інакше: спочатку множимо 1 на 2, далі результат множимо на 3, потім на 4 і так доти, доки не досягнемо n. Для опису цього обчислення необхідно записати правила, за якими змінюються `лічильник` (який набува-

тиме значень від 1 до n) та `результат`:

`результат` = `результат` · `лічильник`

`лічильник` = `лічильник` + 1

Умова закінчення цього процесу: `лічильник > n`.

```
> (define (факторіал n)
    (define (цикл результат лічильник)
        (if (> лічильник n)
             результат
             (цикл (* результат лічильник)
                   (+ лічильник 1))))
    (цикл 1 1))
> (факторіал 4)
24
```

1.9.4.2. Циклічний вираз `do`

Окрім рекурсивних функцій в Scheme є й спеціальна конструкція для організації циклічних обчислень – вираз `do`. Загальна форма:

```
(do ((змінна1 поч_знач1 вираз1)
     (змінна2 поч_знач2 вираз2)
      …)
    (умова результат)
    тіло)
```

Де `поч_знач1` – початкове значення змінної1, а `вираз1` вказує, яким чином змінюється `змінна1` в наступній ітерації; `умова` призначена для перевірки закінчення циклу: якщо вона має значення «істина», значенням циклу є результат, інакше – виконується тіло.

Так, наприклад, розглянемо процес обчислення факторіалу числа n. Для цього нам потрібні змінні:

`лічильник` – початкове значення 1, в циклі збільшується на 1;

`результат` – початкове значення 1, в циклі буде множитись на лічильник.

Якщо `лічильник` стане більшим за число `n` – повертаємо `результат`.

В тілі нічого виконувати не потрібно.

Запишемо проведені міркування у вигляді функції:

```
> (define (факторіал n)
    (do ((лічильник 1 (+ лічильник 1))
         (результат 1 (* результат лічильник)))
       ((> лічильник n) результат)))
> (факторіал 5)
```

```
120
```

У випадку необхідності виведення на термінал значень факторіалів всіх чисел, менших n, модифікуємо функцію факторіал наступним чином:

```
> (define (факторіал n)
     (do ((лічильник 1 (+ лічильник 1))
             (результат 1 (* результат лічильник)))
       ((> лічильник n) результат)
       (print результат)
       (newline)))
> (факторіал 5)
1
1
2
6
24
120
```

1.9.4.3. Вираз «іменований `let`»

Синтаксис:

```
(let ім'я зв'язування тіло)
```

При цьому зв'язування має вигляд:

```
((змінна1 значення1) (змінна2 значення2) … (зміннаn значенняn))
```

Фактично вираз «іменований `let`» є заміною визначення функцій всередині інших функцій за допомогою `define`. Таким чином, аналогічно створенню локальних змінних за допомогою `let`, можна створювати і локальні функції за допомогою `let`.

Обчислення факторіалу з використанням іменованої `let`:

```
> (define (факторіал n)
    (let цикл ((результат 1)
               (лічильник 1))
            (cond ((> лічильник n) результат)
                  (else (цикл (* результат лічильник)
                              (+ лічильник 1))))))
> (факторіал 5)
120
```

1.9.5. Запитання і завдання до п. 1.9.4

1. Дано натуральне число n. Визначте та продемонструйте роботу функції, яка обчислює:

– 2^n;
– $n!$;
– $(1+\frac{1}{1^2})(1+\frac{1}{2^2})...(1+\frac{1}{n^2})$.

2. Визначте та продемонструйте роботу функції, яка обчислює добуток (1+sin 0.1)(1+sin 0.2)…(1+sin 10).

3. Дано натуральне число n. Визначте та продемонструйте роботу функції, яка знаходить:
– кількість цифр в числі n;
– суму цифр числа n;
– останню цифру числа n;
– першу цифру числа n.

4. Дано натуральні числа m, n. Визначте та продемонструйте роботу функції, яка обчислює суму m останніх цифр числа n.

5. Дано дійсне число x. Визначте та продемонструйте роботу функції, яка обчислює

$$x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\frac{x^9}{9!}-\frac{x^{11}}{11!}+\frac{x^{13}}{13!}.$$

6. Алгоритм Евкліда для знаходження найбільшого спільного дільника (НСД) невід'ємних цілих чисел оснований на таких властивостях цієї величини: нехай m і n – нерівні нулю цілі невід'ємні числа, і нехай m≥n. Тоді, якщо n=0, то НСД (m, n)=m, а якщо n ≠0, то для чисел m, n і r, де r – остача від ділення m на n, виконується рівність: НСД (m, n)= НСД (n, r). Наприклад, НСД (15, 6)= НСД (6, 3)= НСД (3, 0)=3.

Дано натуральні числа m, n. Використовуючи алгоритм Евкліда, визначте та продемонструйте роботу функції, яка обчислює НСД(m, n).

7. Кредитний відділ деякого банку хоче обрати систему нумерації для нової кредитної картки таким чином, щоб можна було легко відрізнити справжній номер картки від підробленого. Дизайнерський відділ запропонував наступну перевірку: кредитна картка дійсна, якщо її номер без остачі ділиться на 13, але не ділиться на 2, 3, 5, 7 та 11. Наприклад, номер 1000051 є дійсним, а 1000571 – ні (оскільки 1000571 = 11 * 90961).

Визначте та протестуйте функцію `номер-дійсний?`, яка перевіряє, чи є задане ціле число допустимим для номеру кредитної картки відповідно до цієї умови. Підказка: скористайтесь функцією `remainder`.

Результат:

```
> (номер-дійсний? 1000051)
#t
> (номер-дійсний? 1000751)
#f
```

Маркетингове відділення цього ж банку занепокоєне: чи вистачить номерів для карток, адже, за їх розрахунками, близько 180000 клієнтів захочуть отримати цю картку, а номер картки обмежений семи цифрами. Тобто, кожен номер повинен знаходитись в діапазоні від 1000000 до 9999999. Питання: чи є в діапазоні від 1000000 до 9999999 не менше 180000 чисел, які відповідають умові для обрання їх номером кредитної картки?

Створіть та протестуйте функцію `(число-дійсних-карток початок кінець)`, яка знаходить кількість дійсних номерів для кредитних карт у вказаному діапазоні (початок кінець).

Після перевірки функції `число-дійсних-карток` для невеликих діапазонів, з'ясуйте, чи вистачить-таки номерів для кредитних карт?

8. Дано додатні дійсні числа a, x, e. Визначте та продемонструйте роботу функції, яка в послідовності $y_1, y_2,\ldots$, утвореній за законом

$$y_0 = a,\ y_i = \frac{1}{2}(y_{i-1} + \frac{x}{y_{i-1}}),\ i = 1, 2,\ldots,$$

знаходить перший член y_n, для якого виконується нерівність

$$| y_n^2 - y_{n-1}^2 | < e\ .$$

9. Визначте та продемонструйте роботу функції, яка обчислює:

- $\sum_{j=1}^{100} \frac{1}{j^2}$;
- $\sum_{i=1}^{10} \frac{1}{i!}$;
- $\prod_{k=2}^{20} \frac{k+1}{k+2}$.

10. Визначте та продемонструйте роботу функції, яка табулює функцію $y=f(x)$ для аргументу, що змінюється на інтервалі x є[a,b] з заданим кроком h.

$$y = \begin{cases} \sqrt{\frac{1+x}{1}}, & якщо \quad x > 2 \\ \ln(2x^2 + 5) + 3\sin^2 x, & якщо \quad x =< 2 \end{cases}.$$

Примітки: табулювання функції – це обчислення значень фун-

кції для аргументу, який змінюється від деякого початкового значення до кінцевого значення з певним кроком; складання таблиці значень функції (звідси і назва – табулювання).

11. Багато практичних задач зводяться до розв'язування рівнянь виду f(x) = 0, де функція f(x) визначена і неперервна на деякому інтервалі. Деякі з таких рівнянь (які містять тригонометричні, логарифмічні, показникові і т.д. функції) не можливо розв'язати аналітично. Для таких випадків розроблені чисельні методи.

З використанням чисельних методів, розв'язування рівняння f(x) = 0 складається з двох етапів:

– *відокремлення коренів*, тобто відшукання достатньо малих областей, у кожній з яких є один і тільки один корінь рівняння;

–*уточнення коренів* з заданою точністю.

Одним із методів уточнення коренів на відрізку [x_поч; x_кін] є метод половинного ділення (або дихотомії):

– відрізок [x_поч; x_кін] ділиться точкою x_серед дві рівні частини;

– з двох отриманих відрізків [x_поч; x_серед] і [x_серед; x_кін] вибирається той, на кінцях якого функція f(x) має протилежні знаки;

– отриманий відрізок знову ділиться навпіл і проводяться ті ж міркування;

– процес продовжується доти, доки довжина відрізка, на кінцях якого функція має протилежні знаки, не буде менше заданої точності, тобто, будь-яку точку відрізку можна прийняти за корінь рівняння f(x) = 0.

Визначте та продемонструйте роботу функції, яка методом половинного ділення знаходить розв'язок рівняння $y = x^2 - 3\sin x + 1$ на відрізку [1, 2] з точністю 0,001.

1.9.6. Індивідуальна робота № 3 до п. 1.9.4

1. Обчисліть *S* при заданому *n*.

№	Завдання
1.	$S = \frac{2}{5} + \frac{5}{11} + \ldots + \frac{n^2+1}{3^n+2};$
2.	$S = 1 + \frac{1+2}{1+2^2} + \ldots + \frac{1+n}{1+n^2};$

№	Завдання
3.	$S = \frac{1}{2*5} + \frac{1}{3*6} + ... + \frac{1}{(n+1)(n+4)};$
4.	$S = \frac{2}{1*3} + \frac{3}{2*4} + ... + \frac{n+1}{n(n+2)};$
5.	$S = \frac{1}{1*2} + \frac{1}{3*2^3} + ... + \frac{1}{(2n-1)2^{2n-1}};$
6.	$S = \frac{1}{2!} + \frac{2}{3!} + ... + \frac{n}{(n+1)!};$
7.	$S = \frac{2}{2*1} + \frac{3}{4*2} + ... + \frac{n+1}{2^n * n!};$
8.	$S = \frac{1}{2} + \frac{8}{5} + ... + \frac{n^3}{(3n-1)};$
9.	$S = \frac{5}{2} + \frac{7}{5} + ... + \frac{2n+3}{n^2+1};$
10.	$S = \frac{2}{3} + \frac{3}{8} + ... + \frac{n+1}{(n+2)n};$
11.	$S = \frac{1}{3} + \frac{3}{3^2} + ... + \frac{2n-1}{3^n};$
12.	$S = \frac{2}{1} + \frac{4}{16} + ... + \frac{2^n}{n^4};$
13.	$S = \frac{3}{2} + \frac{9}{8} + ... + \frac{3^n}{n*2^n};$
14.	$S = \frac{3}{2} + \frac{9}{8} + ... + \frac{3^n}{n*2^n};$
15.	$S = \frac{6}{2!} + \frac{8}{3!} + ... + \frac{2n+4}{(n+1)!};$
16.	$S = \frac{3}{1*2} + \frac{6}{3*5} + ... + \frac{3n}{(2n-1)(n+1)};$

№	Завдання
17.	$S = \frac{1*2}{2} + \frac{2*3}{4} + ... + \frac{n*(n+1)}{2^n};$
18.	$S = \frac{3}{2!} + \frac{9}{3!} + ... + \frac{3^n}{(n+1)!};$
19.	$S = \frac{1}{1*3} + \frac{1}{3*5} + ... + \frac{1}{(2n-1)(2n+1)};$
20.	$S = \frac{1}{1*4} + \frac{2}{4*7} + ... + \frac{n}{(3n-2)(3n+1)};$
21.	$S = \frac{2}{1*4} + \frac{3}{2*5} + ... + \frac{n+1}{n(n+3)};$
22.	$S = \frac{3}{1*7} + \frac{4}{3*9} + ... + \frac{n+2}{(2n-1)(2n+5)};$
23.	$S = \frac{1}{1*2*3} + \frac{1}{2*3*4} + ... + \frac{1}{n(n+1)(n+2)};$
24.	$S = \frac{5}{6} + \frac{13}{36} + ... + \frac{3^n + 2^n}{6^n};$
25.	$S = \frac{3}{4} + \frac{5}{36} + ... + \frac{2n+1}{n^2(n+1)^2};$
26.	$S = \frac{1}{9} + \frac{2}{225} + ... + \frac{n}{(2n-1)^2(2n+1)^2};$
27.	$S = \frac{1}{3!} + \frac{1}{5!} + ... + \frac{1}{(2n+1)!};$
28.	$S = \frac{1}{2} + \frac{2}{2^2} + ... + \frac{n}{2^n};$
29.	$S = \frac{2}{1} + \frac{2*5}{1*5} + ... + \frac{2*5*...*(3n-1)}{1*5*...*(4n-3)};$
30.	$S = \frac{1}{1*4} + \frac{4}{4*7} + ... + \frac{n^2}{(3n-2)(3n+1)}.$

2. Визначте та продемонструйте роботу функції, яка методом половинного ділення знаходить розв'язок рівняння на вказаному проміжку з певною точністю.

№	Функція	Проміжок
1.	$y = x - \sin(x + 1) - 0{,}1$	(-5; 2,7)
2.	$y = 0{,}83\cos x + x - 1$	(-4; 4)
3.	$y = 0{,}5x + 4\sin x - 3$	(-10,5; 15)
4.	$y = \cos x + 4x - 3$	(-3; 4)
5.	$y = 3x - \cos x + 1$	(-6; 3)
6.	$y = x^2 - 4\sin x$	(-5; 1)
7.	$y = x^2 - 20\sin x + 2$	(-1,6; 1,4)
8.	$y = x^3 - \sin x - 2$	(-2; 3)
9.	$y = x^3 - 3x^2 + \sin x$	(-2; 5)
10.	$y = x^2 + 15\sin x - 0{,}5$	(-4,8; -1,4)
11.	$y = 2 - x - \cos(x + 4)$	(-5; 5)
12.	$y = 3x - 1{,}5\cos(x + 4) - 6$	(-7; 5)
13.	$y = x^3 + 2{,}23\sin 5x - 16$	(-5; 6)
14.	$y = x^3 - 0{,}1x^2 + 0{,}4x - 2{,}3$	(-4; 5)
15.	$y = x^3 - 4{,}1\sin(x + 0{,}4) - x$	(1; 5)
16.	$y = 0{,}5^x + 1 - (x - 1)^2$	(-4; 1,7)
17.	$y = 5^x + 4x - 4$	(-3; 1,3)
18.	$y = \cos(x + 2) - 3x + 2$	(-3,3; 3,5)
19.	$y = x(2 - \sin x) - 1$	(-2; 3)
20.	$y = 0{,}2x^4 - x^3 - 3x^2 + 5x - 1$	(-4,5; 0)
21.	$y = x^2\cos 2x + 1{,}2$	(-13,2; -10,5)
22.	$y = x^4 + 4x^3 - 7x^2 - 21$	(-15; -3)
23.	$y = \cos(x + 0{,}7) - x^3$	(-8; 3,8)
24.	$y = 0{,}2x^3 - 1{,}3x^2 + 7x - x$	(-10; 4)
25.	$y = 1{,}3x - 3\sin x - 5$	(-14,5; 4,5)
26.	$y = 3^x + 4x + 4{,}5$	(-11; 3)
27.	$y = \sin(x - 0{,}5) - 3x + 4$	(-1; 6)
28.	$y = x^2 - 15\cos(x - 1)$	(-2,2; 0,9)
29.	$y = (x - 1)\sin x - 2$	(-14,2; -11)
30.	$y = 2e^x + 3x + 2$.	(-15; 1)

2. Похідні типи даних

2.1. Рядок (string)

Рядок – послідовність знаків, взятих в подвійні лапки. Довжина рядка – кількість знаків, які він містить. Перший знак має нульовий індекс. Якщо рядок містить подвійні лапки, то для адекватного розуміння їх інтерпретатором, необхідно помістити перед ними символ «\». Приклад:

"Це рядок і \"Цей рядок \" також є рядком"

Логічні функції для роботи з рядками:

Логічна функція	Дія	Приклад
(string? str)	аргумент рядок?	> (string? "Це рядок") **#t** > (string? 5) **#f**
(string=? str1 str2)	рядки str1 і str2 рівні?	> (string=? "привіт" "Привіт") **#f**
(string<? str1 str2)	рядок str1 менший за str2?	> (string<? "квадрат" "трикутник") **#t**
(string>? str1 str2)	рядок str1 більший за str2?	> (string>? "квадрат" "квадратний") **#f**
(string<=? str1 str2)	рядок str1 не більший за str2?	> (string<=? "квадрат" "Квадрат") **#f**
(string>=? str1 str2)	рядок str1 не менший за str2?	> (string>=? "Рядок" "рядок") **#f**
(string-ci=? str1 str2)	те ж, що й string=?, без врахування регістру	> (string-ci=? "привіт" "Привіт") **#t**
(string-ci<? str1 str2)	те ж, що й string<?, без врахування регістру	> (string-ci<? "квадрат" "трикутник") **#t**
(string-ci>? str1 str2)	те ж, що й string>?, без врахування регіс-	> (string-ci>? "кут" "кутовий") **#f**

Логічна функція	Дія	Приклад
	тру	
(string-ci<=? str1 str2)	те ж, що й string<=?, без врахування регістру	> (string-ci<=? "квадрат" "Квадрат") **#t**
(string-ci>=? str1 str2)	те ж, що й string>=?, без врахування регістру	> (string-ci>=? "Рядок" "рядок") **#t**

Функції для створення та зміни рядків:

Функція	Результат	Приклад
(make-string n ch)	рядок довжиною n, який складається зі знаків ch	> (make-string 10 #\s) **"ssssssssss"**
(make-string n)	рядок довжиною n, зміст якого залежить від реалізації Scheme	> (make-string 10) **"\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000"**
(string ch1 ch2 ...)	рядок, що складається зі знаків ch1, ch2 ...	> (string #\П #\р #\и #\в #\і #\т) **"Привіт"**
(string-length str)	довжина рядка str – кількість знаків у ньому	> (string-length "Я - студент!") **12**
(string-append str1 str2 … strn)	рядок, який складається з усіх рядків stri	> (string-append "Привіт!" " " "Я - студент!") **"Привіт! Я - студент!"**
(string-ref str n)	n-й знак рядка str	> (string-ref "Я - студент!" 0) **#\Я**
(substring str n1 n2)	підрядок з рядка str, починаючи з позиції n1 до позиції n2	> (substring "Я - студент!" 4 11) **"студент"**

Також рядок можна створити, використовуючи подвійні лапки. Ці рядки є рядками-константами і їх не можна змінити функціями, такими як **string-set!**:

`(string-set! str n ch)` – замінює в рядку `str` знак з порядковим номером `n` на новий знак `ch`. Наприклад:

```
> (define str (string #\П #\р #\и #\в #\і #\т))
> str
"Привіт"
> (string-set! str 4 #\е)
> str
"Привет"
> (define str2 "Привіт")
> (string-set! str2 4 #\е)
Помилка!!!
> (define (string_1 s)
  (define (iter s kol i)
    (cond ((= i (string-length s)) kol)
              ((equal? (string-ref s i) #\а)
                 (iter s (+ kol 1) (+ i 1)))
              (else (iter s kol (+ i 1)))))
     (iter s 0 0))
> (string_1 "Індивідуальна робота")
3
```

2.2. Пара (pair)

Пара – похідний тип даних. Об'єкти типу пара створюється за допомогою елементарної функції `cons`. Ця функція приймає два аргументи та повертає об'єкт даних, який містить ці два аргументи в якості частин. Наприклад:

```
> (cons 1 2)
(1 . 2)
```

Надати парі ім'я можна звичним для нас способом:

```
> (define a (cons 1 2))
> a
(1 . 2)
```

Отримати частини пари можна, використовуючи елементарні функції `car` і `cdr`:

```
> (car a)
1
> (cdr a)
2
```

Пара є об'єктом, якому можна дати ім'я і працювати з ним, як

і з простими об'єктами даних. Функцію `cons` можна використовувати і для створення пар, елементами яких також є пари:

```
> (define b (cons 3 4))
> b
(3 . 4)
> (define c (cons a b))
> c
((1 . 2) 3 . 4)
> (car c)
(1 . 2)
> (cdr c)
(3 . 4)
> (car (car c))
1
> (car (cdr c))
3
```

Об'єкти даних, які складаються з пар, називаються даними зі списовою структурою `(list-structured data)`.

2.3. Список (list)

Використовуючи пари, можна побудувати список. Раніше ми визначали список як набір елементів, поміщених в круглі дужки. Наприклад:

```
(1 2 3 4 5)
```

Іншими словами, список – ланцюжок пар, який закінчується порожнім списком. Порожній список позначається () або `null`. Список `(1 2 3 4 5)` ідентичний списку `(1. (2. (3. (4. (5. ()))))).`

Для адекватного розуміння інтерпретатором списку, необхідно перед дужками ставити знак «′», що блокує обчислення виразу, інакше інтерпретатор спробує застосувати перший елемент списку до всіх інших елементів як ім'я функції до її параметрів:

```
> '(1 2 3 4 5)
(1 2 3 4 5)
> (1 2 3 4 5)
Помилка! Застосування функції «1» до аргументів: 2 3 4 5
```

Зверніть увагу, що `(cons 1 (cons 2 '()))` – список, в той час, як `(cons 1 2)` – не список, а пара.

Для отримання складових списку використовують функції `car` – для отримання голови списку, і `cdr` – для отримання хвоста спи-

ску:

```
> (car '(1 2 3 4 5))
1
> (cdr '(1 2 3 4 5))
(2 3 4 5)
```

У випадку необхідності отримання другого, третього та ін. елементів списку, можна утворювати комбінації функцій car і cdr:

```
> (car (cdr '(1 2 3 4 5)))
;за допомогою cdr отримуємо (2 3 4)
;застосувавши car до (2 3 4), отримуємо 2
2
> (car (cdr (cdr '(1 2 3 4 5))))
3
```

В Scheme визначені скорочення для комбінацій функцій car і cdr (максимальна довжина комбінації – 4 функції):

Скорочена форма	Повна форма
(caar список)	(car (car список))
(cadr список)	(car (cdr список))
…	
(cddddr список)	(cdr (cdr (cdr (cdr список))))

Тобто, попередньо записана комбінація функцій `car` і `cdr`:

```
(car (cdr (cdr '(1 2 3 4 5))))
```

має такий скорочений вигляд:

```
> (caddr '(1 2 3 4 5))
3
```

В якості елементів список також може містити списки:

```
(define a '(1 2 (3 4)))
```

В такому випадку розрізняють елементи списку на верхньому рівні (підсписок списку в цьому разі – один елемент) і елементи на нижньому рівні – це всі елементарні об’єкти, з яких складається список.

Елементи списку a на верхньому рівні: 1, 2, (3 4).

Елементи списку a на нижньому рівні: 1, 2, 3, 4.

За замовчуванням, розглядають лише елементи списку на верхньому рівні.

Логічні функції для роботи зі списками та парами:

Логічна функція	**Дія**	**#t**	**#f**
`(null? x)`	аргумент порожній список?	`(null? '())`	`(null? '(a))`
`(pair? x)`	аргумент пара?	`(pair? '(1 2))`	`(pair? 2)`
`(list? x)`	аргумент список?	`(list? '(a))`	`(list? 'a)`

Функції для роботи зі списками та парами:

Функція	**Результат**	**Приклад**
`(list x1 x2 … xn)`	список з елементів `xi`	`> (list 1 2 3 4 5)` **`(1 2 3 4 5)`**
`(length lst)`	довжина списку `lst`	`> (length (list 1 2 3 4 5))` **`5`**
`(list-ref lst n)`	n-й елемент списку `lst`	`> (list-ref '(7 9 3 12) 1)` **`9`**
`(list-tail lst n)`	підсписок списку `lst`, з якого вилучено перші `n` елементів	`> (list-tail '(1 2 3 4 5) 2)` **`(3 4 5)`**
`(reverse lst)`	список з елементів списку `lst` в оберненому порядку	`>(reverse '(1 2 3 4))` **`(4 3 2 1)`**
`(append lst1 lst2)`	об'єднаний список з елементів списків `lst1` та `lst2`	`> (append '(1 2 3) '(4 5 6))` `(1 2 3 4 5 6)`
`(last-pair lst)`	остання пара списку `lst`	`> (last-pair '(1 2 3 4 5))` **`(5)`**

`(member item lst)`	якщо елемента item немає в списку `lst` – `#f`, інакше – хвіст списку `lst`, який починається з елемента `item`	`> (member 'a '(b c d))`
		`#f`
		`> (member '(2 3) '((2 3) 4))`
		`((2 3) 4)`
		`> (member 4 '(b c d))`
		`#f`

Функція	Результат	Приклад
		`> (member 4 '(b 4 d))` **`(4 d)`**
`(string->list str)`	список, який складається з усіх знаків рядка `str`	`> (string->list"Я - студент!")` **`(#\Я #\space #\- #\space #\с #\т #\у #\д #\е #\н #\т #\!)`**
`(list->string lst)`	рядок, який складається з усіх знаків списку `lst`	`> (list->string '(#\Я #\space #\- #\space #\с #\т #\у #\д #\е #\н #\т #\!))` **`"Я - студент!"`**

Операції над списками

Інколи буває необхідно застосувати одну й ту ж саму функцію до кількох аргументів. В цьому випадку в нагоді стане функція `map`:

```
> (map sqr (list 1 2 3 4 5))
(1 4 9 16 25)
> (map (lambda (x) (+ x 2)) '(1 2 3))
(3 4 5)
```

Функції, які є аргументами `map`, можуть мати декілька аргументів. В цьому випадку, функція-аргумент `map` буде застосована до кожного кортежу з елементів набору:

```
> (map cons '(1 2 3) '(10 20 30))
((1 . 10) (2 . 20) (3 . 30))
```

Обрати елементи списку, які задовольняють певній умові, можна за допомогою функції `filter`:

```
> (filter odd? (list 1 2 3 4 5))
(1 3 5)
```

2.4. Індивідуальна робота №4 до пп. 2.2–2.3

Визначте функцію:

№	Функція	Дія або результат	Приклад
1.	`(length lst)`	довжина списку `lst`	`> (length '(1 2 (3 4) 5))` **`4`**

№	Функція	Дія або результат	Приклад
2.	(length-all lst)	довжина списку lst (кількість елементів на нижньому рівні)	> (length-all '(1 2 (3 4) 5)) **5**
3.	(list-ref lst n)	n-й елемент списку lst	> (list-ref '(1 2 3 4) 0) **1**
4.	(list-tail lst n)	підсписок списку lst, з якого вилучено перші n елементів	> (list-tail '(1 2 3 4 5) 2) **(3 4 5)**
5.	(reverse lst)	список з елементів списку lst в оберненому порядку	> (reverse '(1 2 3 4)) **(4 3 2 1)**
6.	(reverse2 lst)	список з елементів списку lst в оберненому порядку (на нижньому рівні)	> (reverse '(1 2 3 4)) **(4 3 2 1)**
7.	(removel lst)	видаляє зі списку lst останній елемент	> (removel '(1 2 (3 4) 5)) **(1 2 (3 4))**
8.	(removel2 lst)	видаляє зі списку lst останній елемент на нижньому рівні	> (removel2 '(1 2 (3 4))) **(1 2 (3))**
9.	(revl lst)	повертає обернений та розбитий на рівні список lst	> (revl '(1 2 3) **(((3) 2) 1)**
10.	(devlev1 lst)	розбиває список lst на рівні	> (devlev1 '(1 2 3) **(1 (2 (3)))**
11.	(devlev2 lst)	розбиває список lst на рівні	> (devlev2 '(1 2 3) **(((1) 2) 3)**
12.	(destlev1 lst)	видаляє рівні в списку lst	> (destlev1 '(1 (2 (3)))) **(1 2 3)**
13.	(destlev2 lst)	видаляє рівні в списку lst	> (destlev2 '(((1) 2) 3)) **(1 2 3)**
14.	(remsec lst)	видаляє зі списку lst	> (remsec '(1 2 (3 4) 5))

№	Функція	Дія або результат	Приклад
		кожен другий елемент	**(1 (3 4))**
15.	(remsec2 lst)	видаляє зі списку lst кожен другий елемент	> (remsec2 '(1 2 (3 4) 5)) **(1 (3) 5)**
16.	(devpair lst)	розбиває список lst на пари	> (devpair '(а б в г)) **((а б) (в г))**
17.	(mix lst1 lst2)	утворює новий список з елементів списків lst1 і lst2	> (mix '(а б в) '(1 2 3)) **((а 1) (б 2) (в 3))**
18.	(append lst1 lst2)	об'єднаний список з елементів списків lst1 та lst2	> (append '(1 2 3) '(4 5 6)) **(1 2 3 4 5 6)**
19.	(last lst)	останній елемент списку lst	> (last '(3 2 7 14 2)) **2**
20.	(member item lst)	якщо елемента item немає в списку lst – #f, інакше – хвіст списку lst, який починається з елемента item	> (member 'a '(b c d)) **#f** > (member '(2 3) '((2 3) 4)) **((2 3) 4)** > (member 4 '(b c d)) **#f** > (member 4 '(b 4 d)) **(4 d)**
21.	(even-elem lst)	список, який складається з усіх парних елементів списку lst, які є числами та мають в розряді одиниць цифри від 1 до 3	> (even-elem '(а б 12 21 в 43 к 66)) **(21 43)**
22.	(replsec lst item)	замінює кожен другий елемент списку lst на елемент item	> (replsec '(а б в г д) '1) **(а 1 в 1 д)**

№	Функція	Дія або результат	Приклад
23.	(remove lst item)	видаляє зі списку lst всі елементи на верхньому рівні, що співпадають з item	> (remove '(a (1 в) 1 д) '1) **(а (1 в) д)**
24.	(remove2 lst item)	видаляє зі списку lst всі елементи на нижньому рівні, що співпадають з item	> (remove2 '(a (1 в) 1 д) '1) **(а (в) д)**
25.	(substitue old new lst)	замінює всі входження елемента old в списку lst на елемент new	> (substitue '111 'ааа '(а 111 б 222 в)) **(а ааа б 222 в)**
26.	(addifnone item lst)	перевіряє, чи міститься елемент item в списку lst; якщо ні, то додає цей елемент в початок списку lst	> (addifnone '1 '(а б в)) **(1 а б в)**
27.	(collect lst)	розташовує однакові елементи списку lst підряд	> (collect '(а б в б д а)) **(а а б б в д)**
28.	(depth lst)	глибина списку lst	> (depth '(1 (2 (3 4)))) **3**
29.	(onelevel lst)	перевіряє, чи є список lst одноріневим списком	> (onelevel '(1 2 3 (4))) **#f**
30.	(filter pred lst)	список, утворений з елементів списку lst, які задовольняють умові pred	> (filter odd? '(1 2 3 4 5)) **(1 3 5)**

2.5. Вектор (array)

Вектор – упорядкований набір значень, кожне з яких має свій індекс (починаючи з нуля). Доступ до елементів списку – послідовний, а до елементів вектора – довільний.

Довжина вектора – кількість його елементів. Два вектори рівні, якщо вони мають однакову довжину та якщо їх відповідні значення рівні. Вектори бувають змінні та сталі.

Функції для роботи з векторами:

Функція	Результат	Приклад
(vector v1 v2 … vn)	змінний вектор з елементів vi	> (vector 'C 'x 'e 'м 'a) **#(C x e м a)** > (vector 'CCC 'x 'e 'м 'a) **#(CCC x e м a)**
(vector? vec)	якщо vec вектор – #t, інакше – #f	> (vector? (vector 'C 'x 'e 'м 'a)) *#t*
(make-vector size [v])	змінний вектор з size елементів v	> (make-vector 5 'C) **#(C C C C C)** > (make-vector 5) **#(0 0 0 0 0)**
(vector-immutable v1 v2 …v2)	сталий вектор з елементів vi	> (vector-immutable 'C 'x 'e 'м 'a) **#(C x e м a)**
(vector-length vec)	довжина вектора vec	> (vector-length (vector 'C 'x 'e 'м 'a)) **5**
(vector-ref vec pos)	елемент вектора vec, що знаходиться на позиції pos	> (vector-ref (vector 'C 'x 'e 'м 'a) 0) **C**
Перетворення векторів		
(vector->list vec)	список, який складається з усіх елементів вектора vec	> (vector->list (vector 'C 'x 'e 'м 'a)) **(C x e м a)**
(list->vector lst)	змінний вектор, який складається з усіх елементів списку lst	> (list->vector '(Підйом - о 6 годині)) **#(Підйом - о 6 годині)**
(vector->immutable-	незмінний вектор,	> (vector-

Функція	Результат	Приклад
vector vec)	який складається з усіх елементів змінного вектора vec	>immutable-vector (vector 'CCC 'x 'e 'м 'a)) **#(CCC x e м a)**
(build-vector n proc)	вектор з n елементів, утворений застосуванням функції proc до послідовності цілих чисел від 0 до n	> (build-vector 7 sqr) **#(0 1 4 9 16 25 36)**
Дії над векторами		
(vector-set! vec pos v)	у векторі vec елемент позиції pos замінює на елемент v	> (define вектор (vector 'C 'x 'e 'м 'a)) > (vector-set! вектор 0 'c) > вектор **#(c x e м a)**
(vector-fill! vec v)	замінює всі елементи вектора vec на елемент v	> (vector-fill! (vector 'C 'x 'e 'м 'a) 'c) **#(c c c c c)**
(vector-copy! dest dest-start src [src-start src-end])	повертає вектор dest, в який, починаючи з позиції dest-start, записані елементи вектора src	> (define v (vector 'A 'p 'p 'l 'e)) > (vector-copy! v 4 #(y)) > (vector-copy! v 0 v 3 4) > v **#(l p p l y)**

Приклади:

```
> ; заповнення вектора розмірності n випадковими числами
від 0 до 9
(define (вектор n)
    (let ((a (make-vector n)))
      (do ((i 0 (+ i 1)))
        ((>= i n) a)
        (vector-set! a i (random 10)))))
> (вектор 9)
#(4 1 3 1 5 3 1 5 2)
```

```
> (define (вектор n)
    (let ((a (make-vector n))
          (sum 0))
      ;заповнення вектора
      (do ((i 0 (+ i 1)))
        ((>= i n))
        (vector-set! a i (random 10)))
      (print a)
      ;обчислення суми
      (do ((i 0 (+ i 1)))
        ((>= i n) sum)
        (when (> (vector-ref a i) 4)
          (set! sum (+ sum (vector-ref a i)))))))
> (вектор 5)
#(5 6 7 3 1)
```

2.6. Індивідуальна робота №5 до п. 2.5

Визначте функцію, яка заповнює вектор А випадковими цілими числами з вказаного діапазону та:

№	Завдання
1.	знаходить найбільший елемент і міняє місцями його з першим елементом, якщо в цьому є необхідність; діапазон – (-10; 40)
2.	переписує у вектор В, елементи, які більші середнього значення вектора А; діапазон – (-15; 45)
3.	знаходить суму двох найбільших за абсолютною величиною елементів цього вектора; діапазон – (10; 55)
4.	будує вектор В з позитивних елементів вектора А і впорядковує елементи вектора В за зростанням; діапазон – (0; 20)
5.	визначає максимальну кількість розташованих по зростанню елементів вектора А, які стоять підряд; діапазон – (-11; 19)
6.	записує в цей же вектор спочатку всі негативні числа і нулі, потім усі позитивні, зберігши їх порядок проходження; діапазон – (5; 35)
7.	з’ясовує, чи є у векторі, хоча б два рівних елементи; діапазон – (-20; 30)
8.	знаходить місця розташування найбільшого і найменшого елементів та міняє їх місцями; діапазон – (-15; 25)
9.	знаходить добуток двох найменших за значенням елементів вектора; діапазон – (-17; 23)

№	Завдання
10.	записує у вектор B елементи вектора A у зворотному порядку; діапазон – (-23; 17)
11.	знаходить суму модулів елементів вектора, які більші -10 і менші 10; діапазон – (-45; 45)
12.	знаходить значення та позицію першого елемента, який кратний 7 і менший 20; діапазон – (-100; 100)
13.	з'ясовує, чи є у векторі A два розташованих поряд елементи, значення яких рівні 5; діапазон – (-7; 7)
14.	знаходить добуток елементів вектора, які більші 20 і мають парні індекси; діапазон – (-20; 20)
15.	знаходить добуток найбільшого та найменшого за модулем елементів вектора; діапазон – (-15; 50)
16.	знаходить суму елементів вектора, кратних 5 і менших 10; діапазон – (-50; 50)
17.	знаходить мінімальний елемент вектора та видаляє його; діапазон – (-30; 10)
18.	знаходить суму трьох останніх елементів вектора й перевіряє, чи є у векторі елемент, більший за цю суму; діапазон – (10; 50)
19.	знаходить кількість додатних та кількість від'ємних елементів у цьому векторі; діапазон – (-30; 30)
20.	знаходить добуток вектора A на задане число; діапазон – (-15; 45)
21.	знаходить добуток від'ємних елементів вектора A, які мають непарний індекс. Визначає кількість таких елементів; діапазон – (-30; 30)
22.	скалярний добуток вектора A на вектор B. Розмірності векторів A і B рівні; діапазон елементів вектора A – (-15; 25); діапазон елементів вектора B – (-25; 15)
23.	знаходить значення та позицію максимального елемента. Якщо таких елементів декілька, то визначає їх кількість; діапазон – (-19; 31)
24.	будує перетин послідовностей A і B. Розмірності векторів A і B рівні; діапазон елементів вектора A – (-10; 25); діапазон елементів вектора B – (-35; 40)
25.	будує об'єднання послідовностей A і B. Розмірності векторів A і B рівні; діапазон елементів вектора A – (-27; 27); діапазон елементів вектора B – (-35; 10)

№	Завдання
26.	знаходить суму елементів вектора A, які є простими числами. Визначає кількість таких елементів; діапазон – (-5; 100)
27.	з’ясовує, чи вірно, що всі елементи вектора A входять у вектор B? Розмірності векторів A і B різні; діапазон елементів вектора A – (-10; 10); діапазон елементів вектора B – (-25; 25)
28.	будує вектор B з елементів вектора A, які входять у нього лише 1 раз; діапазон – (-40; 15)
29.	суму векторів A і B. Розмірності векторів A і B рівні; діапазон елементів вектора A – (-15; 10); діапазон елементів вектора B – (-18; 26)
30.	знаходить добуток елементів вектора, які більші 10 і менші 30; діапазон – (-10; 60)

3. Практична Scheme

Проект 1. Психотерапевт

Психотерапію часто визначають як діяльність, спрямовану на позбавлення людини від емоційних, особистісних, соціальних та ін. проблем. Як правило, психотерапія проводиться спеціалістом-психотерапевтом шляхом встановлення глибокого особистого контакту з пацієнтом (шляхом бесід та обговорень)

Однією з основних форм психотерапії є індивідуальна психотерапія. В якості головного інструменту лікувального впливу індивідуальної психотерапії виступає лікар-психотерапевт та стосунки між лікарем та пацієнтом.

Найпершим і одним із найвагоміших моментів у індивідуальній психотерапії є момент встановлення «психотерапевтичного контакту» – довірливих стосунків лікаря з пацієнтом, послаблення напруги, пов'язаної з можливістю для пацієнта обговорювати проблему з людиною, яка може йому допомогти, встановлення скарг пацієнта, характеру його проблем та їх рівня, навчання за рахунок відомостей, отриманих від психотерапевта; визначається, що привело пацієнта до лікаря, що пацієнт хоче змінити в результаті проходження курсу. На думку багатьох спеціалістів, успішне проходження цього етапу терапії забезпечує 80% успіху всієї подальшої індивідуальної психотерапії.

Призначення цієї лабораторної роботи – розробка надзвичайно спрощеного варіанту відомої програми «Еліза», розробленої в Масачусетському технологічному інституті на початку 1960-х Джозефом Вейзенбаумом. «Еліза» – віртуальний співбесідник, імітує діалог з психотерапевтом, реалізуючи техніку активного слухання. Програма названа в честь Елізи Дуліттл, героїні п'єси «Пігмаліон» Бернарда Шоу, яку навчали мові «вищого класу людей».

Приклад роботи програми «Еліза» (слова «Елізи» – курсив, слова пацієнта – звичайний шрифт):

Я психотерапевт. Опишіть, будь-ласка, Ваші проблеми.

У мене страшенно болить голова.

Можливо, Вам варто проконсультуватись з іншим лікарем, я психотерапевт.

Добре, я так і зроблю.

Ви можете детальніше про це розповісти?

Що Ви говорите? Ні, я не можу вдаватись в подробиці. Побачимося пізніше.
Чому Ви так говорите?
Тепер Ви говорите дурниці!
Що змушує Вас та говорити?
Це не має ніякого сенсу
…

Приклад роботи розробленої в цьому проекті програми `Лікар`:

```
> (Лікар 'Іван)
(Доброго дня Іван)(Як справи?)
**(Болить голова)
(Чому Ви говорите, що Болить голова)
**(у мене нічого не виходить)
(Багато людей відчувають це ж саме)
**(все насправді так)
(Будь ласка, далі)
**(до побачення)
(До побачення Іван)(Побачимося наступного тижня)
```

Хоча на перший погляд здається, що програма все «розуміє» й відповідає користувачеві, але вона використовує лише 2 простих способи отримання відповіді.

Перший спосіб – прийняття «скарги» пацієнта та зміна займенника першої особи (я, мене, мені, мною, мій, моя, моє, мої) на займенник другої особи (ви, вас, вам, вами, ваш, ваша, ваше, ваші) та уточнення відповіді шляхом додавання таких словосполучень, як «`Здається, Ви думаєте, що`», «`Ви відчуваєте, що`», «`Чому Ви вважаєте, що`», «`Чому Ви говорите, що`».

Другий спосіб – ігнорування «скарг» пацієнта та використання таких речень, як «`Будь ласка, продовжуйте`», «`Багато людей відчувають це ж саме`», «`Багато людей вважають так само`», «`Будь ласка, далі`».

Головна функція програми `Лікар` має аргументом лише ім’я пацієнта й викликає функцію `запустити-цикл-лікар`, яка багаторазово приймає дані – скарги пацієнта з клавіатури, здійснює їх обробку та видає результат.

```
(define (Лікар імя)
  (print 'Доброго-дня імя))
  (print '(Як справи?))
  (запустити-цикл-лікар імя))
```

Якщо користувач говорить (до побачення), програма завершує свою роботу, інакше – у відповідності з описаними вище способами вона генерує відповідь.

```
(define (запустити-цикл-лікар імя)
  (newline)
  (print '**)
  (let ((скарга (read)))
    (cond ((equal? скарга '(до побачення))
                (printf "До побачення, ~v. Побачимося на-
ступного тижня." імя))
               (else (print (відповісти скарга))
                        (запустити-цикл-лікар імя)))))

(define (відповісти скарга)
  (cond ((50-50) (append (уточнити)
                         (зміна-особи скарга)))
        (else (продовжити))))
```

Функція `50-50` повертає істину або хибу з однаковою ймовірністю:

```
(define (50-50)
  (= (random 2) 0))
```

Уточнення або ігнорування відповіді здійснюється в функціях `уточнити` та `продовжити` шляхом обрання випадковим чином деякого словосполучення з наведених у списку:

```
(define (уточнити)
  (вибір-випадку '((Здається, Ви думаєте, що)
                   (Ви відчуваєте, що)
                   (Чому Ви вважаєте, що)
                   (Чому Ви говорите, що))))

(define (продовжити)
  (вибір-випадку '((Будь ласка, продовжуйте)
                   (Багато людей відчувають це ж саме)
                   (Багато людей вважають так само)
                   (Будь ласка, далі))))
```

Основою для функції `уточнити` є функція `зміна-особи`, яка приймає відповідь користувача та змінює займенники першої особи на займенники другої особи:

```
(define (зміна-особи фраза)
```

```
  (замінити-все '((я ви) (мені вам) (мною вами)
                  (моє ваше) (мене вас) (мій ваш)
                  (моя ваша) (мої ваші))
                 фраза))

(define (замінити-все пари список)
  (cond ((null? пари) список)
        (else (let ((стар-нов (car пари)))
                (замінити (car стар-нов)
                          (cadr стар-нов)
                          (замінити-все (cdr пари)
                                        список))))))

(define (замінити старий новий список)
  (cond ((null? список) '())
        ((equal? (car список) старий)
          (cons новий
                (замінити старий новий (cdr список))))
        (else (cons (car список)
                    (замінити старий новий (cdr спи-
сок))))))
```

Функція `вибір-випадку` викликає функцію `list-ref` з двома аргументами:

1) список можливих випадків;

2) випадкове число від 0 до довжини списку.

```
(define (вибір-випадку список)
  (list-ref список (random (length список))))
```

Загальну схему викликів функцій зображено на рис. 2.

Завдання:

1. Протестуйте програму. Запишіть результати.

2. Розширте можливості функцій `уточнити` та `продовжити` – додайте нові елементи в їх списки. Протестуйте програму, зберігши зміни.

3. Що буде результатом введення виразу

```
> (зміна-особи '(ви не дуже мені допомагаєте!))
```

Вдоскональте програму так, щоб вона замінювала не лише першу особу на другу, але й другу особу на першу. Наприклад, якщо користувач вводить

```
(ви не дуже мені допомагаєте!))
```

програма має «відповісти» щось приблизно таке:

```
(я відчуваю, що я не дуже вам допомагаю!)
```

Таким чином, необхідно замінити «ви» на «я», «вам» на «мені» і т.д.

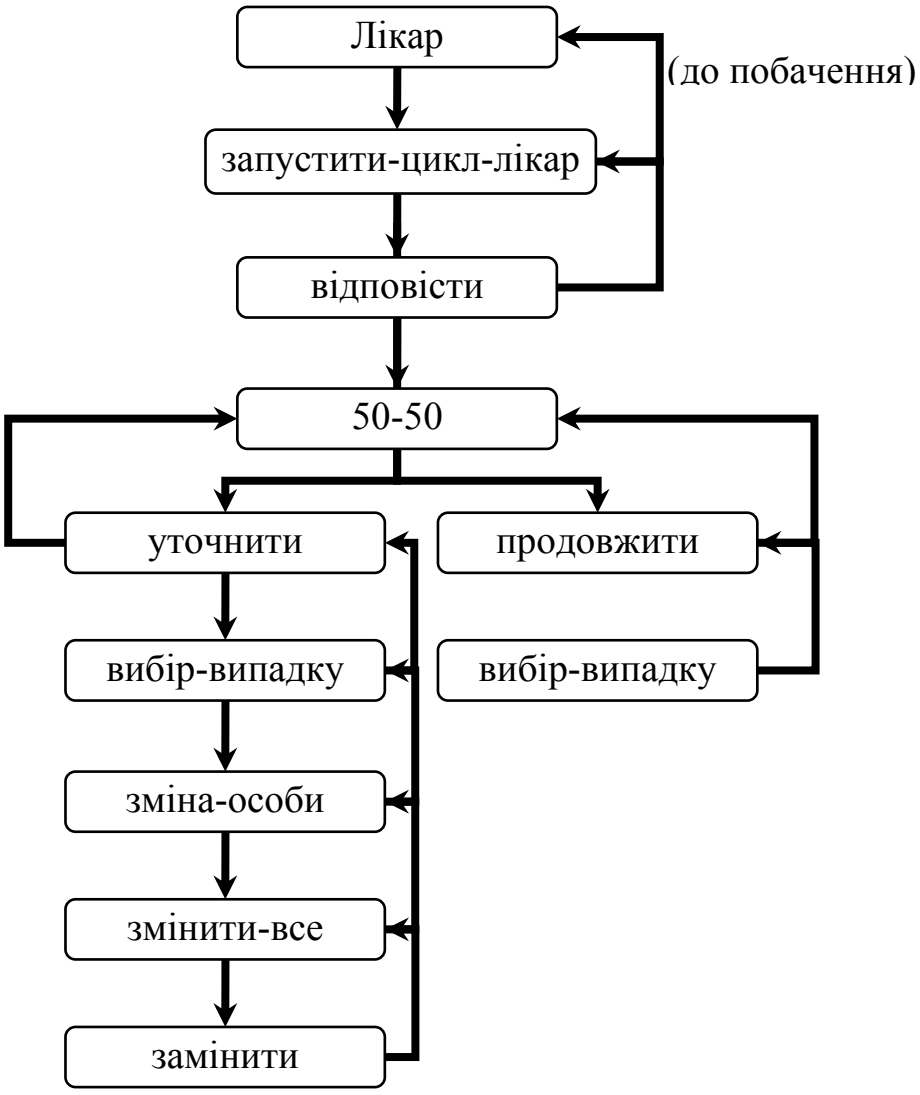


Рис. 2. Загальна схема викликів функцій програми `Лікар`

Спробуйте в функцію `зміна-особи` додати відповідні пари. Протестуйте змінену функцію з запропонованим вище варіантом.

Чи є задовільним отриманий результат? Чи має значення порядок розміщення пар у списку функції `зміна-особи` (спочатку пари «перша особа – друга особа», а потім «друга особа – перша особа» і навпаки)?

Визначте та протестуйте функцію, яка буде працювати коректно для обох видів замін. Результати запишіть.

4. Ще одним видом вдосконалення програми є третій спосіб отримання відповіді: в деяких випадках лікар може відповідати «раніше Ви говорили, що …» або «попередньо Вами було сказано, що …». Наприклад:

«Раніше Ви говорили, що Ви нікого не любите».

Модифікуйте програму так, щоб функція запустити-цикл-лікар зберігала список усіх можливих відповідей.

Внесіть зміни у функцію відповісти, щоб вона використовувала третій спосіб отримання відповіді.

5. Поки що програма працює лише з одним пацієнтом, ім'я якого вона отримує при виклику функції Лікар. Коли пацієнт говорить (до побачення), програма завершується. Внесіть зміни у програму так, щоб функція Лікар могла працювати не лише з одним пацієнтом, а й з декількома. Наприклад, після завершення роботи з одним пацієнтом, результат – запрошення для наступного (наприклад, «Наступний»), а завершуватись програма може, коли замість імені пацієнта вона отримує певний сигнал (наприклад, «Вечір») або після прийому завчасно вказаного числа пацієнтів.

Визначте допоміжну функцію:

```
(define (пацієнт)
(print '(Наступний!))
(print '(Представтесь, будь ласка))
(car (read)))
```

Тепер робота функції «лікар» може виглядати так:

```
(Лікар)
(Наступний!)
(Представтесь, будь ласка) (Іван)
(Доброго дня Іван)
(Як справи?)
   ** (…)
(Будь ласка, далі)
   …
** (До побачення)
(До побачення Іван)
(Побачимося наступного тижня)
(Наступний!)
```

```
(Представтесь, будь ласка) (Петро)
(Доброго дня Петро)
(Як справи?)
** (…)
   ...
** (До побачення)
(До побачення Петро)
(Побачимося наступного тижня)
(Наступний!)
(Представтесь, будь ласка) (Вечір)
(Час додому)
```

Внесіть відповідні зміни у програму.

Протестуйте модифіковану програму.

6. Ще одним способом вдосконалення програми є зміна називного відмінку імені пацієнта на окличний. При цьому будемо вважати, що пацієнт вводить лише своє ім'я.

Внесіть відповідні зміни у програму.

Протестуйте модифіковану програму.

Проект 2. Дилема ув'язненого

Поліція заарештувала двох підозрюваних. Але доказів, достатніх для їх звинувачення, поліція не має. Тому в'язнів ізолювали й зробили обом однакові пропозиції: якщо перший свідчить проти другого, а другий мовчить, то першого звільняють, а другий одержує 10 років тюрми. Якщо мовчать обидва, то їх обох засуджують до 6 місяців. Якщо обидва свідчать один проти одного, то обох засуджують до 2 років. Кожен ув'язнений самостійно обирає: мовчати чи свідчити проти іншого. Проте жоден не знає, що обере інший.

Складемо для цього випадку таблицю. Для більшої узагальненості позначимо в ній випадок мовчання – як співпрацю, а свідчення проти іншого ув'язненого – як зраду, а їх покарання в усіх випадках позначимо як пару (x y), де x та y – строк (у роках) покарання першого та другого в'язнів відповідно.

	Другий співпрацює	Другий зраджує
Перший співпрацює	(0,5 0,5)	(10 0)
Перший зраджує	(0 10)	(2 2)

Припускаючи, що обидва в’язні намагаються мінімізувати свій строк, можна розмірковувати так:

Перший в’язень:

Якщо я викажу другого в’язня, а він буде мовчати, я максимально виграю – мене випускають. Якщо ж другий в’язень теж не буде мовчати – ми обоє отримуємо по 2 роки. Але, якщо я буду мовчати, а він мене зрадить – я отримую максимальний строк – 10 років. Таким чином, для мене краще – виказати другого в’язня.

Аналогічно розмірковує і другий в’язень.

Наведений вище приклад – класична дилема ув’язненого – відома гра в теорії ігор.

Теорія ігор – це математичний метод вивчення оптимальних стратегій в іграх. Гра – процес, в якому можуть приймати участь двоє або більше учасників, кожен з яких намагається отримати максимальний виграш. Кожен із гравців використовує певну стратегію, яка, в залежності від стратегій інших гравців, може привести або до виграшу, або до програшу.

Будь-яку гру можна повністю описати, маючи список її учасників, набір стратегій для кожного учасника та визначення виграшів або платежів гравців для кожної комбінації стратегій.

В нормальній (або стратегічній) формі гра описується матрицею виграшів (платіжною матрицею). Так, зображена вище таблиця – матриця виграшів для двох гравців, у кожного з яких є 2 стратегії: співпрацювати або зрадити.

Зазвичай в нормальній формі подаються ігри, в яких гравці мають одночасно робити ходи, і/або всі гравці не знають про те, що роблять інші учасники. Це ігри з неповними відомостями.

Призначення цієї лабораторної роботи – багаторазове моделювання дилеми ув’язненого.

Загальну схему гри зображено на рис. 3.

Наведений приклад демонструє гру «Дилема ув’язненого» з двома гравцями, кожен з яких на початку програми обирає власну стратегію.

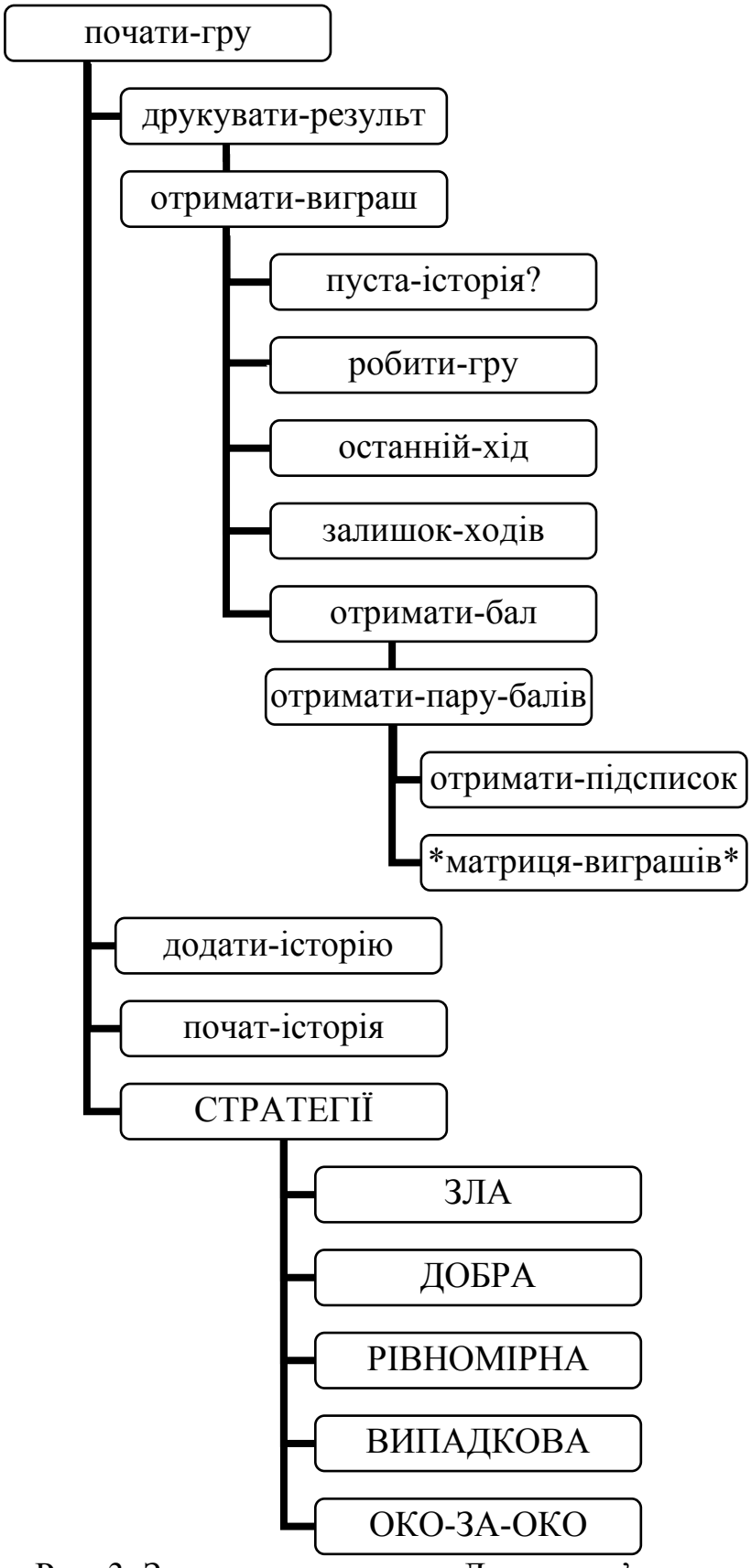


Рис. 3. Загальна схема гри «Дилема ув’язненого»

```
(define (почати-гру страт1 страт2)
  (define (гра-цикл страт1 страт2 лічильник історія1 історія2 ліміт)
    (cond ((= лічильник ліміт)
           (друкувати-результ історія1 історія2 ліміт))
      (else (let ((результ1 (страт1 історія1 історія2))
                  (результ2 (страт2 історія2 історія1)))
              (гра-цикл страт1
                    страт2
                    (+ лічильник 1)
                    (додати-історію результ1 історія1)
                    (додати-історію результ2 історія2)
                     ліміт))))))
  (гра-цикл страт1 страт2 0 початк-історія початк-історія
                 (+ 90 (random 21))))
```

Функція `почати-гру` запускає функцію `гра-цикл`, яка утворює два списки – історії гри двох гравців для обраних ними стратегій.

Кожна стратегія також визначена як функція: в якості аргументів вона приймає історії попередніх ходів гравців – списки, утворені з елементів "співпраця" та "зрада".

Виграші гравців обчислюються у відповідності з `*матрицею-виграшів*`.

```
(define *матриця-виграшів*
  '((("співпраця" "співпраця") (3 3))
    (("співпраця" "зрада") (0 5))
    (("зрада" "співпраця") (5 0))
    (("зрада" "зрада") (1 1))))
```

Таким чином, якщо обидва гравці співпрацюють, їх виграш – по 3 бали кожному, якщо один з гравців співпрацює, а інший – зраджує, то перший отримує 0 балів, а другий – 5 балів.

В програмі описано такі типові стратегії:

`ЗЛА`: завжди зраджує:

```
(define (ЗЛА перша-історія друга-історія)
  "зрада")
```

`ДОБРА`: завжди співпрацює:

```
(define (ДОБРА перша-історія друга-історія)
  "співпраця")
```

РІВНОМІРНА: аналізує історію попередніх ходів другого гравця: якщо кількість співпраць більше за кількість зрад, результат – співпраця, інакше – зрада:

```
(define (РІВНОМІРНА  перша-історія друга-історія)
  (define (к_ть-випадків тест істор)
    (cond ((пуста-історія? істор) 0)
          ((string=? (останній-хід істор) тест)
           (+ (к_ть-випадків тест (залишок-ходів істор))
              1))
          (else
           (к_ть-випадків тест (залишок-ходів істор)))))
  (let ((к-зрад (к_ть-випадків "зрада" друга-історія))
        (к-співпр (к_ть-випадків "співпраця" друга-
історія)))
       (if (> к-зрад к-співпр) "зрада" "співпраця")))
```

ВИПАДКОВА: вибір між співпрацею та зрадою здійснюється випадковим чином:

```
(define (ВИПАДКОВА перша-історія друга-історія)
  (if (= (random 2) 0)
      "співпраця"
      "зрада"))
```

ОКО-ЗА-ОКО: результат – останній хід другого гравця:

```
(define (ОКО-ЗА-ОКО перша-історія друга-історія)
  (if (пуста-історія? перша-історія)
      "співпраця"
      (останній-хід друга-історія)))
```

Для виведення результатів на екран визначена функція друкувати-результ:

```
   (define (друкувати-результ історія1 історія2 к-ходів)
     (let ((виграш (отримати-виграш історія1 історія2)))
       (printf "Кількість ходів: ~v~n
                Виграш 1 гравця: ~v~n
                Виграш 2 гравця: ~v~n"
               к-ходів
               (* 1.0 (/ (car виграш) к-ходів))
               (* 1.0 (/ (cadr виграш) к-ходів)))))
```

Безпосередній підрахунок балів здійснюється функцією отримати-виграш:

```
(define (отримати-виграш історія1 історія2)
  (define (допоміжна історія1 історія2 виграш1 виграш2)
    (cond ((пуста-історія? історія1)
           (list виграш1 виграш2))
     (else (let ((хід (робити-гру(останній-хід історія1)
                                 (останній-хід історія2))))
             (допоміжна (залишок-ходів історія1)
                        (залишок-ходів історія2)
                  (+ (отримати-бал 0 хід) виграш1)
                  (+ (отримати-бал 1 хід) виграш2)))))))
  (допоміжна історія1 історія2 0 0))

(define (отримати-бал номер хід)
  (list-ref (отримати-пару-балів хід) номер))

(define (отримати-пару-балів хід)
  (cadr (отримати-підсписок хід *матриця-виграшів*)))
```

Допоміжні функції:

```
(define робити-гру list)

(define почат-історія '())

(define додати-історію cons)

(define пуста-історія? null?)

(define останній-хід car)

(define залишок-ходів cdr)
```

Завдання:

1. Для перевірки роботи програми необхідно визначити функцію отримати-підсписок, яка в якості аргументів приймає дії гравців – "співпраця" або "зрада" (від кожного гравця, таким чином, змінна «хід» – пара, елементами якої можуть бути всі можливі пари з "співпраця" і "зрада"), в залежності від їх стратегій та список, який складається з підсписків, елементами яких є пари ходів та пари балів, які відповідають цим ходам. Наприклад – *матриця-

виграшів*.

Результатом функції є підсписок, що відповідає даному ходу гравців. Наприклад:

```
> (отримати-підсписок '("зрада" "співпраця") *матриця-виграшів*)
(("зрада" "співпраця") (5 0))
```

Визначте функцію `отримати-підсписок` та протестуйте програму.

2. Передбачте результати для таких стратегій:

– ДОБРА – ДОБРА;

– ЗЛА – ЗЛА;

– ДОБРА – ЗЛА (і навпаки);

– ВИПАДКОВА – ВИПАДКОВА;

– ЗЛА – ВИПАДКОВА (і навпаки);

– ДОБРА – ВИПАДКОВА (і навпаки);

– РІВНОМІРНА – РІВНОМІРНА;

– ЗЛА – РІВНОМІРНА (і навпаки);

– ДОБРА – РІВНОМІРНА (і навпаки);

– ВИПАДКОВА – РІВНОМІРНА (і навпаки);

– ОКО-ЗА-ОКО – ОКО-ЗА-ОКО;

– ЗЛА – ОКО-ЗА-ОКО (і навпаки);

– ДОБРА – ОКО-ЗА-ОКО (і навпаки);

– ВИПАДКОВА – ОКО-ЗА-ОКО (і навпаки);

– РІВНОМІРНА – ОКО-ЗА-ОКО (і навпаки).

Перевірте припущення. Зробіть висновок, для яких стратегій результат можна передбачити заздалегідь. Результати занотуйте.

3. Перевизначте стратегію ОКО-ЗА-ОКО так, щоб гравець з цією стратегією завжди співпрацював, якщо суперник співпрацював обидва попередні ходи.

Протестуйте стратегію.

Зробіть висновок про її ефективність, порівняно з попередньою версією стратегії ОКО-ЗА-ОКО.

4. Реалізуйте програму для трьох гравців, `*матриця-виграшів*` для якої має вигляд:

```
(define *матриця-виграшів*
  '((("співпраця " "співпраця" "співпраця") (4 4 4))
    (("співпраця" "співпраця" "зрада") (2 2 5))
    (("співпраця" "зрада" "співпраця") (2 5 2))
```

```
(("зрада" "співпраця" "співпраця") (5 2 2))
(("співпраця" "зрада" "зрада") (0 3 3))
(("зрада " "співпраця" "зрада") (3 0 3))
(("зрада " "зрада" "співпраця") (3 3 0))
(("зрада " "зрада" "зрада") (1 1 1))))
```

Внесіть зміни лише у функції `почати-гру`, `друкувати-результ` та `отримати-виграш`, хоча, можливо, доведеться змінити й `отримати-пару-балів`.

Протестуйте програму.

Проект 3. Система комп'ютерної алгебри

1. Міксіма – символьний обчислювач

Максіма – це відома алгебраїчна система, розробка якої почалася в Массачусетському технологічному інституті в 60-х роках у рамках проекту MAC. Спочатку дослідження символьної та алгебраїчної обробки математичних виразів (symbolic and algebraic computing, SAC) було пов'язане зі штучним інтелектом, так як до пропонованим нею добре визначеним прикладам були застосовні загальні способи вирішення проблем, однак невдовзі вона перетворилася на окрему самостійну галузь досліджень, що відноситься зараз більше до математики, ніж до штучного інтелекту.

`Міксіма` – це невелика програма символьної математики, схожа на Максіма. Вона може опрацьовувати речення як в чисельному, так і в символьному вигляді. `Міксіма` вміє розпізнавати, диференціювати, спрощувати вирази та виводити їх без зайвих дужок. Діалог з `Міксіма` відбувається шляхом введення виразу або операторів присвоювання, які передбачається обчислити або спростити їх запис. Наприклад:

```
<= 2 + 3 * 4 !          ; арифметична дія
=> 14                   ; значення
<= a : = 2 * b !        ; символьна дія
=> 2 * b                ; символьне значення
<= b : = 3 !            ; присвоювання
=> 3                    ; значення
<= a !                  ; ім'я виразу
=> 6                    ; значення виразу
<= f : = x + 3 * x !    ; присвоювання виразу
=> x + 3 * x            ; значення
```

```
<= f d x !                ; диференціювання
=> 4                      ; похідна
```

Далі структура і робота програми, а також необхідні форми подання та алгоритми будуть розглянуті більш детально.

2. Дії та їх порядок

Для простоти припустимо, що працюючи з Міксіма, при введенні символи відокремлюються один від одного пробілами. Міксіма зчитує вираз, обчислює його значення або перетворює його і виводить відповідь. Поки що при програмуванні обмежимося наведеними нижче діями:

```
(define *дії* '(/ * - + d := ::))
```

Тут +, *, – і + відповідають нормальним арифметичним діям, : = – це присвоювання і d – диференціювання. Також припустимо, що декілька функцій можна писати в одному рядку, розділяючи їх подвійною двокрапкою. *Порядок* виконання дій такий же, як і порядок їх розміщення в списку *дії*. Наприклад, у виразі

x : = a * b / c

наступний порядок дій:

x: = (a * (b / c))

В інших випадках обчислення здійснюються зліва направо. Наприклад:

a – b – c = (a – b) – c

Для простоти обмежимося бінарними операціями (з двома аргументами). Тоді від'ємне символьне значення, наприклад –a, буде записуватись у вигляді 0 – a.

3. Читання виразу з перетворенням в спискову форму

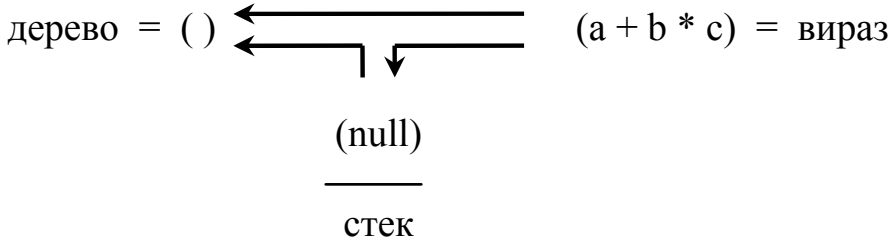
Визначимо перш за все функцію читай, яка зчитує введений користувачем вираз, що закінчуються знаком оклику, і перетворює його в список:

```
(define (читай)
  (let ((x (read)) (res null))
    (do () ((eq? x '!) res)
      (set! res (append res (list x)))
      (set! x (read)))))
> (читай)
a b c !
'(a b c)
```

4. Перетворення виразу у форму дерева

Мова виразів, рекурсивно побудованих з бінарних операцій – це простий контекстно-вільна мова. Аналіз її виразів можна здійснити, наприклад, за допомогою наступного досить простого алгоритму, який називається методом *операторного передування*.

Алгоритм заснований на використанні стеку і його можна подати у вигляді схеми, що складається з виразу, який аналізується, `вираз`, стеку операцій `стек` і дерева `дерево`, яке є результатом аналізу. Спочатку дерево порожнє, а стек – список, що містить порожній список:



Основна ідея алгоритму полягає в наступному. З виразу послідовно один за одним вибираються символи і розміщуються або безпосередньо у дереві, або в стеку на той час, доки інші символи не будуть розміщені або в дереві, або в стеку над цим символом.

Якщо зчитаний символ – це величина (константа або змінна), то його розміщують у дереві. Якщо це – операція, то її розміщують або в дереві, або в стеку відповідно до того, вище або нижче її пріоритет, ніж пріоритет операції, що знаходиться на вершині стеку (або `null`). Коли прочитані всі символи до кінця, всі операції, які після цього знаходяться в стеку, переносяться до дерева і на цьому аналіз закінчується. Стек тут відіграє роль кімнати очікування, в якій операції з більш низьким пріоритетом чекають, доки не виконаються операції, що мають вищий пріоритет. Більш точно алгоритм розбору можна сформулювати наступним чином:

1. Якщо речення порожнє, то перейти до п. 6, інакше взяти наступний символ.

2. Якщо символ є змінною або константою, то перенести його до дерева і перейти до п. 1.

3. Якщо символ – це операція, і стек порожній, то помістити символ у стек і перейти до п. 1.

4. Якщо пріоритет операції вищий за пріоритет верхньої операції, що вже знаходиться в стеку, то помістити її в стек і перейти

до п. 1.

5. Якщо пріоритет операції менший або рівний за пріоритет верхньої операції в стеку, то перемістити операцію зі стеку до дерева і перейти до п. 3.

6. Якщо стек порожній, то дерево розбору готове і алгоритм на цьому закінчується.

7. Перемістити символ з вершини стеку до дерева і перейти до п. 6.

Алгоритм перетворює вираз, дотримуючись правил префіксної нотації.

Якщо у виразі є підвираз, то його можна аналізувати, викликаючи рекурсивно той же алгоритм:

((a + b) * c) → (a b + c *)

В міру переміщення символів в стек або до дерева над ними можна було б виконувати різні дії, що стосуються їх форми або порядку. Переміщуючи, наприклад, до дерева символ операції, можна виконати і саму операцію, якщо тільки, звичайно, у символів-аргументів, що знаходяться в дереві є значення, для яких визначена і здійсненна дана операція. Можна також визначити і деяку форму запису або мову, до якої перетвориться вираз у разі, якщо виконання операції ще неможливо.

5. Подання виразу у формі дерева

З точки зору символьної обробки надається перевага перетворенням виразів у форму дерева, в якій легко опрацьовувати всі підвирази повністю. Так і будемо вчиняти: завжди, коли до дерева переміщується символ операції, будемо вносити зміни, що перетворюють два верхніх елементи дерева і символ операції в дерево в префіксній формі:

(... a * b) → ; два верхніх елементи дерева і;
; символ операції, що додається,
(... (* a b)) ; перетворюються в дерево в префіксній формі

Це перетворення введемо у функцію `аналізуй`, яка реалізує наш алгоритм розбору у вигляді функції `перетвори`.

```
(define (аналізуй дерево стек вираз)
  (cond ((null? вираз)
         (if (not (null? (car стек)))
             (перетвори дерево стек вираз)
             (car дерево)))
```

```
        ((atom? вираз) вираз)
        ((atom? (car вираз))
         (cond ((not (member (car вираз) *дії*))
                (аналізуй (cons (car вираз) дерево)
                          стек (cdr вираз)))
               ((старше (car вираз) (car стек))
                (аналізуй дерево (cons (car вираз) стек)
(cdr вираз)))
               (else (перетвори дерево стек вираз))))
        (else (аналізуй (cons (аналізуй null '(()) (car
вираз))
                              дерево)
                        стек (cdr вираз)))))

(define (перетвори дерево стек вираз)
  (аналізуй (cons (list (car стек)
                        (cadr дерево)
                        (car дерево))
                  (cddr дерево))
            (cdr стек)
            вираз))
```

Визначати пріоритет операцій `p` і `q` буде логічна функція `старше`, яка повертає значення `#t`, якщо `p` старше, тобто стоїть у списку `*дії*` раніше `q`, а інакше повертає `#f`:

```
(define (старше p q)
  (or (null? q) (member q (member p *дії*))))
```

Тепер можна спробувати перетворити вираз у форму дерева:

```
> (аналізуй null (list null) '(a + b * c))
; цей запис аналогічний запису
; (аналізуй '() '(()) '(a + b * c))
'(+ a (* b c))
```

6. Порядок обходу дерева

Вирази, що представляють дерева, можуть бути отримані за допомогою трьох різних порядків обходу, що дозволяють систематично обійти всі вузли дерева. Можливими порядками обходу є:

1. Прямий порядок (preorder).

Префіксний запис (prefix): (+ a (* b c)).

1. Обробити вузол.

2. Обійти ліве піддерево.
3. Обійти праве піддерево.

2. Проміжний порядок (inorder).

Інфіксна запис (infix): (a + (b * c)).

1. Обійти ліве піддерево.
2. Обробити вузол.
3. Обійти праве піддерево.

3. Зворотний порядок (postorder).

Постфіксний запис (postfix, suffix): (a (b c *) +).

1. Обійти ліве піддерево.
2. Обійти праве піддерево.
3. Обробити вузол.

Перший і третій способи названі польським записом на честь їх винахідника польського математика Л. Лукасевича. Їхня перевага перед другим способом полягає в тому, що для них дужки взагалі не потрібні. Порядок виконання операцій і так зрозумілий на підставі порядку проходження символів. Польський запис часто використовують в машинних програмах, трансляторах і інтерпретаторах.

7. Інтерпретація та обчислення виразів

Розглянемо як приклад обчислення виразів в зворотному записі. Обчислення можна проводити, використовуючи простий стековий алгоритм за наступною схемою:

Стек = () <——————————— (2 3 4 * +) = Дерево

1. З дерева зчитується символ і переміщується в стек.
2. Якщо зчитаний символ – це символ операції, то операція виконується над двома верхніми елементами стеку і всі три символи замінюються результатом.
3. Перейти до п. 1.

Наприклад:

Стек	Дія	Дерево
`()`	←	`(2 3 4 * +)`
`(2)`	←	`(3 4 * +)`
`(2 3)`	←	`(4 * +)`
`(2 3 4)`	←	`(* +)`
`(2 3 4 *)`	застосування	`(+)`
`(2 (3 4 *))`	*	`(+)`
`(2 12)`	←	`()`

Стек	Дія	Дерево
(2 12 +)	застосування	()
(2 12 +)	+	()
(14)	←	()

Визначимо тепер функцію `обчисли`, яка діє за тим же принципом, але замість спискової форми використовує дерево в префіксній формі з усіма розставленими дужками. Значення виразу отримується в результаті застосування операції, зазначеної у вузлі, до піддерев, значення яких вже попередньо обчислені рекурсивно за тим же алгоритмом.

```
(define (обчисли x)
  (define значення null)
  (cond ((number? x) x)
        ((atom? x) (set! значення (getprop x 'значення))
                   (if (not (eq? #f значення))
                       (обчисли значення)
                      x))
        (else (застосуй (first x)
                        (map обчисли (cdr x))))))
```

Значення змінних будемо зберігати в списку властивостей символів з ім'ям `значення`.

Функція `застосуй` виконує операцію з виразу над піддеревами:

```
(define (застосуй операція аргументи)
  (define op (getprop операція 'fn))
  (if op
    (apply op аргументи)
    (list op
          (обчисли (first аргументи))
          (обчисли (second аргументи)))))
```

Щоб її можна було здійснити, операція повинна бути визначена як функція, визначення якої зберігається під ознакою `fn` у списку властивостей символу, що є ім'ям операції. У нашому випадку для позначення операції можна використовувати ті ж символи (+, -, * і /), що і в самій Scheme, але вкладаючи в них трохи змінений сенс. Це необхідно, наприклад, під час спрощення запису виразу або щоб повернути операцію в символьному вигляді, коли вираз неможливо обчислити. Якщо операція невідома, то функція засто-

сувати повертає її у формі списку, аргументи якого все ж таки обчислили.

Після цього треба тільки визначити операцію. Використовуємо форму, що поєднує тіло функції з властивостю:

```
(putprop 'дія 'fn (lambda аргументи тіло))
```

8. Спрощення виразів

З метою спрощення арифметичних виразів визначимо для операцій правила спрощення. Будемо враховувати тільки елементарні спрощення, пов'язані з константами 0 і 1 або з рівністю аргументів. Опції спрощення для різних операцій можна визначити наступним чином:

```
(putprop '+ 'fn (lambda (x y)
  (cond ((and (number? x) (zero? x)) y)
        ((and (number? y) (zero? y)) x)
        ((and (number? x) (number? y))  (+ x y))
        (else `(+ ,x ,y)))))

(putprop '- 'fn (lambda (x y)
                  (cond
                    ((and (number? y) (zero? y)) x)
                    ((and (number? x) (number? y))  (- x
y))
                    (else `(- ,x ,y)))))

(putprop '* 'fn (lambda  (x y)
                  (cond
                    ((eq? x 1) y)
                    ((eq? y 1) x)
                    ((or (and (number? x) (zero? x))
(and (number? y) (zero? y))) 0)
                    ((and (number? x) (number? y)) (* x
y))
                    (else `(* ,x ,y)))))

(putprop '/ 'fn (lambda  (x y)
                  (cond
                    ((and (number? x) (zero? x)) 0)
                    ((and (number? y) (zero? y)) 'inf)
                    ((eq? x y) 1)
                    ((and (number? x) (number? y)) (/ x
y))
```

```
                    (else `(/ ,x ,y)))))
```

Семантика команди подвійна двокрапка, призначеної для розділення операторів, і команди присвоювання дещо відрізняється від семантики арифметичних операцій, але і для неї застосуємо використаний вище прийом:

```
(putprop ':: 'fn (lambda  (x y)
                   y))

(putprop ':= 'fn (lambda  (x y)
  (if (symbol? x)
    (begin (putprop x 'значення y) y)
    (error 'помилка "~a неможливо надати значення ~a" x
y))))
```

Операцію диференціювання `d` можна визначити так:

```
(putprop 'd 'fn (lambda  (l x)
  (обчисли (диференціюй l x))))
```

Функцію диференціювання виразу `диференціюй` пропонуємо визначити самостійно, припускаючи, що перший параметр функції – диференційований вираз в префіксній формі, другий – змінна, за якою здійснюється диференціювання.

9. Зняття дужок та виведення

Нам ще потрібна програма виведення результатів, яка перетворює вираз в префіксній формі, отриманий в результаті, в інфіксну форму та прибирає в ньому непотрібні дужки. Зайвими будуть дужки, які не змінюють порядку виконання операцій:

```
(define (зняти-дужки x)
  (if (atom? x)
    x
    (append
      (зняти-у-оператора (first x) (second x))
      (list (first x))
      (зняти-у-оператора (first x) (third x)))))

(define (зняти-у-оператора оператор вираз)
  (define x (зняти-дужки вираз))
  (if (or (atom? x) (старше оператор (second x)))
    (list x)
```

```
    x))
```

10. Діалог з Міксіма

Нарешті необхідна ще програма, яка підтримує діалог між користувачем і Міксіма:

```
(define (Міксіма)
  (print "Міксіма: ")
  (newline)
  (print "<= ")

  (do (( вираз (читай) ) )
    ((equal? вираз '(кінець)) #t)
    (print "=> ")
    (print (зняти-дужки (обчисли (аналізуй null '(()) 
вираз))))
    (newline)
    (print "<= ")
    (set! вираз (читай))))
```

Протестуємо програму:

```
> (Міксіма)
Міксіма:
<= 1
a + b !
=> (a + b)
<= a := 2 !
=> 2
<= b := 5 !
=> 5
<= a + b !
=> 7
<= c := 5 * ( a + b ) !
=> 35
<= ...
```

Завдання:

1. Змініть функцію обчисли таким чином, щоб при виконанні операції присвоювання ліва частина виразу (до знаку :=) не обчислювалася – це дозволить перевизначати значення змінних.

2. Реалізуйте функцію піднесення до степеня ^ та операцію її диференціювання.

3. Реалізуйте функцію ділення без остачі `remainder`.

4. Реалізуйте функцію знаходження залишку від ділення `quotient`.

5. Реалізуйте операцію «порозрядне і» `and`.

6. Реалізуйте операцію «порозрядне або» `or`.

7. Реалізуйте операцію «виняткове або» `xor`

8. Реалізуйте операцію «Арифметичний зсув вліво» `shl`.

9. Реалізуйте операцію «Арифметичний зсув вправо» `shr`.

№	Завдання	№	Завдання
1.	1, 2, 3	16.	1, 2, 4
2.	1, 2, 4	17.	1, 2, 5
3.	1, 2, 5	18.	1, 2, 6
4.	1, 2, 6	19.	1, 2, 7
5.	1, 2, 7	20.	1, 2, 8
6.	1, 2, 8	21.	1, 2, 9
7.	1, 2, 9	22.	1, 2, 3
8.	1, 2, 3	23.	1, 2, 4
9.	1, 2, 4	24.	1, 2, 5
10.	1, 2, 5	25.	1, 2, 6
11.	1, 2, 6	26.	1, 2, 7
12.	1, 2, 7	27.	1, 2, 8
13.	1, 2, 8	28.	1, 2, 9
14.	1, 2, 9	29.	1, 2, 4
15.	1, 2, 3	30.	1, 2, 6

Проект 4. Експертна система

1. Структура експертної системи

Експертна система (система обробки знань) – це програмна система, базу знань та базу даних якої можна порівняти з уміннями та знаннями фахівців у певній спеціальній галузі знань. Експертні системи разом з системами обробки природних мов є найбільш важливими в комерційному плані галузями використання штучного інтелекту.

У рамках досліджень штучного інтелекту створено численні експертні системи для різних галузей знань, таких, наприклад, як медична діагностика та обстеження пацієнтів, генні та молекулярні дослідження, складання конфігурацій комп'ютерів, освіта, пошук

несправностей у пристроях і системах та багато інших практичних застосувань. У цій роботі ми запрограмуємо невелику експертну систему, яку назвемо Експерт.

Типова експертна система складається з двох головних частин: машини виведення і бази знань. База знань містить дані і знання з області застосування; спосіб вирішення проблем, що використовується в машині виведення, не пов’язаний із даними з предметної області. Крім того, в систему обов'язково входить якась мова взаємодії людини з машиною, за допомогою якого користувач-фахівець веде «діалог» з системою, а також до неї входять засоби, за допомогою яких інженер знань та експерт(и) підтримують базу знань.

2. Подання знань

Для подання знань використовують різні формалізми і мови подання даних. Найбільш часто зустрічається подання знань за допомогою продукційних правил типу ЯКЩО-ТО. У системі Експерт правила, що описують прийняття рішення, можна задавати у формі, що схожа на природну мову:

```
(ЯКЩО умова-1
І умова-2
...
І умова-і
ТО висновок-1
І висновок-2
...
І висновок-j)
```

Умови та висновки – це прості речення природної мови. Наприклад:

```
(ЯКЩО на лампочку подано напругу
І лампочка не горить
ТО лампочка, ймовірно, перегоріла)
(ЯКЩО читач перестав розуміти
І читач хоче вчитися
І читач ще може читати
ТО читачеві потрібно почати все спочатку
І читачеві слід бути уважнішим)
```

3. Машина виведення

Машина виведення – це універсальний механізм (програма або апарат), який за допомогою правил бази знань будує нові висновки, задає додаткові питання і так далі доти, доки не прийде до якого-небудь прийнятного кінцевого результату або відповіді. Вихідні дані і висновки, отримані в результаті прийняття рішень, надалі будемо називати відомими системі *фактами*.

У більш великих системах база знань містить окрім правил знання і інших типів: факти про об'єкти проблемної області й їх властивості, дані про обставини та події, ієрархії понять, метадані і т.д.

Робота експертних систем ґрунтується, в першу чергу, на великій базі знань. Робота машин виведення часто досить проста і прямолінійна. Розглянемо більш детально структуру машини прийняття рішень програми Експерт.

4. Факти і правила

У системі Експерт факти подаються просто у вигляді списків символів. Наприклад, наступний список міг би описувати наші знання про яку-небудь тварину:

```
((Тварина має шерсть)
 (Тварина смугаста)
 (Тварина жуйна))
```

База знань системи Експерт утворюється зі списків, подібних раніше описаним правилами ЯКЩО-ТО. Наприклад:

```
(ПРАВИЛО12 .
     (ЯКЩО тварина жуйна
      І тварина смугаста
      ТО тварина зебра))
```

Щоб правила можна було звести до фактів, їх умови і висновки необхідно представляти списками фактів. Для цього вирази, які є правилами, можна перетворити в структури, які складаються з імені правила, умов і висновків, представлених у вигляді списку фактів:

```
; Перетворює правило у формі речення
; на структуру – список з 3-х компонентів
(define (аналізуй-правило правило)
  (list (car правило)
        (умови (cdr правило))
        (висновки (cdr правило))))
```

Якщо умови правила є елементами якого-небудь списку фактів, то застосування правила до цього списку фактів можна реалізувати шляхом додавання в список висновків з правила. Наприклад, застосувавши ПРАВИЛО12 до розглянутого вище списку даних, можна зробити висновок: тварина, про яку йде мова – зебра.

5. Правила виведення бази знань

Знання системи Експерт про тварин містяться в базі знань, яка містить 14 правил:

```
(define *база-знань* '(
  (ПРАВИЛО1 .
    (ЯКЩО тварина має шерсть
     ТО тварина ссавець))
  (ПРАВИЛО2 .
    (ЯКЩО тварина годує дитинчат молоком
     ТО тварина ссавець))
  (ПРАВИЛО3 .
    (ЯКЩО тварина має пір'я
     ТО тварина птах))
  (ПРАВИЛО4 .
    (ЯКЩО тварина вміє літати
     І тварина несе яйця
     ТО тварина птах))
  (ПРАВИЛО5 .
    (ЯКЩО тварина їсть м'ясо
     ТО тварина хижак))
  (ПРАВИЛО6 .
    (ЯКЩО тварина має гострі зуби
     І тварина має пазурі
     І очі тварини посаджені прямо
     ТО тварина хижак))
  (ПРАВИЛО7 .
    (ЯКЩО тварина ссавець
     І тварина має копита
     ТО тварина жуйна))
  (ПРАВИЛО8 .
    (ЯКЩО тварина ссавець
     І тварина жує жуйку
     ТО тварина жуйна))
  (ПРАВИЛО9 .
    (ЯКЩО тварина ссавець
```

```
      І тварина хижак
      І тварина жовто-коричнева
      І тварина має темні плями
      ТО тварина гепард))
   (ПРАВИЛО10 .
     (ЯКЩО тварина ссавець
      І тварина хижак
      І тварина жовто-коричнева
      І тварина смугаста
      ТО тварина тигр))
   (ПРАВИЛО11 .
     (ЯКЩО тварина жуйна
      І тварина довгошия
      І тварина довгонога
      І тварина має темні плями
      ТО тварина жираф))
   (ПРАВИЛО12 .
      (ЯКЩО тварина жуйна
       І тварина смугаста
       ТО тварина зебра))
   (ПРАВИЛО13 .
      (ЯКЩО тварина птах
       І тварина не вміє літати
       І тварина довгошия
       І тваринадовгонога
       І тварина чорно-біла
       ТО тварина страус))
   (ПРАВИЛО14 .
      (ЯКЩО тварина птах
       І тварина не вміє літати
       І тварина плаває
       І тварина чорно-біла
       ТО тварина пінгвін))))
```

Наступна функція `аналізуй` бере список правил (у нашому випадку – це `*база-знань*`), аналізує кожне правило і перетворює його до раніше описаної структури (`ПРАВИЛО`), полями якої є: ім'я правила, умови і висновки:

```
; Функція для додавання елемента у кінець списка
(define (приєднай x y)
  (append x (list y)))

; Аналізує правила та створює з них список
```

```
(define (аналізуй правила)
  (map аналізуй-правило правила))
; Функції доступу до елементів структури
(define (правило-ім'я правило)
   (car правило))

(define (правило-умови правило)
   (cadr правило))

(define (правило-висновки правило)
   (caddr правило))

; Повертає умови у вигляді списка
(define (умови речення)
  (речення-і (cdr речення) null null))

; Повертає висновки у вигляді списка
(define (висновки речення)
  (речення-і (cdr (member 'ТО речення)) null null))

; Виділяє речення, що розділяються прийменниками
; І, ТА, Й
(define (речення-і речення частина результат)
  (cond
    ((null? речення) ; умова закінчення
     (приєднай результат частина))
    ((member (car речення) '(ТО то)) ;; умова закінчення
     (приєднай результат частина))
    ((member (car речення) '(І ТА Й і та й))
; нове речення
     (речення-і
        (cdr речення)
        null
        (приєднай результат частина)))
    (else ; наступне слово
      (речення-і
        (cdr речення)
        (приєднай частина (car речення))
        результат))))
```

Правила у вигляді записів легко обробляти, і функціям більш високого рівня нічого не треба знати про аналіз даних або деталі подання правил. Надалі це полегшує можливу модифікацію про-

грам і їх подальший розвиток.

Проаналізовані варіанти правил збережемо у змінній *правила* наступною командою:

```
(define *правила* (аналізуй *база-знань*))
```

Значенням змінної *правила* буде список, що складається з структур.

Припустимо, що система Експерт зберігає відомі їй факти у списку *факти*, що складається з елементів даних. У цьому випадку застосовність правила можна перевірити функцією перевірити-правило і виведення правила можна при необхідності додати до фактів функцією додати-висновки:

```
; Перевіряє, чи можна застосувати правило
(define (перевірити-правило правило)
  (підмножина (правило-умови правило) *факти*))

; перетин множин
(define (перетин l1 l2)
  (cond ((or (not (list? l1)) (not (list? l2)))
         (error "Параметрами мають бути списки"))
        ((or (null? l1) (null? l2)) null)
        ((not (member (car l1) l2))
         (remove-duplicates (перетин (cdr l1) l2)))
        (else (remove-duplicates
                    (cons (car l1)
                          (перетин (cdr l1) l2))))))

; об'єднання множин
(define (об’єднання l1 l2)
  (cond ((or (not (list? l1)) (not (list? l2)))
         (error "Параметрами мають бути списки"))
        ((null? l1) l2)
        ((null? l2) l1)
        ((member (car l1) l2)
         (remove-duplicates (об’єднання (cdr l1) l2)))
        (else (remove-duplicates
                    (cons (car l1)
                          (об’єднання (cdr l1) l2))))))

; Перевіряє, чи є множина підмножиною
(define (підмножина підмножина множина)
  (equal? підмножина (перетин підмножина множина)))
```

```
; Розширює список фактів висновками правила
(define (додати-висновки правило)
  (do ((висновки (правило-висновки правило)
                 (cdr висновки)))
    ((null? висновки) *факти*)
    (if (member (car висновки) *факти*)
      null
      (begin
        (printf "Згідно правила ~a: " (правило-ім'я пра-
вило))
        (вивести-елементи (car висновки))
        (set! *факти* (cons (car висновки) *факти*))))))

; Виводить елементи списка
(define (вивести-елементи список)
  (for ([елемент список])
    (printf "~a " елемент))
  (newline)
  #t)
```

6. Стратегія оберненого виведення

Машина виведення застосовує правила до відомих у поточний момент системі фактів для знаходження нових фактів доти, доки в результаті застосування не буде отриманий шуканий результат. Якщо у системи недостатньо відомостей для подальшого пошуку рішення, то вона запитує у користувача додаткові відомості і зберігає їх у своєму списку фактів, а потім намагається ще раз застосувати до своїх правил нові додаткові факти і т.д.

У дослідженнях зі штучного інтелекту створені різні способи знаходження рішення і виконання дій. У системі `Експерт` ми застосуємо *зворотнє виведення*, в якому рішення намагаються знайти, йдучи у зворотному напрямі від кінцевого результату.

7. Робота системи `Експерт`

Систему `Експерт` на самому верхньому рівні можна представити як розпізнавача тварин, що намагається на основі своїх правил довести деяку гіпотезу.

Подамо можливі кінцеві результати у вигляді списку *гіпотези*:

```
(define *гіпотези*
  '((тварина пінгвін)
```

```
    (тварина страус)
    (тварина зебра)
    (тварина жираф)
    (тварина тигр)
    (тварина гепард)))
```

Всі ці гіпотези зустрічаються в частині виведення деяких правил. При зворотному виведенні умови цих правил можна інтерпретувати як нові гіпотези, якщо висновок є кінцевим результатом, і так далі. Так система породжує гіпотези нижчого рівня до тих пір, доки не виявить, що нових правил для породження гіпотез більше немає. Тоді система запитує умови правила безпосередньо у користувача, які він на цьому рівні вже, ймовірно, зможе поставити сам і не будучи фахівцем, потім система за допомогою нових відомостей намагається рухатися далі у своїх висновках:

```
(define *факти* null)
(define *запити* null)

(define (Експерт)  ; Експертна система
  (set! *факти* null)
  (set! *запити* null)
  (print (машина-виведення *гіпотези*)))

; Намагається перевірити яку-небудь гіпотезу
(define (машина-виведення гіпотези)
  (cond ((null? гіпотези)
      "Не можу довести жодну з відомих мені гіпотез")
        ((доведи (car гіпотези))
         (car гіпотези)) ; результат
        (else (машина-виведення (cdr гіпотези)))))
; нова спроба
```

Доведення гіпотези або твердження, можна виконати за допомогою такої функції:

Функція `доведи`:

Дано: гіпотеза, що доводиться.

Значення: значення функції – істина, якщо дану гіпотезу можна довести

Дії:

1. Якщо гіпотеза вже є в списку фактів, значить вона доведена.

2. Якщо це не так, то зберемо всі правила, за допомогою яких можна було б довести гіпотезу, тобто ті правила, в частині виве-

дення яких є ця гіпотеза. Якщо яке-небудь з цих правил можна прямо застосувати до списку фактів, то гіпотеза доведена.

Якщо жодне з вибраних правил не можна застосувати, то спробуємо довести застосовність будь-якого правила, довівши всі умови правила, взявши їх як нові гіпотези і рекурсивно застосувавши функцію `доведи`.

3. Якщо дії 1 і 2 не дають результату, то запитаємо у користувача, чи вірна гіпотеза (якщо тільки це ще не питалось), і якщо вона вірна, то приєднаємо твердження до списку фактів. Приєднаємо твердження до списку `*запити*` незалежно від відповіді, щоб потім не довелося повторно запитувати це ж саме.

Цю рекурсивну функцію можна безпосередньо запрограмувати. Всі три гілки описаної функції відмічені у визначенні відповідними коментарями:

```
(define (доведи гіпотеза)
  (cond
    ((member гіпотеза *факти*) #t)          ; гілка 1
    ((not (null? (можливі гіпотеза)))       ; гілка 2
       (if (прямо гіпотеза (можливі гіпотеза))
          #t
          (рекурсивно гіпотеза (можливі гіпотеза))))
    (else (гілка3 гіпотеза))))              ; гілка 3

(define (гілка3 гіпотеза)
  (cond
    ((member гіпотеза *запити*) #f)
    ((and(display "Чи це правда, що: ")
         (вивести-елементи гіпотеза)
         (member (read) '(Так так)))
     (begin
    (set! *факти* (об'єднання (list гіпотеза) *факти*))
       #t))
    (else
     (begin
     (set! *запити* (cons гіпотеза *запити*))
       #f))))

(define (можливі гіпотеза)
  (let ((result null))
    (for ([правило *правила*])
      (when (member гіпотеза (правило-висновки правило))
```

```
        (set! result (append result (list правило)))))
    result))

; Перевіряє, чи можна довести гіпотезу безпосередньо
; за допомогою якого-небудь правила
(define (прямо гіпотеза можливі)
  (cond ((null? можливі) #f)
        ((null? *факти*) #f)
        ((перевірити-правило (car можливі))
         (додати-висновки (car можливі)))
        (else (прямо гіпотеза (cdr можливі)))))

; Рекурсивно перевіряє гіпотезу
(define (рекурсивно гіпотеза можливі)
  (cond ((null? можливі) #f)
        ((перевірити-непрямо (правило-умови (car можли-
ві)))
         (додати-висновки (car можливі)))
        (else (рекурсивно гіпотеза (cdr можливі)))))

; Рекурсивно перевіряє всі умови
(define (перевірити-непрямо умови)
  (for/and ([умова умови])
    (доведи умова)))
```

Керуючись своїми правилами, програма Експерт може задавати користувачеві лише розумні з точки зору гіпотези питання. Проте першу гіпотезу доводиться вибирати навмання.

8. Приклади запитів

Тепер трохи «поговоримо» з програмою Експерт. У першому прикладі користувач бачить перед собою пінгвіна, у другому – зебру, але тварин він не знає. За допомогою програми Експерт він може визначити види цих тварин, даючи системі прості відповіді на її питання:

```
> (Експерт)
Чи це правда, що: тварина має пір'я
Так
Згідно правила ПРАВИЛО3: тварина птах
Чи це правда, що: тварина не вміє літати
так
Чи це правда, що: тварина плаває
```

```
так
Чи це правда, що: тварина чорно-біла
так
Згідно правила ПРАВИЛО14: тварина пінгвін
(тварина пінгвін)
> (Експерт)
Чи це правда, що: тварина має пір 'я
ні
Чи це правда, що: тварина вміє літати
ні
Чи це правда, що: тварина має шерсть
так
Згідно правила ПРАВИЛО1: тварина ссавець
Чи це правда, що: тварина має копита
так
Згідно правила ПРАВИЛО7: тварина жуйна
Чи це правда, що: тварина смугаста
так
Згідно правила ПРАВИЛО12: тварина зебра
(тварина зебра)
```

Завдання:

1. Доповніть експертну систему меню, що містить наступні пункти:

1. Провести експертну оцінку.
2. Вивести дані про ... .
3. Пояснити останнє виведення.
4. Завершення роботи.

В результаті обрання першого пункту повинна запускатися машина виведення, другого – вводитися запит про предмет і виводитися всі правила і гіпотези, його містять, третього – в зручній формі виводитись ланцюжок висновків останнього твердження (гіпотези) у вигляді послідовностей «Так як ..., можна зробити висновок, що ...». Вибір четвертого пункту повинен призводити до завершення роботи системи.

2. Створіть експертну систему, що допомагає вирішити, яку формулу застосовувати при роз’язуванні завдань зі шкільного курсу фізики (розділи кінематика і динаміка). В якості гіпотези задайте формули, правила побудуйте на основі наступного фрагменту бази знань:

багатозначний(формула)

```
дозвзн(розділ)=кінематика,динаміка
дозвзн(рух_тіла)=прямолінійний,обертальний
дозвзн(швидкість)=постійна,рівнозмінна
дозвзн(причини_руху)=не_враховуються,враховуються
до-
звзн(сила)=сила_тертя,сила_тяжіння,сила_пружності,декілька_сил
дозвзн(зад_сил)=задані,не_задані
питання(причини_руху)=Чи враховуються причини руху?
питання(рух_тіла)=Який характер руху тіла?
питання(швидкість)=Який характер швидкості?
питання(сила)=Яка сила розглядається в задачі?
питання(зад_сил)=Чи задані сили?

правило0:
якщо розділ=кінематика
то підрозділ=невизн.

правило1:
якщо причини_руху=не_враховуються
то розділ=кінематика.

правило2:
якщо причини_руху=враховуються
то розділ=динаміка.

правило3:
якщо розділ=кінематика
і рух_тіла=прямолінійний
і швидкість=постійна
то формула= X=X0+V*t
і формула= V=S/t.

правило4:
якщо розділ=кінематика
і рух_тіла=прямолінійний
і швидкість=рівнозмінна
то формула= a=(V-V0)/t
і формула= V=V0+a*t
```

і формула= X=X0+V0*t+A*t^2/2
bmp=pryskor.

правило5:
якщо розділ=кінематика
і рух_тіла=обертальний
то формула= w=f/t
і формула= V=w*r
і формула= a=V^2/r.

правило6:
якщо розділ=динаміка
і зад_сил=задані
то підрозділ=рух_під_дією_сил.

правило7:
якщо розділ=динаміка
і зад_сил=не_задані
то підрозділ=закони_ньютона.

правило8:
якщо розділ=динаміка
і підрозділ=закони_ньютона
то формула= F=M*a
і формула= F1=-F2.

правило9:
якщо розділ=динаміка
і підрозділ=рух_під_дією_сил
і сила=сила_тертя
то формула= F=-k*N.

правило10:
якщо розділ=динаміка
і підрозділ=рух_під_дією_сил
і сила=сила_тяжіння
то формула= F=G*M1*M2/r^2.

правило11:
якщо розділ=динаміка
і підрозділ=рух_під_дією_сил
і сила=сила_пружності
то формула= F=-k*X.

правило12:
якщо розділ=динаміка
і підрозділ=рух_під_дією_сил
і сила=декілька_сил
то формула= m*a=F1+F2+ +FN.
кінець

3. Створіть експертну систему `Класифікатор-рослин` на основі наступного фрагменту бази знань:

Якщо клас голонасінні і форма листка лускоподібна, то сімейство – кипарисові.

Якщо клас голонасінні і форма листка голкоподібна і конфігурація хаотична, то сімейство – соснові.

Якщо клас голонасінні і форма листка голкоподібна і конфігурація – 2 рівних ряди і сріблястa смуга, то сімейство – соснові.

Якщо клас голонасінні і форма листка голкоподібна і конфігурація – 2 рівних ряди і сріблястої смуги немає, то сімейство – болотний кипарис.

Якщо тип – дерева і форма листка широка і пласка, то клас – покритонасінні.

Якщо тип – дерева і невірно, що форма листка широка і пласка, то клас – голонасінні.

Якщо стебло зелене, то тип – трав’янисті.

Якщо стебло дерев’янисте і положення, що стелиться, то тип – ліани.

Якщо стебло дерев’янисте і положення прямостояче і один основний стовбур, то тип – дерева.

Якщо стебло дерев’янисте і невірно, що положення прямостояче і один основний стовбур, то тип – чагарникові.

4. Створіть експертну систему `Прогнозування-повеней` на основі наступного фрагмента бази знань:

1. якщо рівень-води = високий і
   дощ = рясний,

то місто = евакуювати
2. якщо рівень-води = високий і
дощ = не сильний і
снігу = багато,
температура = висока,
то місто = евакуювати
3. якщо рівень-води = високий і
дощ = не сильний і
снігу = багато,
температура = середня і
дощ = помірний,
то місто = посилити увагу
4. якщо рівень-води = високий і
дощ = ні і
снігу = багато,
температура = середня і
дощ = слабкий,
то місто = не турбуватись
5. якщо рівень-води = високий і
дощ = не сильний і
снігу = мало,
то місто = не турбуватись
6. якщо рівень-води = невисокий і
дощ = сильний і
снігу = багато,
температура = висока,
то місто = посилити увагу
7. якщо рівень-води = невисокий і
дощ = сильний і
снігу = багато,
температура = середня,
то місто = не турбуватись
8. якщо рівень-води = невисокий і
дощ = сильний і
снігу = мало,
то місто = не турбуватись
9. якщо рівень-води = високий і
дощ = несильний,

то місто = не турбуватись

5. Створіть експертну систему `Класифікатор-птахів`, яка розрізняє орла, сокола і горобця, використовуючи в якості принципів класифікації ім'я об'єкта, довжину дзьоба, забарвлення пір'я, розмах крил, висоту польоту, аеродинамічні принципи.

6. Створіть експертну систему `Прийом-на-роботу`, яка реалізує схему, зображену на рис. 4.

№	Завдання	№	Завдання
1.	1, 2, 3	16.	1, 2, 4
2.	1, 2, 4	17.	1, 2, 3
3.	1, 2, 5	18.	1, 2, 4
4.	1, 2, 6	19.	1, 2, 5
5.	1, 2, 7	20.	1, 2, 6
6.	1, 2, 3	21.	1, 2, 7
7.	1, 2, 4	22.	1, 2, 3
8.	1, 2, 5	23.	1, 2, 4
9.	1, 2, 6	24.	1, 2, 5
10.	1, 2, 7	25.	1, 2, 6
11.	1, 2, 3	26.	1, 2, 7
12.	1, 2, 4	27.	1, 2, 3
13.	1, 2, 5	28.	1, 2, 4
14.	1, 2, 6	29.	1, 2, 5
15.	1, 2, 7	30.	1, 2, 6

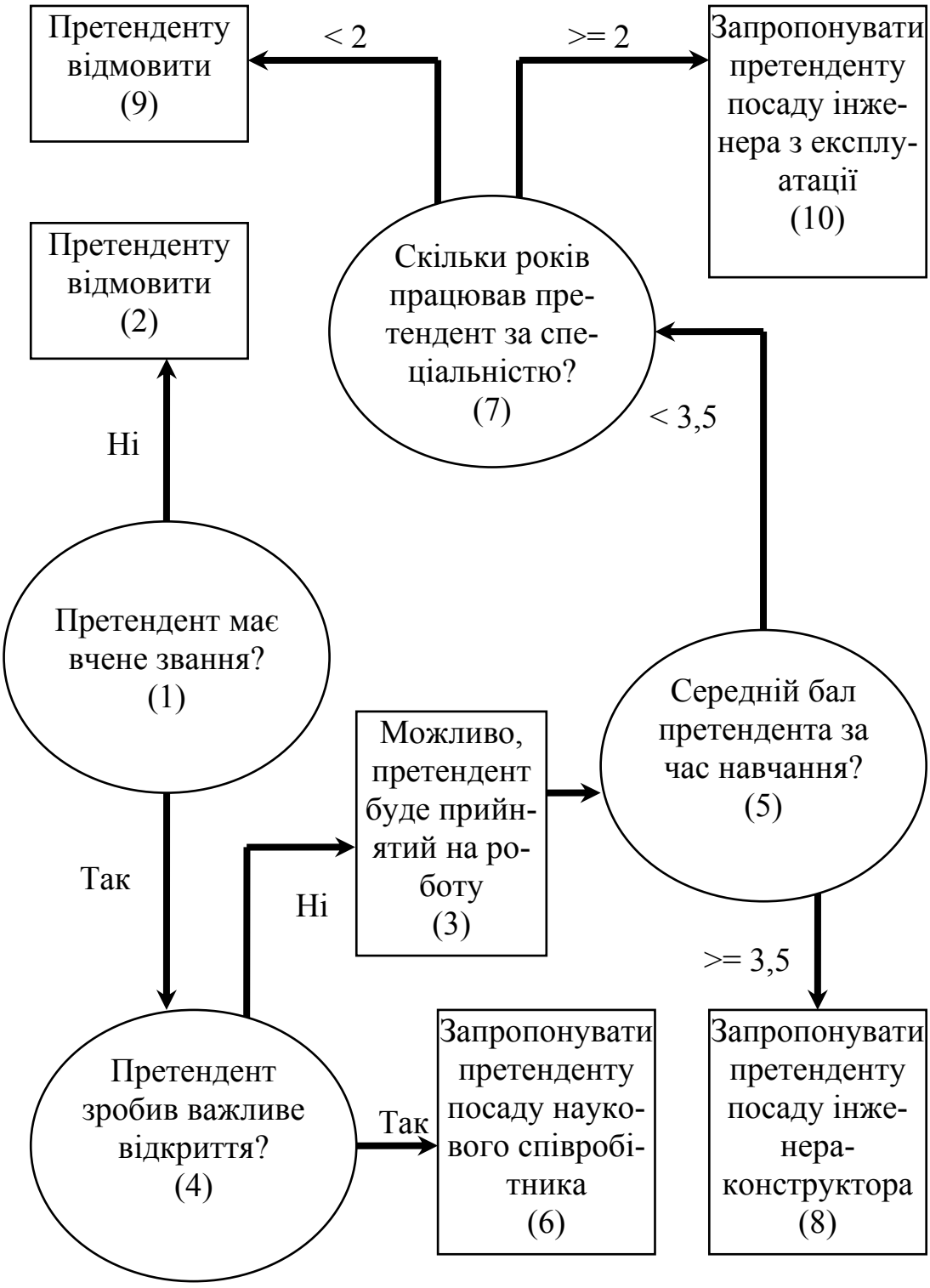


Рис. 4. Схема експертної системи `Прийом-на-роботу`

Додатки

А. Додаткові можливості Scheme – графічний інтерфейс користувача

Для створення графічного інтерфейсу користувача необхідно визначити мову racket/gui, оскільки вона поєднує в собі всі прив'язки необхідних модулів. Тому будь-яку програму з графічним інтерфейсом користувача необхідно починати з `#lang racket/gui`.

Основні конструкції для програм з графічним інтерфейсом – це вікна: рамки, діалоги, меню, кнопки, флажки, текстові поля, радіо-кнопки та ін. Цей інструментарій надається через вбудовані класи. За домовленістю, імена класів закінчуються %. Наприклад, `frame%`:

```
; створимо Рамку як об'єкт класу frame%
(define Рамка (new frame% [label "Приклад"]))
; відобразимо Рамку
; пошлемо об'єкту Рамка
; повідомлення show з параметром #t
(send Рамка show #t)
```

Результат – рис. 5.

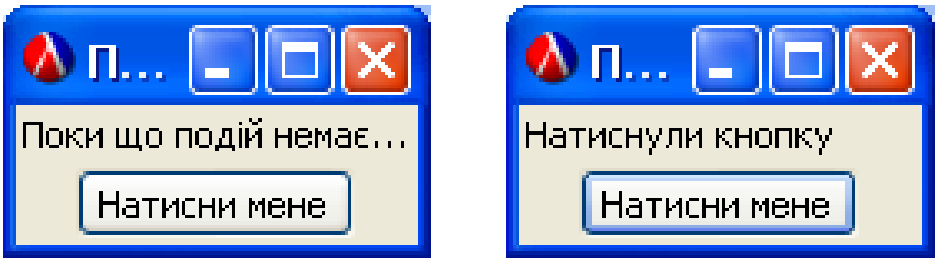
Рис. 5. Вікно-рамка мінімальних розмірів

Будь-який клас має визначені поля та повідомлення (функції). Вбудовані класи забезпечують різні способи обробки повідомлень. Наприклад, при створенні об'єкту класу `button%`, можна визначити функцію оберненого виклику для випадку, коли користувач натискає кнопку.

Наступний приклад створює рамку, повідомлення з заголовком «Поки що подій немає» та кнопку. При натисненні кнопки, заголовок повідомлення змінюється на «Натиснули кнопку» (рис. 6):

```
; створимо рамку як об'єкт класу frame%
  (define Рамка (new frame%
      [label "Приклад"]))

; створимо повідомлення на рамці
  (define Повідомлення (new message%
```

```
      [parent Рамка]
      [label "Поки що подій немає..."]))

; створимо кнопку на рамці
  (new button%
      [parent Рамка]
      [label "Натисни мене"]
; функція оберненого виклику для натиснення кнопки
      [callback (lambda (button event)
                  (send Повідомлення set-label "Натис-
нули кнопку"))]))

; відобразимо рамку
  (send Рамка show #t)
```



а) до натиснення кнопки б) після натиснення кнопки

Рис. 6. Обробка подій

В системі розсилок графічного інтерфейсу користувача повідомлення опрацьовуються послідовно, тобто, наступна функція оберненого виклику або повідомлення чекають, доки обробник подій повернеться за наступним повідомленням. Для ілюстрації послідовності обробки повідомлень додамо у попередній код ще одну кнопку:

```
; створимо на рамці ще одну кнопку,
; натиснення якої призводить до без дієвості
; - "засипання" об'єкту на 5 секунд
  (new button%
      [parent Рамка]
      [label "Пауза"]
      [callback (lambda (button event)
        (sleep 5))])
```

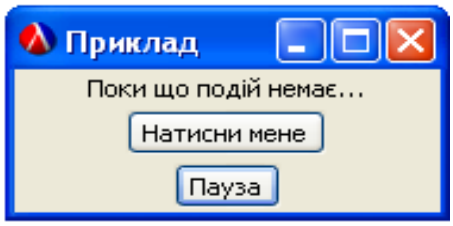


Рис. 7. Послідовність обробки подій

При натисненні кнопки «Пауза», вся рамка перестає реагувати на будь-які події протягом 5 секунд. Але після закінчення 5 секунд, всі неопрацьовані події будуть послідовно виконані.

На додачу до обробки повідомлень та функцій оберненого виклику, в графічному інтерфейсі користувача можна керувати розміщенням елементів, як шляхом порядку їх створення, так і шляхом задання їх батьківських об'єктів. Наприклад, всі об'єкти на вертикальній панелі розміщуються зверху вниз, а на горизонтальній – зліва направо (якщо не задано їх інакше розміщення шляхом указування координат верхнього лівого кута).

Типи вікон:

– **контейнери** – вікна, що можуть містити інші вікна:

– `frame%` – рамка – **вікно верхнього рівня** класу Вікон, які користувач може переміщувати та змінювати розмір;

– `dialog%` – діалогове **вікно верхнього рівня** класу Вікон, коли відображається діалогове вікно, всі інші вікна є недієздатними, доки це вікно не буде закрите;

– `panel%` – панель – підконтейнер всередині контейнерів. Існують такі підкласи класу panel%: vertical-panel% та horizontal-panel%;

– **підконтейнери** – вікна, що мають міститись в інших вікнах:

– `panel%`;

– `pane%`;

– **полотна:**

– `canvas%` – полотно для малювання на екрані;

– `editor-canvas%` – редактор, допоміжне полотно для відображення текстового редактора або вклеєного редактора;

– **елементи керування:**

– `message%` – напис;

– `button%` – кнопка;

– `check-box%` – кнопка-прапорець, віконце, яке користувач клацанням миші може вмикати або вимикати; зазвичай кнопку «ввімкнуто», якщо це віконце має вигляд квадрата з буквою X або галочкою всередині; якщо квадрат порожній, то кнопку «вимкнуто»; стан такої кнопки не впливає на інші кнопки в діалоговому вікні;

– `radio-box%` – кнопка із залежною фіксацією; вибір (активізація) такої кнопки (зазвичай позначається точкою всередині неї) визначає одну з взаємовиключних функцій;

– `choice%` – вибір, пункт меню;

– `list-box%` – вікно з елементами списку, користувач обирає один або декілька елементів (в залежності від стилю списку);

– `text-field%` – текстове поле для введення тексту;

– `combo-field%` – комбінований елемент: поєднує текстове поле з вибором меню;

– `slider%` – бігунок, повзунок на лінійці зі шкалою з числами;

– `gauge%` – індикатор тільки вихідного контролю (користувач не може змінити значення) для подання цілого числа у заданому діапазоні.

Діаграма, зображена на рис. 8, показує ієрархію віконних класів. Розглянемо детальніше виокремлені класи.

Клас «рамка» – `frame%`

Рамка є вікном верхнього рівня. Вона має рядок з заголовком, рядок меню та рядок статусу.

Поле	**Пояснення**	**Можливі значення # значення за замовчуванням**
обов’язкові поля		
label	заголовок вікна	рядок
необов’язкові поля		
parent	батьківський об’єкт	об’єкт класу `frame%` # `#f`
width	ширина	ціле число (0; 10000) # `#f`
height	висота	ціле число (0; 10000) # `#f`
x	x-координата лівого	ціле число (-10000; 10000) # `#f`

Поле	Пояснення	Можливі значення # значення за замовчуванням
	верхнього кута	
y	y-координата лівого верхнього кута	ціле число (-10000; 10000) # `#f`
style	стиль	список з елементів: `no-resize-border` – неможливо змінити розміри вікна `no-caption` –без заголовку `no-system-menu` – без системного меню (згорнути, розгорнути, закрити) `hide-menu-bar` – без рядка меню `toolbar-button` – без панелі інструментів `float` – завжди поверх інших вікон `metal` – фон металевого кольору # `null`
enabled	«ввімкненість»	логічний # `#t`
border	стиль границі	ціле число (0; 1000) # 0
alignment	вирівнювання тексту	список, утворений по 1 елементу з підсписків '(`left center right`) '(`top center bottom`) # '(`center top`)
min-width	мінімальна ширина	ціле число (0; 10000) # мінімальна графічна ширина
min-height	мінімальна висота	ціле число (0; 10000) # мінімальна графічна висота
stretchable-widtht	здатність розтягуватись у ширину	логічний # `#t`
stretchable-height	здатність розтягуватись у висоту	логічний # `#t`

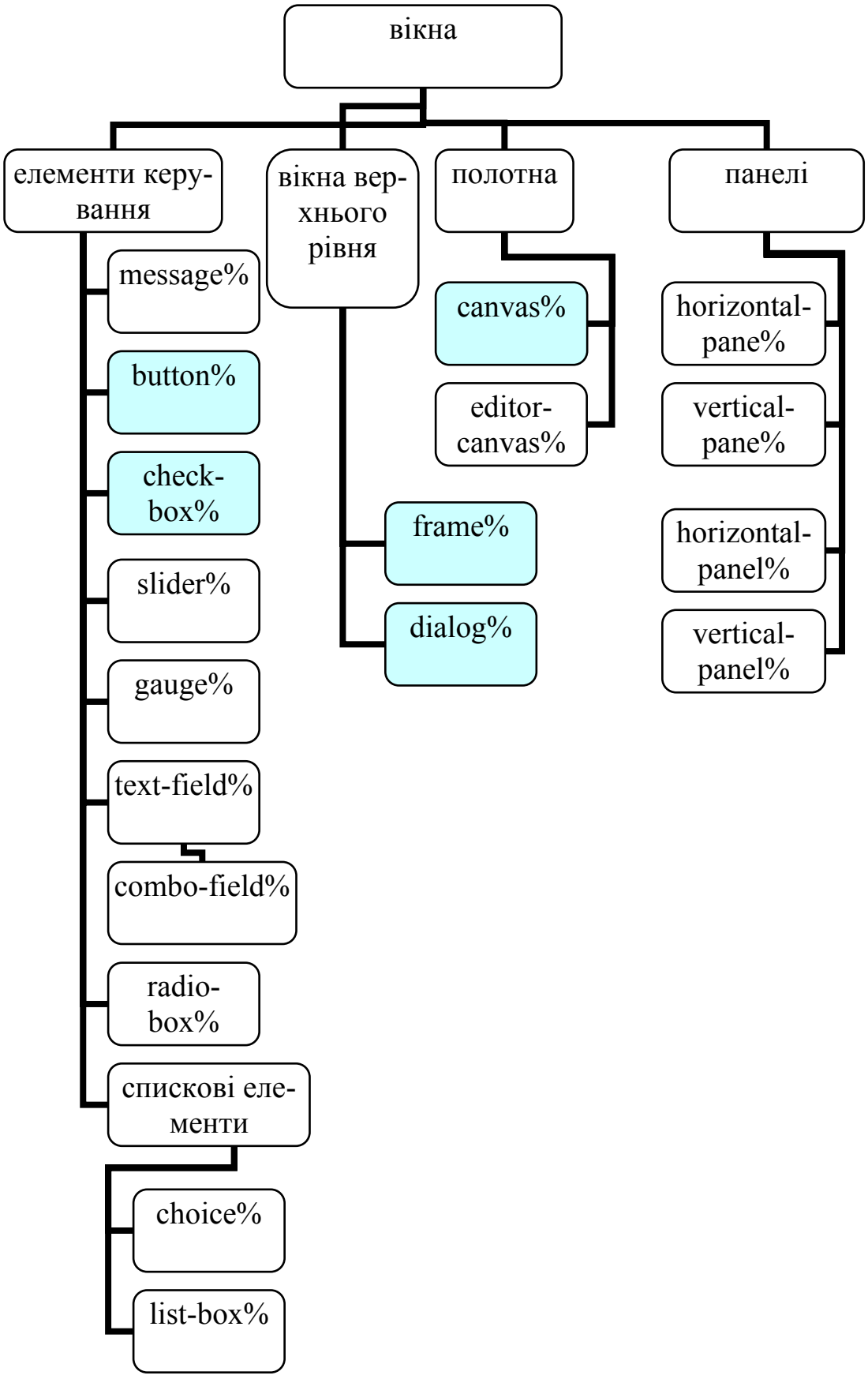


Рис. 8. Ієрархія віконних класів

Повідомлення для рамок:

Загальна форма	Дія, результат
(send об'єкт-рамка iconize iconize?)	згортання або розгортання вікна в залежності від значення iconize? (логічне значення)
(send об'єкт-рамка is-iconized?)	якщо вікно іконізоване – #t (згорнуте на панель задач) і #f – інакше
(send об'єкт-рамка maximize maximize?)	розгортає вікно до максимальних розмірів або згортає залежно від значення maximize? (логічне значення)
(send об'єкт-рамка is-maximized?)	якщо вікно максимально розгорнуте – #t і #f – інакше

Наприклад:

```
; створимо наступне вікно-рамку:
; заголовок – Приклад;
; розмір – 200x100;
; координати верхнього лівого кута – (100, 150);
; мінімальна ширина – 100;
; мінімальна висота – 30;
; стиль – без системного меню та металевий
; вирівнювання – зліва (по горизонталі),
; зверху (по вертикалі)
  (define Рамка (new frame%
      [label "Приклад вікна-рамки"]
      [width 200]
      [height 100]
      [x 100]
      [y 150]
      [min-width 100]
      [min-height 30]
      [style '(no-system-menu metal)]
      [alignment '(left top)]))
 ; відобразимо Рамку
  (send Рамка show #t)
```

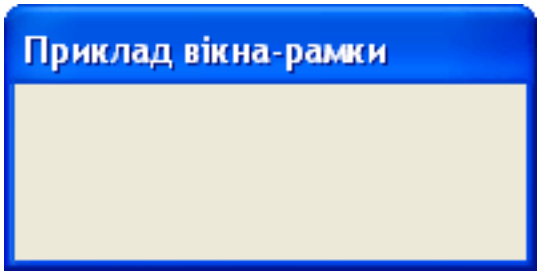


Рис. 9. Створення вікна-рамки

Якщо вікно не відображено, деякі повідомлення залишаються в черзі до відображення вікна.

Клас «діалог» – `dialog%`

Основна відмінність вікна-діалогу від вікна-рамки полягає в тому, що вікно-діалог є модальним: доки воно відображається, всі інші вікна верхнього рівня не ввімкнені.

Поле	**Пояснення**	**Можливі значення # значення за замовчуванням**
обов’язкові поля		
label	заголовок вікна	рядок
необов’язкові поля		
parent	батьківський об’єкт	об’єкт класу `frame%` або `dialog%` # `#f`
width	ширина	ціле число (0; 10000) # `#f`
height	висота	ціле число (0; 10000) # `#f`
x	x-координата лівого верхнього кута	ціле число (-10000; 10000) # `#f`
y	y-координата лівого верхнього кута	ціле число (-10000; 10000) # `#f`
style	стиль	список з елементів: `no-resize-border` – неможливо змінити розміри вікна `no-caption` –без заголовку # `null`
enabled	«ввімкненість»	логічний # `#t`
border	стиль границі	ціле число (0; 1000) # 0
alignment	вирівнювання тексту	список, утворений по 1 елементу з підсписків `'(left center right) '(top`

Поле	Пояснення	Можливі значення # значення за замовчуванням
		center bottom) # '(center top)
min-width	мінімальна ширина	ціле число (0; 10000) # мінімальна графічна ширина
min-height	мінімальна висота	ціле число (0; 10000) # мінімальна графічна висота
stretchable-widtht	здатність розтягуватись у ширину	логічний # #t
stretchable-height	здатність розтягуватись у висоту	логічний # #t

Наприклад:

```
; створимо наступне вікно-діалог:
; заголовок – Приклад вікна-діалогу
; розмір – 200x100;
; координати верхнього лівого кута – (100, 150);
  (define Діалог (new dialog%
      [label "Приклад вікна-діалогу"]
      [width 400]
      [height 100]
      [x 100]
      [y 150]))
; відобразимо вікно – пошлемо об'єкту Діалог
; повідомлення show з параметром #t
  (send Діалог show #t)
```

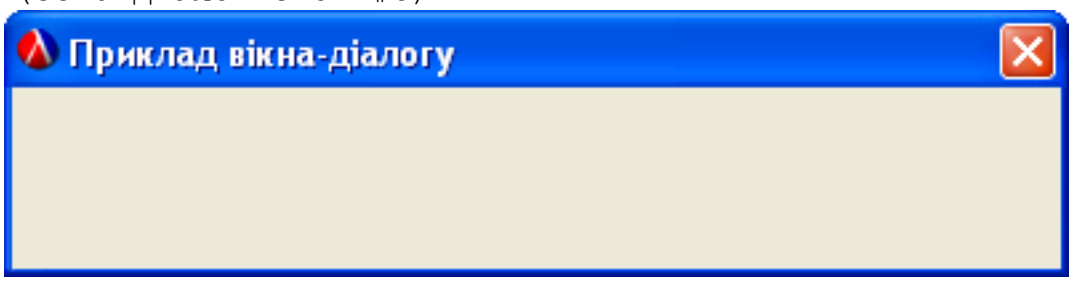


Рис. 10. Створення вікна-діалогу

Як бачимо, у об'єктів класів класу рамка та діалог багато однакових полів. Те ж саме стосується і повідомлень, оскільки об'єкти цих класів належать до вікон верхнього рівня (ВВР)

Повідомлення для ВВР:

Загальна форма	Дія, результат
(send об'єкт-ВВР center [параметр])	центрування вікна на екрані або ж на батьківському об'єкті; параметр: 'horizontal –по-горизонталі 'vertical – по-вертикалі 'both – і по-горизонталі, і по-вертикалі
(send об'єкт-ВВР move x y)	розміщення вікна в заданій позиції на екрані; параметри x і y: x – ціле число (-10000; 10000) y – ціле число (-10000; 10000)
(send об'єкт-ВВР on-exit)	закриття вікна
(send об'єкт-ВВР on-activate active?)	активація або деактивація вікна
(send об'єкт-ВВР resize ширина висота)	встановлення вікну розмірів висота ширина; висота – ціле число (0; 10000) ширина – ціле число (0; 10000)
(send об'єкт-ВВР show відображеність?)	відображення / приховування вікна залежно від параметру відображеність? якщо вікно вже було відображено – розміщення над іншими ВВР; якщо вікно згорнуте (тільки для рамок) – розгортання

Клас «кнопка» – `button%`

Поле	Пояснення	Можливі значення # значення за замовчуванням
обов'язкові поля		
label	заголовок	рядок або об'єкт класу `bitmap%`
parent	батьківський об'єкт	об'єкт класу `frame%`, `dialog%`, `panel%` або `pane%`
необов'язкові поля		
callback	функція оберненого виклику	
style	стиль	список з елементів: `border` – з підсвічуванням гра-

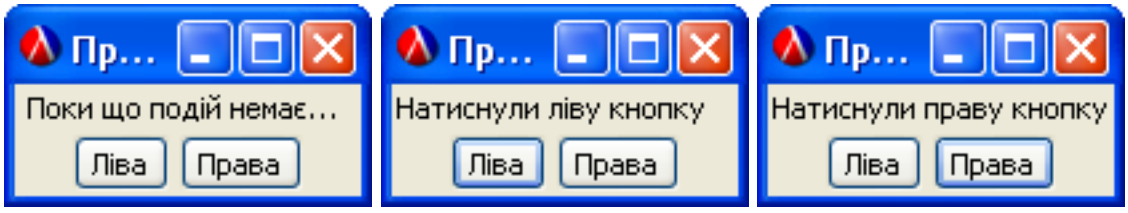
Поле	Пояснення	Можливі значення # значення за замовчуванням
		ниці `deleted` – видалений # `null`
font	шрифт	об’єкт класу `font%` # `normal-control-font`
enabled	ввімкненість	логічний # `#t`
vert-margin	відстань по вертикалі до границі	ціле число (0; 1000) # 2
horiz-margin	відстань по горизонталі до границі	ціле число (0; 1000) # 2
min-width	мінімальна ширина	ціле число (0; 10000) # мінімальна графічна ширина
min-height	мінімальна висота	ціле число (0; 10000) # мінімальна графічна висота
stretchable-widtht	здатність розтягуватись у ширину	логічний # `#f`
stretchable-height	здатність розтягуватись у висоту (якщо (memq 'multiple style))	логічний # `#f`

Коли користувач натискає кнопку, кожного разу викликається функція оберненого виклику, яка передбачена в якості аргументу при ініціалізації кнопки.

Наприклад: створимо рамку, на якій розмістимо повідомлення і 2 кнопки з написами «Ліва» і «Права». При натисненні кнопки «Ліва» на повідомленні з’являється напис «Натиснули ліву кнопку», а при натисненні кнопки «Права» – «Натиснули праву кнопку»:

```
; створимо рамку як об’єкт класу frame%
  (define Рамка (new frame%
      [label "Приклад"]))

; створимо повідомлення на рамці
  (define Повідомлення (new message%
      [parent Рамка]
```

```
      [label " Поки що подій немає...  "]))

; створимо горизонтальну панель, на якій
; розмістимо кнопки з заголовками "Ліва" та "Права"
  (define Панель (new horizontal-panel%
      [parent Рамка]
      [alignment '(center center)]))
  (new button% [parent Панель]
      [label "Ліва"]
      [callback (lambda (button event)
        (send Повідомлення set-label "Натиснули ліву
кнопку"))])
  (new button% [parent Панель]
      [label "Права"]
      [callback (lambda (button event)
        (send Повідомлення set-label "Натиснули праву
кнопку"))])
; відобразимо рамку
  (send Рамка show #t)
```



а) б) в)

а) початковий вигляд

б) після натиснення кнопки «Ліва»

в) після натиснення кнопки «Права»

Рис. 11. Використання кнопок

Клас «текстове поле» – `text-field%`

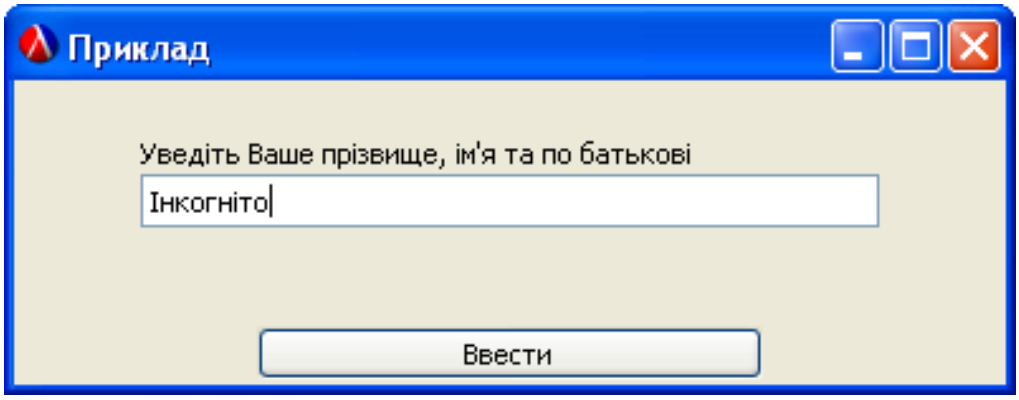
Текстовове поле призначене для введення та подальшого опрацювання повідомлень.

Поле	**Пояснення**	**Можливі значення # значення за замовчуванням**
обов'язкові поля		
label	заголовок	рядок або логічний
parent	батьківський об'єкт	об'єкт класу `frame%`, `dialog%`, `panel%` або `pane%`

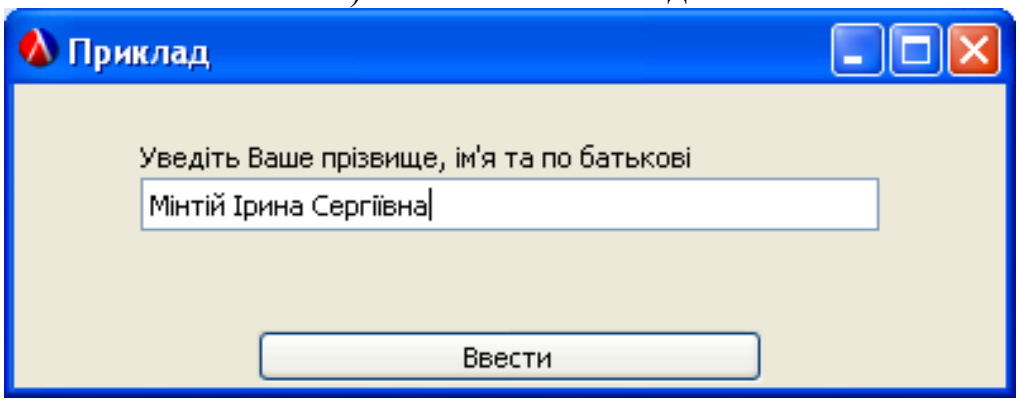
Поле	Пояснення	Можливі значення # значення за замовчуванням
необов’язкові поля		
callback	функція оберненого виклику	
init-value	початкове значення	рядок # «»
style	стиль	список з елементів: `single` – однорядковий `multiple` – багаторядковий `hscroll` – з полосами прокручування (тільки для `multiple`) `password` – з паролем `vertical-label` – вертикальний `horizontal-label` – горизонтальний `deleted` – видалений # '(`single`)
font	шрифт	об’єкт типу `font%` # normal-control-font
enabled	ввімкненість	логічний # `#t`
vert-margin	відстань по вертикалі до границі	ціле число (0; 1000) # 2
horiz-margin	відстань по горизонталі до границі	ціле число (0; 1000) # 2
min-width	мінімальна ширина	ціле число (0; 10000) # мінімальна графічна ширина
min-height	мінімальна висота	ціле число (0; 10000) # мінімальна графічна висота
stretchable-widtht	здатність розтягуватись у ширину	логічний # `#t`
stretchable-height	здатність розтягуватись у висоту (якщо `(memq 'multiple style)`)	логічний # `#t`

Повідомлення для текстових полів:

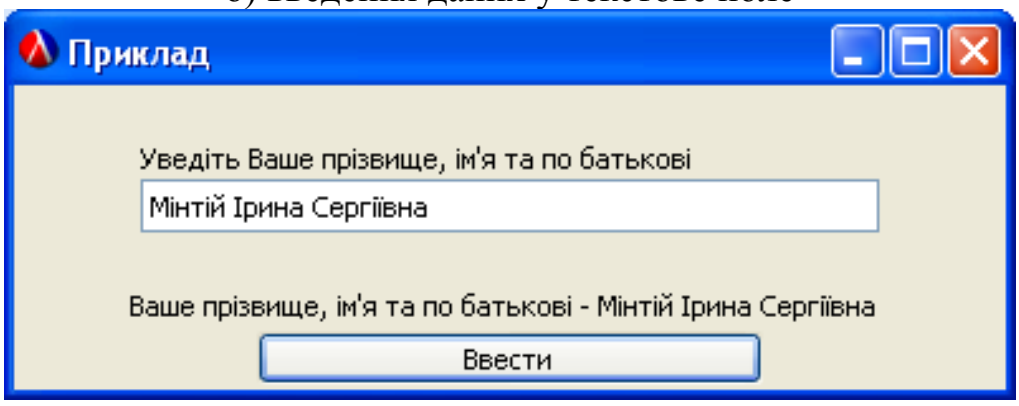
Загальна форма	Дія, результат
`(send об'єкт-текстове-поле get-value)`	рядок-значення текстового поля
`(send об'єкт-текстове-поле set-value рядок)`	надання текстовому полю значення рядок



а) початковий вигляд



б) введення даних у текстове поле



в) відображення введених даних

Рис. 12. Використання текстового поля

Наприклад:

```
; створимо рамку 400 x 100
  (define Рамка (new frame%
      [label "Приклад"]
      [width 400]
      [height 100]))
; створимо на рамці текстове поле
; заголовок поля – Уведіть Ваше ім'я
; початкове значення – Інкогніто
; при кожному наступному введенні тексту
; початкове значення знищується
; стиль текстового поля – однорядковий, вертикальний
; відстань до країв рамки до текстового поля
; 20 зверху та знизу, 50 – зліва та справа
 ; мінімальні розміри текстового поля – 10 x 20
  (define Поле (new text-field%
    [label "Уведіть Ваше прізвище, ім'я та по батькові"]
    [parent Рамка]
    [callback (lambda (text-field event)
                (when (= кількість 0)
                  (let* ((рядок (send Поле get-value))
                         (довжина (string-length ря-
док)))
                    (send text-field set-value
                        (substring рядок (- довжина 1)
                                   довжина)))
                    (set! кількість 1)))]
      [init-value "Інкогніто"]
      [style '(single vertical-label)]
      [vert-margin 20]
      [horiz-margin 50]
      [min-width 10]
      [min-height 20]))
; створимо на рамці напис повідомлення
; спочатку виокремимо на повідомленні
; місце під 50 знаків
  (define Повідомлення (new message%
      [parent Рамка]
      [label (make-string 100 #\ )]))
; створимо на рамці кнопку Прийняти
; по натисненні кнопки у заголовку повідомлення
; відображаються дані, введені користувачем у полі
  (define Прийняти (new button%
```

```
      [parent Рамка]
      [label "Ввести"]
      [min-width 200]
      [callback (lambda (button event)
                  (begin (send Повідомлення set-label
  (string-append "Ваше прізвище, ім'я та по батькові - "
                 (send Поле get-value)))
                         (set! кількість 0)))]))
  (define кількість 0)
; змінна кількість рахує,
; скільки разів введено дані від користувача
; відобразимо рамку
  (send Рамка show #t)
```

Клас «кнопка-прапорець» – `check-box%`

Щоразу, коли користувач натискає кнопку-прапорець, на ній встановлюється / знімається прапорець і викликається функція оберненого виклику, передбачена в якості аргументу при створенні.

Поле	**Пояснення**	**Можливі значення # значення за замовчуванням**
обов’язкові поля		
label	заголовок	рядок або об’єкт класу `bitmap%`
parent	батьківський об’єкт	об’єкт класу `frame%`, `dialog%`, `panel%` або `pane%`
необов’язкові поля		
callback	функція оберненого виклику	
style	стиль	список з елементів: `deleted` – видалений # `null`
value	значення	логічний # `#f`
font	шрифт	об’єкт класу `font%` # `normal-control-font`
enabled	ввімкненість	логічний # `#t`
vert-margin	відстань по вертикалі до границі	ціле число (0; 1000) # 2
horiz-margin	відстань по гозизо-	ціле число (0; 1000) # 2

Поле	Пояснення	Можливі значення # значення за замовчуванням
	нталі до границі	
min-width	мінімальна ширина	ціле число (0; 10000) # мінімальна графічна ширина
min-height	мінімальна висота	ціле число (0; 10000) # мінімальна графічна висота
stretchable-widtht	здатність розтягуватись у ширину	логічний # `#f`
stretchable-height	здатність розтягуватись у висоту (якщо (memq 'multiple style))	логічний # `#f`

Повідомлення для кнопок-прапорців:

Загальна форма	Дія, результат
`(send об'єкт-кнопка-прапорець get-value)`	якщо встановлено прапорець – `#t`, інакше – `#f`
`(send об'єкт-кнопка-прапорець set-value положення)`	встановлення/зняття прапорця, в залежності від параметру положення (логічний)

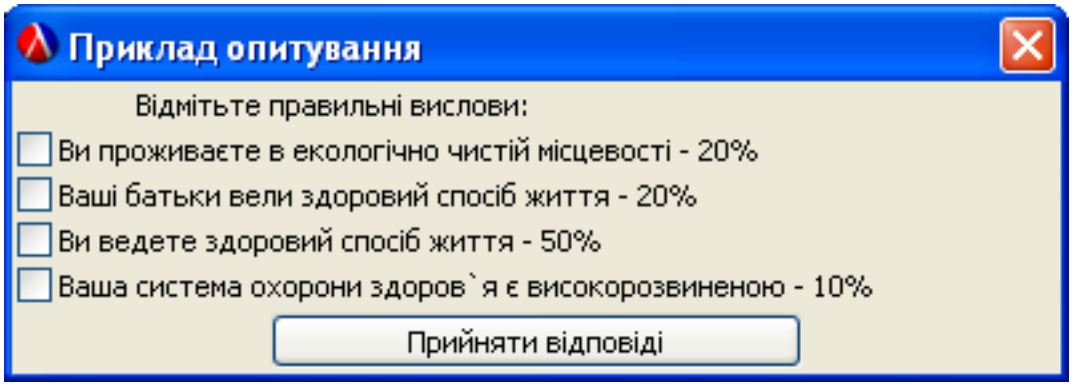
Наприклад:

```
; створимо діалогове вікно,
; на якому розмістимо інші об'єкти
; вирівнювання всіх об'єктів,
; для яких воно буде батьківським об'єктом,
; - по центру по-вертикалі та по-горизонталі
; об'єкти на вікні Діалог
; будуть розташовані по вертикалі у тій послідовності,
; в якій вони будуть створені
  (define Діалог (new dialog%
      [label "Приклад опитування"]
      [width 400]
      [height 100]
      [x 100]
      [y 150]
```

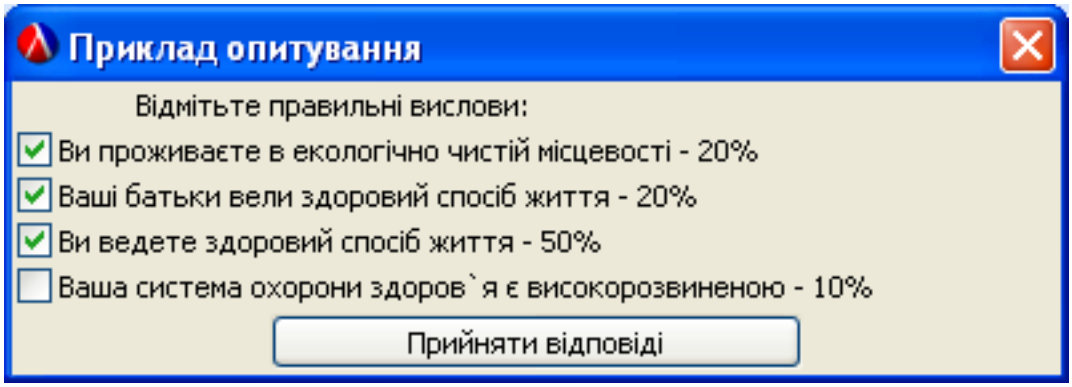
```
      [alignment '(center center)]))

; створимо на діалозі напис
; спочатку виокремимо на напису місце під 100 знаків
  (define Повідомлення (new message%
      [parent Діалог]
      [label (make-string 100 #\ )]))

; пошлемо об’єкту Повідомлення повідомлення set-label
; з параметром "Відмітьте правильні вислови:"
  (send Повідомлення set-label "Відмітьте правильні ви-
слови:")

; створимо на діалозі вертикальну панель
;- на ній розмістимо всі кнопки-прапорці
; вирівнювання об’єктів на панелі - зліва по-горизонталі
; та по-центру по-вертикалі
  (define Панель (new vertical-panel%
      [parent Діалог]
      [alignment '(left center)]))

; кнопки-прапорці розміщуємо на панелі,
; а не на діалозі для того,
; щоб вони були вирівняні зліва
; якщо кнопка-прапорець була натиснута
; непарну кількість раз:
; - збільшуємо на 1 кількість разів натиснення кнопки
; та збільшуємо бали
; інакше (тобто, якщо користувач зняв позначку) –
; зменшуємо бали
  (define Питання1 (new check-box%
       [label "Ви проживаєте в екологічно чистій
місцевості - 20%"]
      [parent Панель]
      [callback (lambda (check-box event)
        (if (odd? кількість1)
            (begin
              (set! кількість1 (add1 кількість1))
              (set! бали (+ бали 20)))
            (set! бали (- бали 20))))]))

  (define Питання2 (new check-box%
      [label "Ваші батьки вели здоровий спосіб життя -
```

```
20%"]
      [parent Панель]
      [callback (lambda (check-box event)
        (if (odd? кількість2)
            (begin
              (set! кількість2 (add1 кількість2))
              (set! бали (+ бали 20)))
            (set! бали (- бали 20)))))]))

  (define Питання3 (new check-box%
      [label "Ви ведете здоровий спосіб життя - 50%"]
      [parent Панель]
      [callback (lambda (check-box event)
        (if (odd? кількість3)
            (begin
              (set! кількість3 (add1 кількість3))
              (set! бали (+ бали 50)))
            (set! бали (- бали 50)))))]))

  (define Питання4 (new check-box%
      [label "Ваша система охорони здоров`я є високороз-
виненою - 10%"]
      [parent Панель]
      [callback (lambda (check-box event)
        (if (odd? кількість4)
            (begin
              (set! кількість4 (add1 кількість4))
              (set! бали (+ бали 10)))
            (set! бали (- бали 10)))))]))

; останньою на діалозі створимо кнопку Кінець
; спочатку її заголовок - "Прийняти відповіді"
; якщо кнопка натиснута вперше (кількість = 1):
; - зникають усі кнопки-прапорці
; - кількість натискань кнопки  - 2
; - її назва змінюється на "Вихід"
; - напис на об’єкті Повідомлення
; - відомості про шанси на довголіття
; якщо кнопка Кінець натиснута двічі (кількість = 2):
; - діалогу посилається повідомлення on-exit
  (define Кінець (new button%
      [parent Діалог]
      [label "Прийняти відповіді"]
```

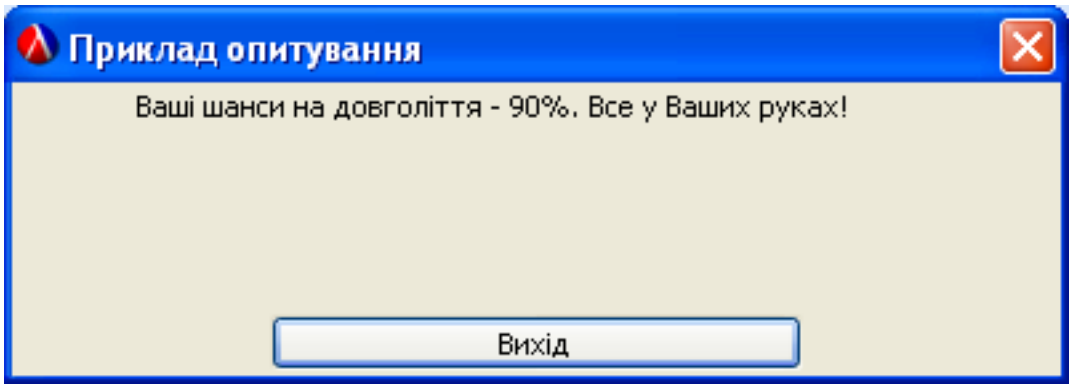
```
      [min-width 200]
      [callback (lambda (button event)
                  (if (= кількість 1)
                      (begin
                        (set! кількість 2)
                        (send Питання1 show #f)
                        (send Питання2 show #f)
                        (send Питання3 show #f)
                        (send Питання4 show #f)
                        (send Кінець set-label "Вихід")
                        (send Повідомлення set-label
            (string-append "Ваші шанси на довголіття - "
                           (number->string бали)
                           "%. Все у Ваших руках!")))
                      (send Діалог on-exit)))]))

; спочатку сума балів 0
 (define бали 0)
; змінна кількість рахує,
; скільки разів натиснута кнопка Кінець
  (define кількість 1)
; змінна кількістьі рахує,
; скільки разів натиснута кнопка-прапорець Питанняі
; (і - від 1 до 4)
  (define кількість1 1)

  (define кількість2 1)

  (define кількість3 1)

  (define кількість4 1)

; відобразимо Діалог
  (send Діалог show #t)
```



а) початковий вигляд



б) вибір вірних висловів



в) показ результатів

Рис. 13. Використання кнопок-прапорців

Повідомлення для вікон:

Загальна форма	**Дія, результат**
`(send об'єкт-вікно enable ввімкненість?)`	вмикання / вимикання вікна залежно від параметру ввімкненість?; коли вікно вимкнене повідомлен-

Загальна форма	Дія, результат
	ня для нього ігноруються (це стосується і всіх об'єктів, для яких вікно є батьківським)
(send об'єкт-вікно focus)	вікну надається фокус введення
(send об'єкт-вікно get-cursor)	вікну надається курсор
(send об'єкт-вікно get-height)	ціле число (0; 10000) – висота вікна
(send об'єкт-вікно get-label)	рядок – заголовок вікна
(send об'єкт-вікно get-size)	розміри вікна
(send об'єкт-вікно get-x)	ціле число (-10000 10000) – позиція (по вісі x) лівого краю на екрані або на батьківському об'єкті
(send об'єкт-вікно get-x)	ціле число (-10000 10000) – позиція (по вісі y) верхнього краю на екрані або на батьківському об'єкті
(send об'єкт-вікно is-enabled?)	#t – якщо ввімкнене вікно і всі його батьківські об'єкти, інакше – #f
(send об'єкт-вікно is-shown?)	#t – якщо показане вікно і всі його батьківські об'єкти, інакше – #f
(send об'єкт-вікно on-focus надати?)	надходить, коли вікно отримує або втрачає фокус введення (#t – якщо отримує і #f – якщо втрачає)
(send об'єкт-вікно on-move x y)	надходить при переміщенні вікна
(send об'єкт-вікно refresh)	перерисування вікна
(send об'єкт-вікно set-label рядок)	надання заголовку вікна значення рядок
(send об'єкт-вікно get-label)	рядок – заголовок вікна

Клас «полотно» – `canvas%`

Поле	Пояснення	Можливі значення # значення за замовчуванням
обов'язкові поля		
parent	батьківський об'єкт	об'єкт класу `frame%`, `dialog%`, `panel%` або `pane%`
необов'язкові поля		
style	стиль	список з елементів: `'border` – тонка границя `'vscroll` – з вертикальною полосою прокручування `'hscroll` – з горизонтальною полосою прокручування `'transparent` – прозорий `'deleted` – видалений # `null`
paint-callback	функція оберненого виклику	
label	заголовок	рядок # `#f`
gl-config		
enabled	ввімкненість	логічний # `#t`
vert-margin	відстань по вертикалі до границі	ціле число (0; 1000) # 0
horiz-margin	відстань по гозизонталі до границі	ціле число (0; 1000) # 0
min-width	мінімальна ширина	ціле число (0; 10000) # мінімальна графічна ширина
min-height	мінімальна висота	ціле число (0; 10000) # мінімальна графічна висота
stretchable-widtht	здатність розтягуватись у ширину	логічний # `#t`
stretchable-height	здатність розтягуватись у висоту (якщо (memq 'multiple style))	логічний # `#t`

Наприклад:

```
; створимо рамку 300 x 400
  (define Рамка (new frame%
                     [label "Малюнок - \"Посміхнись!\""]
                     [width 300]
                     [height 350]))

; створимо на рамці полотно для малювання
  (define Полотно
    (new canvas% [parent Рамка]
                 [paint-callback
                  (lambda (Полотно dc)
                     (if (= кількість 1)
                         (посмішка dc)
                         (без-посмішки dc)))]))
; визначимо змінну dc - графічний драйвер полотна
  (define dc (send Полотно get-dc))

; створимо кілька олівців та щіток
  (define немає-ліній (make-object pen% "black" 1
'transparent))
  (define немає-заливки (make-object brush% "black"
'transparent))
  (define пурпурова-заливка (make-object brush%
"Magenta" 'solid))
  (define жовта-заливка (make-object brush% "yellow"
'solid))
  (define чорна-заливка (make-object brush% "Black"
'solid))
  (define червона-лінія (make-object pen% "red" 4
'solid))

; визначимо функцію малювання обличчя з посмішкою
  (define (посмішка dc)
; малювання еліпса олівцем немає-ліній
; та щіткою пурпурова-заливка
    (send dc set-pen немає-ліній)
    (send dc set-brush  пурпурова-заливка)
    ; центр кола - (50; 50) радіус 200x200
    (send dc draw-ellipse 50 50 200 200)
; малювання очей тим же олівцем і щіткою жовта-заливка
    (send dc set-brush жовта-заливка)
; очі - прямокутники
```

```
; координати верхнього лівого кута (100; 100)
; (200; 200) - правого
; розмір - 10x10
    (send dc draw-rectangle 100 100 10 10)
    (send dc draw-rectangle 200 100 10 10)
; малювання посмішки олівцем червона-лінія
; та щіткою немає-заливки
    (send dc set-brush немає-заливки)
    (send dc set-pen червона-лінія)
; посмішка – дуга - частина кола, вписаного в прямокутник
; координати верхнього лівого кута - (75; 75)
; правого нижнього - (150; 150)
; дуга малюється від кута (* 5/4 pi) (* -1/4 pi)
; проти годинникової стрілки
; початок відліку - положення годинникової стрілки
; о 3:00
    (let ([-pi (atan 0 -1)])
      (send dc draw-arc 75 75 150 150 (* 5/4 pi) (* -1/4
pi))))

; визначимо функцію малювання обличчя без посмішки
; зміни порівняно з попередньою функцією:
; очі - щітка - чорна-заливка
; рот - лінія
; координати початку лінії (100; 200)
; координати кінця - (200; 200)
  (define (без-посмішки dc)
    (send dc set-pen немає-ліній)
    (send dc set-brush  пурпурова-заливка)
    (send dc draw-ellipse 50 50 200 200)

    (send dc set-brush чорна-заливка)
    (send dc draw-rectangle 100 100 10 10)
    (send dc draw-rectangle 200 100 10 10)

    (send dc set-brush немає-заливки)
    (send dc set-pen червона-лінія)
    (send dc draw-line 100 200 200  200))
; створимо на рамці кнопку
; натискання кнопки змінюватиме малюнок з посмішкою
; на малюнок без посмішки
  (define Кнопка (new button% [parent Рамка]
      [label "Сумувати"]
```

```
      [callback (lambda (button event)
        (if (= кількість 1)
            (begin
              (без-посмішки dc)
              (set! кількість 2)
              (send Кнопка set-label "Посміхатись"))
            (begin
              (посмішка dc)
              (set! кількість 1)
              (send Кнопка set-label "Сумувати")))))]))
; змінна кількість зберігає кількість натискань кнопки
  (define кількість 1)
; відобразимо рамку
  (send Рамка show #t)
```

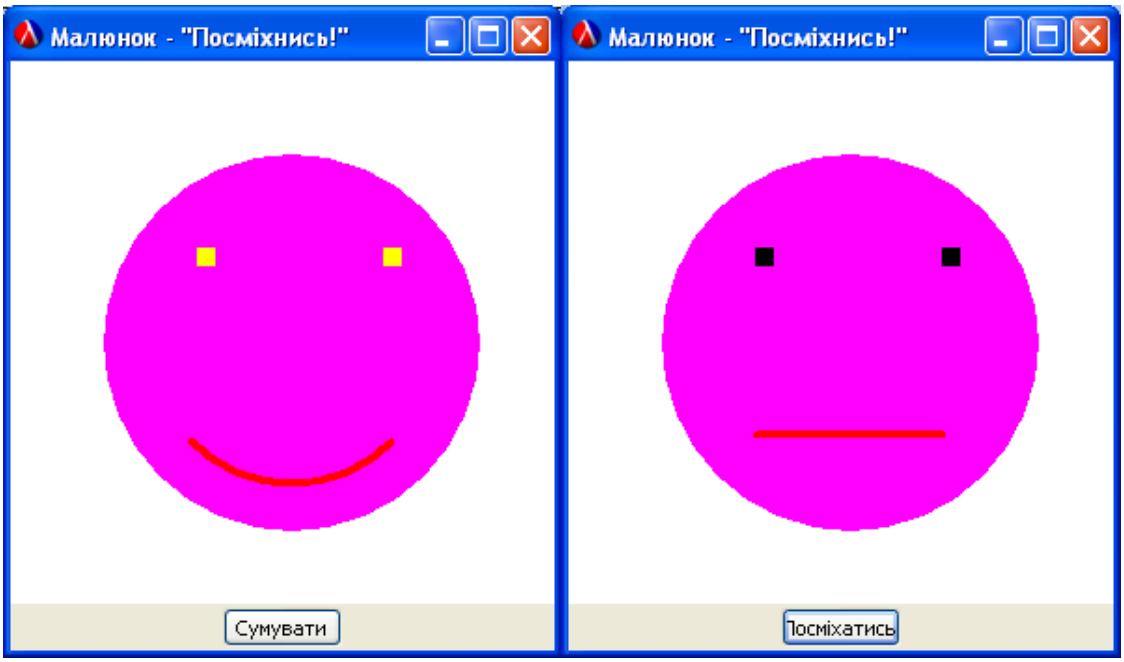


Рис. 14. Використання полотна

Дивись детальніше: Довідка → Допомога → GUI and Graphics Libraries (Графічний інтерфейс користувача і графічні бібліотеки) → GUI: Racket Graphics Toolkit (Графічний інтерфейс користувача: графічний інструментарій Racket)

Проект «Калькулятор»

Створити графічний інтерфейс та реалізувати відповідні функції програми «Калькулятор Плюс» (рис. 15).

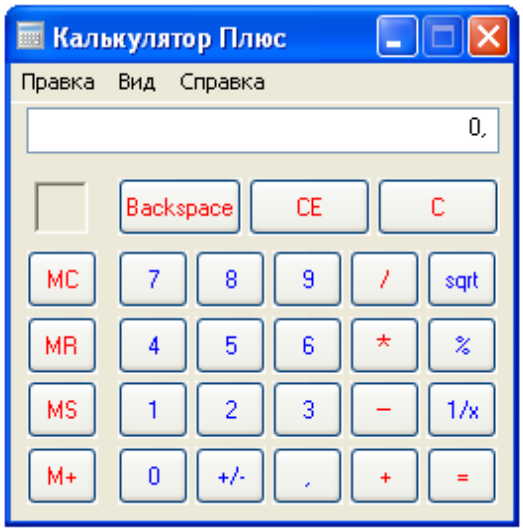


Рис. 15. Програма «Калькулятор Плюс»

Б. Підготовка до роботи та огляд можливостей DrRacket

Б.1. Підготовка DrRacket до роботи

1. Для початку необхідно завантажити файл для інсталяції за посиланням:

http://racket-lang.org/download/

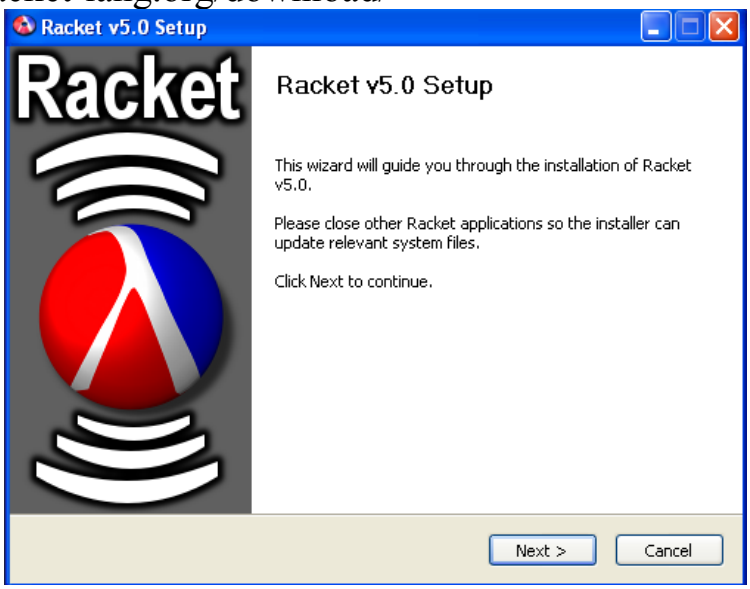


Рис. 16

2. Після запуску завантаженого файлу з’явиться вікно майстра (рис. 16). Натисніть кнопку Next (Далі). Якщо вже були відкриті інші додатки Racket (DrRacket, GRacket, Racket), спершу необхідно їх закрити.

3. В діалоговому вікні, що з’явилось (рис. 17), необхідно вка-

зати папку для інсталяції. За замовчуванням, це C:\Program Files\Racket. Для встановлення в інше місце натисніть кнопку Browse… (Огляд…) та оберіть бажану папку. Попередньо переконайтесь, що на диску, де розташована папка для інсталяції, вільні близько 250 Мб, інакше майстер запропонує обрати інший диск. Для продовження натисніть кнопку Next (Далі).

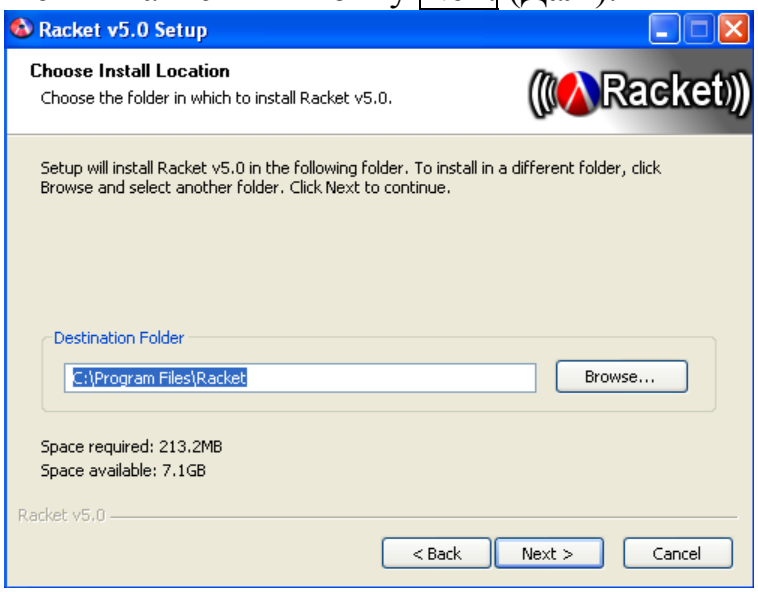


Рис. 17

4. В наступному діалоговому вікні (рис. 18) необхідно вказати ім’я, під яким Racket буде відображатись в меню «Пуск». За замовчуванням, це – Racket, але можна вказати й інше ім’я. Натисніть кнопку Install (Встановити).

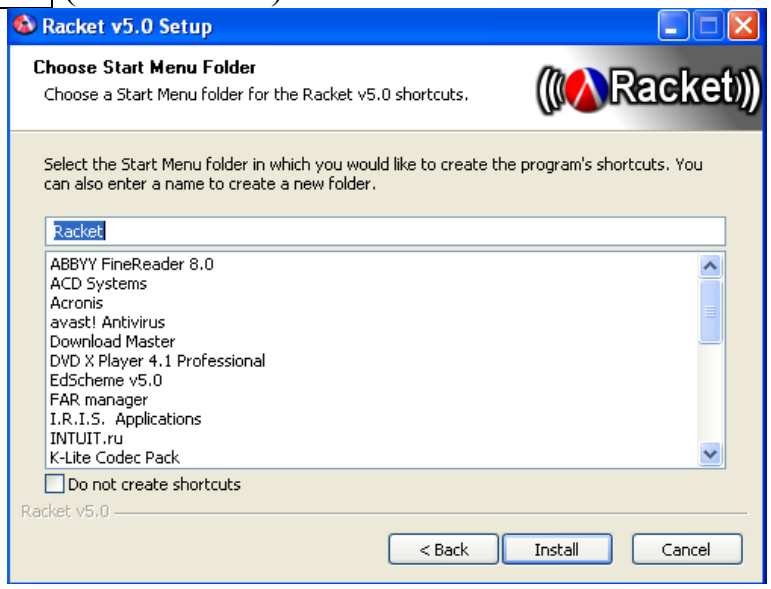


Рис. 18

5. Чекайте, доки не завершиться процес інсталяції (рис. 19):

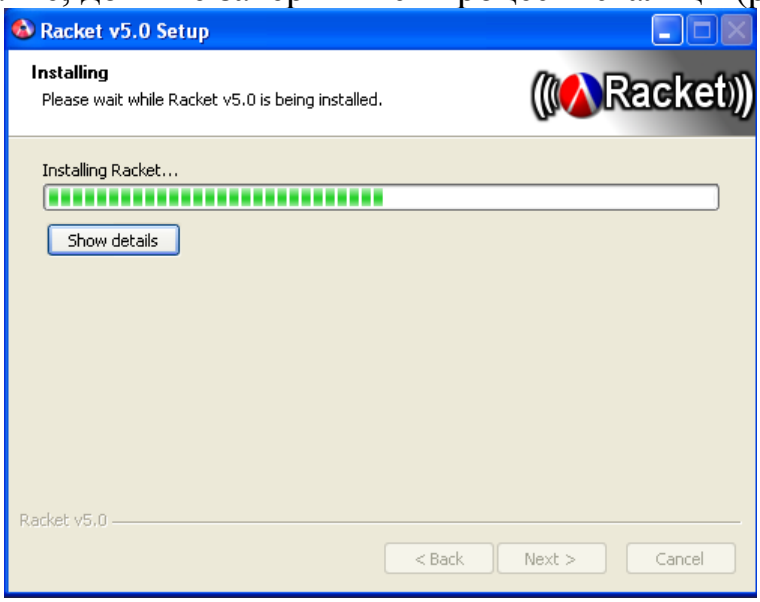


Рис. 19

6. Тепер Racket встановлено. Натисніть кнопку Finish (Закінчити) (рис. 20) для закриття майстра. Якщо перед натисканням кнопки Finish (Закінчити) обрати Run DrRacket (Запустити DrRacket), після завершення роботи майстра буде запущено середовище програмування DrRacket.

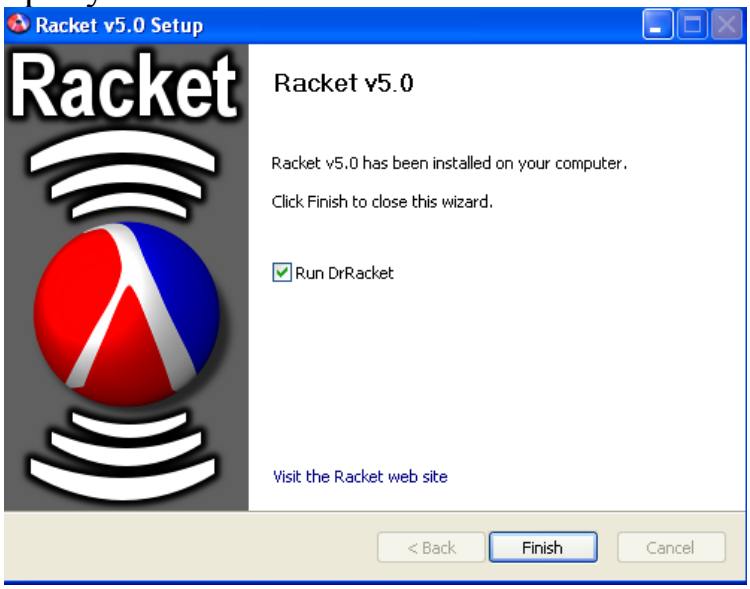


Рис. 20

Б.2. Короткий огляд DrRacket

Основні елементи вікна середовища програмування DrRacket:

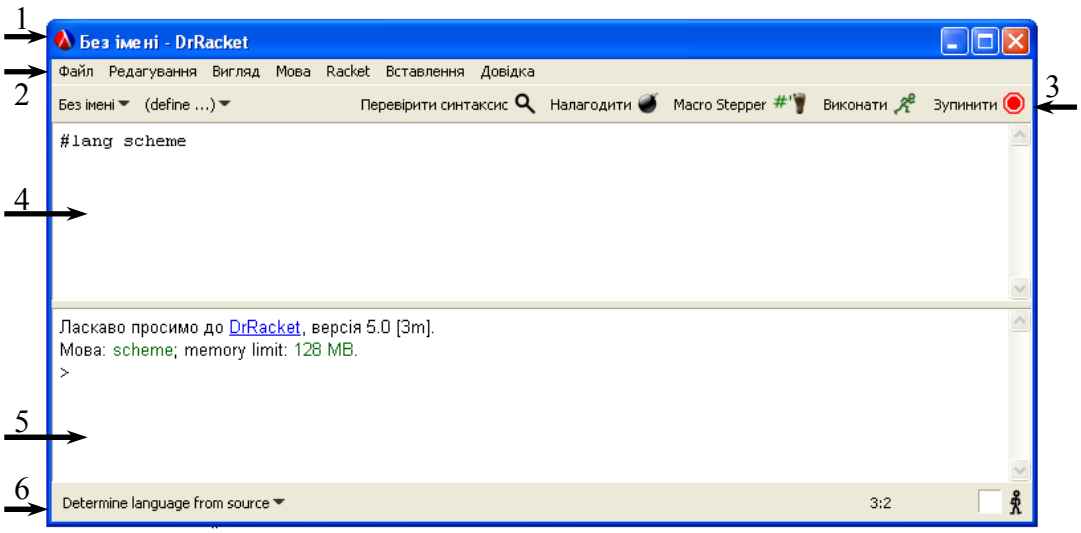


Рис. 21. Основні елементи вікна середовища програмування DrRacket

1 – рядок із заголовком вікна та кнопками керування вікном  (згорнути, розгорнути, закрити);

2 – рядок меню;

3 – панель інструментів;

4 – вікно визначень;

5 – вікно інтерпретатора;

6 – поточна мова програмування.

Призначення вікна визначень

Визначення функцій вводяться у вікні визначень. Для перевірки правильності функцій необхідно натиснути кнопку Виконати. УВАГА! При натисненні кнопки Виконати вікно інтерпретатора очищається, тому попередньо всі потрібні для роботи визначення необхідно скопіювати у вікно визначень або зберегти вміст вікна інтерпретатора.

Призначення вікна інтерпретатора

– Написання простих програм

Перші кроки у DrRacket доцільно робити у вікні інтерпретатора.

Для набору програми необхідно стати в рядку після знаку > та

розпочати введення. Наприклад:

```
> (+ 2 3 5)
```

Після закінчення введення виразу необхідно натиснути клавішу Enter – інтерпретатор виведе результат на екран:

```
10
```

Якщо вираз є складеним, бажано форматувати програму шляхом переходу на новий рядок:

```
> (+ 2
         (* 4 5 6)
         (/ 3 6))
```

Результат буде виведено лише після введення останньої парної дужки, що закривається (під час переміщення курсору по програмі інтерпретатор виділяє кольором вирази, що заключені в парні дужки (рис. 22)):

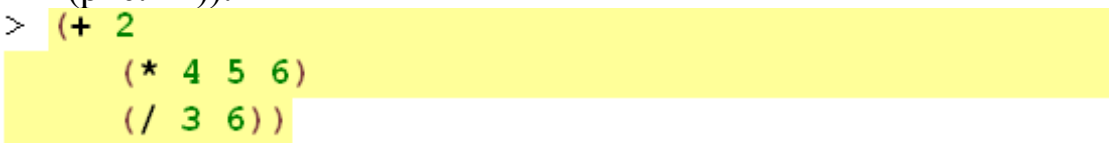


Рис. 22

Якщо при наборі виразу допущена помилка, інтерпретатор виведе повідомлення про це. Наприклад:

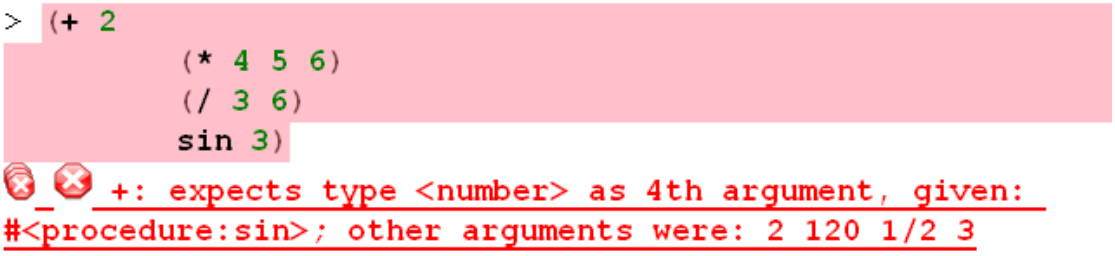


Рис. 23

Для усунення помилки необхідно набрати вираз заново або виділити вираз з помилкою мишею та натиснути клавішу Enter – вираз скопіюється в активний рядок (рядок, де знаходиться курсор) та буде доступний для редагування.

– *Тестування програм*

Після перевірки програми на наявність помилок за допомогою кнопки Виконати бажано протестувати програму. Для цього у вікні інтерпретатора після знаку > необхідно викликати функцію.

Налагодження програми

Покрокове виконання можливе лише для функцій, визначених у вікні визначень. Для покрокового виконання після визначення функції натисніть кнопку Налагодити – з'явиться панель інструментів покрокового виконання та вікна з відображенням змісту стеку і поточних значень змінних. Для переходу до наступного кроку натисніть кнопку Step (Крок). Спостерігайте за змінами вмісту стеку та значень змінних.

Збереження результатів роботи

Збереження вікна визначень: в меню Файл оберіть Зберегти визначення. У діалоговому вікні, що з'явилось, вкажіть ім'я файлу та папку для збереження.

Збереження і вікна визначень, і вікна інтерпретатора: Файл → Зберегти вікна визначень та інтерпретатора.

Завантаження програм

Відкриття файлу: Файл → Відкрити…

Для одночасної роботи з багатьма файлами в одному вікні необхідно створити нову вкладку: Файл → Нова вкладка, а далі в цій вкладці відкрити необхідний файл.

Отримання допомоги

Введення виразу в рядок для пошуку: Довідка → Допомога → введіть вираз в рядок для пошуку або оберіть розділ із запропонованих на екрані.

Допомога для введеного виразу у вікні визначень або вікні інтерпретатора: розмістіть курсор всередині виразу (наприклад, dialog%) → натисніть клавішу F1.

Завершення роботи

Закриття вікна: Файл → Закрити.

Закриття вкладки: Файл → Закрити вкладку.

В. Схематичний світ

№	Адреса	Короткий опис, зміст
Середовища програмування		
1.	http://www.racket-lang.org/	DrRacket
2.	http://inst.eecs.berkeley.edu/	STk-ucb
3.	http://schemers.com/	EdScheme
4.	http://www.gnu.org/software/mit-scheme/	MIT-GNU Scheme

№	Адреса	Короткий опис, зміст
Книги та посібники		
5.	http://www.htdp.org/	Felleisen M. How to Design Programs. An Introduction to Programming and Computing / Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. – Cambridge, Massachusetts : The MIT Press, 2003.
6.	http://mitpress.mit.edu/sicp/	Abelson H. Structure and Interpretation of Computer Programs / Harold Abelson, Gerald Jay Sussman, Julie Sussman. – Cambridge, Massachusetts : The MIT Press, 1996.
7.	http://www.scheme.com/tspl3/	Dybvig R. K. The Scheme Programming Language, 3rd Edition / R. Kent Dybvig. – Cambridge, Massachusetts : The MIT Press, 2003.
8.	http://www.cs.hut.fi/Studies/T-93.210/schemetutorial/	Scheme Tutorial [Electronic resource] / [Catharina Candolin] – Mode of access : http://www.cs.hut.fi/Studies/T-93.210/schemetutorial/
9.	http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html	Teach Yourself Scheme in Fixnum Days [Electronic resource] / [Dorai Sitaram] – Mode of access : http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html
Вибране навчально-методичне забезпечення		
10.	http://schemers.com/	список навчальних закладів, в яких використовується мова програмування

№	Адреса	Короткий опис, зміст
		Scheme
11.	http://www.cs.brandeis.edu /~cs21b/	Курс «Structure and Interpretation of Computer Programs», Університет Брандейса, м. Велтам, штат Массачусетс, США
12.	http://www.math.grin.edu/ ~stone/courses/scheme/	Курс «Fundamentals of computer science I», Гріннелл-коледж, м. Гріннелл, штат Айова, США
13.	http://cs.bilgi.edu.tr/pages/ courses/year_1/comp_149/	Курс «The Fundamentals of Computer Science I», Стамбульський університет, м. Стамбул, Туреччина
14.	http://sicp.csail.mit.edu/Fall-2005/	Курс «Structure and Interpretation of Computer Programs», Массачу́сетський технологі́чний інститут, м. Кембридж, штат Массачусетс, США
15.	http://www.cs.northwestern.edu/ academics/courses/111/	Курс «Fundamentals of Computer Programming», Північно-Західний університет, м. Еванстон, штат Іллінойс, США
16.	http://www.cs.trinity.edu/About/ The_Courses/cs301/	Курс «Great Ideas in Computer Science», Тріністі університет, м. Сан-Антоніо, штат Техас, США
17.	http://www.cs.trinity.edu/ ~meggen/Classes/2322/	Курс «Principles of Functional Languages», Тріністі університет, м. Сан-Антоніо, штат Техас, США
18.	http://www.dccia.ua.es/ dccia/inf/asignaturas/LPP/	Курс «Programming Languages and Paradigms», Університет Аліканте, м. Аліканте, Іспанія

№	Адреса	Короткий опис, зміст
19.	http://www.classes.cs.uchicago.edu/archive/2009/fall/15100-1/	Курс «Introduction to Computer Science I», Чиказький університет, м. Чикаго, штат Іллінойс, США
20.	http://zoo.cs.yale.edu/classes/cs201/	Курс «Introduction to Computer Science», Єльський університет, м. Нью-Хейвен, штат Коннектикут, США
21.	http://www.cs.caltech.edu/courses/cs1/	Курс «Introduction to Computation», Каліфорнійський університет, м. Берклі, Каліфорнія, США
22.	http://webcast.berkeley.edu/course_details_new.php?seriesid=2008-D-26263&semesterid=2008-D	Відеолекції курсу «Structure and Interpretation of Computer Programs». Викладач – Брайан Харві.

Література

1. Абельсон Х. Структура и интерпретация компьютерных программ : пер. с англ. / Х. Абельсон, Д. Д. Сассман, Дж. Сассман. – М. : Добросвет, 2006. – 608 с.
2. Англо-український тлумачний словник з обчислювальної техніки, Інтернету і програмування / [Е. М. Пройдаков, Л. А. Теплицький]. – Вид. 2. – К. : СофтПрес, 2007. – 824 с.
3. Теплицкий И. А. Введение в программирование систем искусственного интеллекта на языке Лисп : Лабораторный практикум / Теплицкий И. А., Семериков С. А. – Кривой Рог : КГПУ, 2004. – 88 с.
4. Филд А. Функциональное программированию : пер. с англ. / А. Филд, П. Харрисон. – М. : Мир, 1993. – 640 с.
5. Хювёнен Э. Мир Лиспа. В 2-х т. Т. 1. : Введение в язык Лисп и функциональное программирование : пер. с финск. / Хювёнен Э., Сеппянен Й. – М. : Мир, 1990. – 447 с.
6. Scheme Tutorial [Electronic resource] / [Catharina Candolin] – Mode of access :
http://www.cs.hut.fi/Studies/T-93.210/schemetutorial/
7. Schools Using Scheme [Electronic resource] – Mode of access :
http://www.schemers.com/schools.html
8. Sussman G. J. Scheme: An Interpreter for Extended Lambda Calculus // MIT Artificial Intelligence Memo 349. – December 1975. – Cambridge : MIT, 1975.
9. Teach Yourself Scheme in Fixnum Days [Electronic resource] / [Dorai Sitaram] – Mode of access :
http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html
10. Times Higher Education-QS World University Rankings 2009. [Electronic resource] – Mode of access :
http://www.timeshighereducation.co.uk/Rankings2009-Top200.html

Мінтій Ірина Сергіївна

Схематичне програмування

Початки програмування: функціональний підхід

Науковий редактор
дійсний член НАПН України
М.І. Жалдак

Підп. до друку 04.10.2010
Папір офсетний №1
Ум. друк. арк. 8,5

Формат 80×84 1/16
Зам. №1-0810
Тираж 100 прим.

Жовтнева районна друкарня
50014, м. Кривий Ріг, вул. Електрична, 5
Тел. (0564) 407-29-02
E-mail: irina.mintiy@gmail.com