

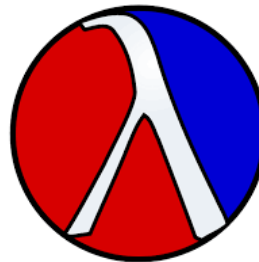
Функціональне програмування

*Лектор Ковалюк Тетяна Володимирівна
к.т.н., доцент*

tkovalyuk@ukr.net
<https://github.com/tkovalyuk/functional-program>

Лекція 4

Вирази, стандартні процедури та процедури вищого порядку в SCHEME/Lisp/...



План лекції 4

1. Стандартні форми в R5RS Scheme
2. Бібліотечні форми в Scheme
3. Стандартні (вбудовані) процедури в Scheme
 - 3.1. Предикати
 - 3.2. Процедури для обробки чисел
 - 3.3. Процедури перетворення
4. Умовні вирази
5. Процедури вищого порядку
 - 5.1. Процедури як аргументи
 - 5.2. Процедури як значення, що повертаються.

Стандартні форми в R5RS Scheme

Призначення	Форми
Визначення	<code>define</code>
Конструкції прив'язки	<code>lambda</code> , <code>do</code> , <code>let</code> , <code>let*</code> , <code>letrec</code>
Умовні обчислення	<code>if</code> , <code>cond</code> , <code>case</code> , <code>and</code> , <code>or</code>
Послідовні обчислення	<code>begin</code>
Ітерації	<code>lambda</code> , <code>do</code> , <code>named let</code>
Розширення синтаксиса (макроси)	<code>define-syntax</code> , <code>let-syntax</code> , <code>letrec-syntax</code> , <code>syntax-rules</code> (R5RS), <code>syntax-case</code> (R6RS)
Квотування	<code>quote(')</code> , <code>unquote(,)</code> , <code>quasiquote(`)</code> , <code>unquote-splicing(,@)</code>
Присвоєння	<code>set!</code>
Відкладені обчислення	<code>delay</code>

Бібліотечні форми в Scheme

Призначення	Форми
Конструкції прив'язки	<code>do</code>
Конструкції прив'язки	<code>let, let*, letrec</code>
Умовні обчислення	<code>cond, case</code>
Умовні обчислення	<code>and, or</code>
Ітерації	<code>named let</code>
Відкладені обчислення	<code>delay</code>
Послідовні обчислення	<code>begin</code>

Умовна форма if

Синтаксис

```
(if <перевірка> <наслідок> <альтернативна гілка>)  
    (if <перевірка> <наслідок>)
```

<перевірка ">, " наслідок " і " альтернатива> можуть бути будь-якими виразами

Семантика

1. Спочатку виконується <перевірка>.
2. Якщо перевірка дає істинне значення, виконується <наслідок> і повертається його значення .
3. В іншому випадку виконується <альтернативна гілка> і повертається його значення.
4. Якщо <перевірка> дає помилкове значення і <альтернативна гілка> не вказана, то результат виразу не визначено.

Умовна форма cond

Бібліотечний синтаксис:

(cond <клауза1> <клауза2> ...)

Будь-яка <клауза> має форму:

(<перевірка> <вираз1> ...)

де <перевірка> це довільний вираз. Також, <клауза> може мати форму: (<перевірка> => <вираз>)

Остання <клауза> є <клауза інакше>, яка має форму:

(else <вираз1> <вираз2> ...).

Семантика:

1. Вираз **cond** визначається обчисленням виразів <перевірки> успішної <клаузи> до поки одна з них не визначиться як істинне значення.
2. Коли <перевірка> обчислюється як істинне значення, вирази, що залишилися в своїй <Клаузе> обчислюються в порядку, а результат останнього виразу в <Клаузе> повертається як результат всього виразу **cond**.
3. Якщо обрана <клауза> містить тільки <перевірку> і не містить <Виразів>, то в результаті повертається значення <перевірки>.
4. Якщо обрана <клауза> використовує альтернативну форму =>, то обчислюється <вираз>. Цей вираз має бути процедурою, яка приймає один аргумент. Цей вираз викликається зі значенням <перевірки> і значення, що повертаються цією процедурою, повертаються виразом **cond**.
5. Якщо всі <перевірки> обчислюються із значенням хибності, і немає клаузи **else**, результат умовного виразу не визначено;
6. Якщо є клауза **else**, то <вираз> обчислюються, і повертається значення.

Умовна форма case

Бібліотечний синтаксис:

(case <ключ> <клауза1> <клауза2> ...)

Синтаксис

<Ключем> може бути будь-який вираз. Будь-яка <клауза> повинна мати форму:

((<Елемент даних1>) ...) <вираз1> <вираз2> ...),

де кожен <елемент даних> є зовнішнім поданням деякого об'єкту. Всі <елементи даних> повинні бути різні. Остання <клауза> може бути «клаузою else», яка має форму.

(else <вираз1> <вираз2> ...).

Семантика:

1. Вираз **case** обчислюється так. Обчислюється <ключ>, його результат порівнюється з кожним <елементом даних>.
2. Якщо результат обчислення <ключа> збігається з <елементом даних>, то вираз у відповідній <Клаузе> обчислюється зліва направо і результат останнього виразу в <Клаузе> повертається, як результат виразу case.
3. Якщо результат обчислення <ключа> відмінний від кожного <елемента даних>, то якщо є клауза case, його вирази обчислюються і результат останнього результату є результатом виразу case, інакше результат виразу case не визначений

Стандартні форми прив'язки змінних

- ❑ SCHEME— це блочно-структурована мова з вкладеними областями.
 - ❑ Можна оголосити локальні змінні, область видимості яких є блоком коду, і блоки можуть мати всередині блоки з власними локальними змінними.
 - ❑ У SCHEME використовується правило **лексичного обсягу (lexical scope)**. Можна сказати, що Scheme має статичну область видимості, а не динамічну. Коли ви бачите назву змінної в коді, ви можете визначити, до якої змінної вона відноситься, просто подивившись на вихідний код програми.
 - ❑ Програма складається з вкладених блоків коду, а значення назви визначається зв'язуванням змінної, у якій вона використовується.
-
- ❑ **let: let** зв'язує локальні змінні
 - ❑ **let*: let*** прив'язує змінні послідовно, у вкладених областях

Стандартна форма let

Ви можете створювати блоки коду, які мають локальні змінні, використовуючи спеціальну форму **let**.

Наприклад:

```
(let ((x 10)
      (y 20))
  (func x y))
```

- ❑ **Перша частина let** — це речення зв'язування змінних, яке в даному випадку складається з двох підречень `(x 10)` і `(y 20)`. Це означає, що **let** створить змінну з іменем `x`, початкове значення якої дорівнює `10`, і іншу змінну `y`, початкове значення якої дорівнює `20`.
- ❑ Речення зв'язування змінної **let** може містити будь-яку кількість речень, створюючи будь-яку кількість змінних **let**. Кожне підречення дуже схоже на назву та початкове значення частини форми визначення.
- ❑ **Решта let** - це послідовність виразів, яка називається **тілом let**. Вирази просто обчислюються по порядку, а значення **останнього виразу повертається як значення всього виразу let**.
- ❑ У Scheme можна використовувати локальні змінні майже так само, як і в більшості мов. Коли ви **вводите вираз let**, змінні **let** будуть зв'язані та ініціалізовані значеннями. Коли ви **вийдете з виразу let**, ці зв'язки зникнуть.
- ❑ Загалом прив'язки для змінних схеми розміщуються **не в стеку активації, а в купі**. Це дозволяє зберігати прив'язки після того, як процедура, яка їх створює, повернеться.
- ❑ Більшість змінних насправді **знаходяться в регістрах**, коли це має значення, так що згенерований код є швидким

Приклади 28-32 використання стандартних форм в R5RS-R6RS Scheme/Racket

define

```
(define (average x y)
  (/ (+ x y) 2))
(display "ex28 - define\n")
(average 5 10)
```

$$7\frac{1}{2}$$

let set!

```
(let
  (
    (a "My")
    (b " name")
    (c " is")
    (d " Michael")
  )
  (set! d " Lesia") ;d= " Lesia")
(display "Ex 29 let & set!\n")
(display a)
(display b)
(display c)
(display d)
(display "\n")
```

Ex28 let & set!
My name is Lesia

lambda

```
(define hello-world (lambda () "Hello World"))
(hello-world)

(define (f g)
  (g 2))

(display "Ex30 якщо x=2, тоді (x+1)*x=")
(f (lambda (x) (* x (+ x 1))))

"Hello World"
Ex30 якщо x=2, тоді (x+1)*x=6
```

Стандартна форма set!

Синтаксис:

set! <variable> <expression>

<Expression> обчислюється, а отримане значення зберігається в місці, до якого прив'язана <variable>.

<Змінна> повинна бути прив'язана до певної області, що охоплює set! вираз або на найвищому рівні.

Результат set! виразу не визначений.

Приклади 31-32 використання стандартних форм в R5RS-R6RS Scheme/Racket

If cond begin

```
(display "Ex31 - if: ")
(if (< 3 1)
  (begin
    (display "one line")
    (newline)
    (display "two line\n")
  )
  (display "no line\n"))

(display "Ex31 - cond: ")
(cond
  ((> 2 2) "wrong!")
  ((< 2 2) "wrong again!")
  ((= 2 2) "ok")
  (else "there is no other")
)

Ex31 - if: no line
Ex31 - cond: "ok"
```

when для Racket

```
(let loop ((i 0)) ; definition
  (when (< i 4) ; condition
    (display (format "i=~a\n" i)) ; body
    (loop (+ 1 i)) ; next iteration
  )
)
i=0
i=1
i=2
i=3
```

Цикл через рекурсію

```
(display "Ex32 \n")
(define (repeat number)
  (if (> number 0) ; якщо кількість повторів > 0
    (begin (display "hello ") ; виконується дія
            (repeat (- number 1))) ; повтор дії
    (display "\n end loop"))
)
(repeat 3)
Ex32
hello hello hello
end loop
```

```
(define (repeat number function)
  (if (> number 0)
    (begin (function)
            (repeat (- number 1) function)))
  (display "Ex33 lambda in loop\n")
  (repeat 3 (lambda() (display "2022 ")))
)
2022 2022 2022
```

Приклади 34-35 використання стандартних форм в R5RS-R6RS Scheme/Racket

and or not

```
(display "\n ex34 \n")
(not "hot")
(and (odd? 4) ;перевірка на
      ;непарність
      (even? 3)) ;перевірка на
      ;парність
(not #t)      ;=> #f
(not 3)       ;=> #f
(not (list 3)) ;=> #f
(not 'nil)    ;=> #f

(and (= 2 2) (> 2 1)) ;#t
(or (= 2 2) (< 2 1))  ;#t
(and #f #t #t)        ;#f
(or #f #t #f )        ;#t
```

cond case

```
(define (classify x)
  (cond
    ((< x 0) "Negative")
    ((case x((13 42 100) #t)
      (else #f))
     "Special")
    (else "Unknown")))
(display "\n Ex 35 cond case\n" )
(classify 15)
Ex 35 cond case
"Unknown"
```

case

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ==> composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b)) ==> unspecified
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else 'consonant)) ==> consonant
```

Діаграма структури блоків для let

Розглянемо **фрагмент** коду

```
(let ((x 10) ; зовнішня прив'язка x
      (a 20)) ; прив'язка a
  (func x)
  (let ((x (bar)) ; внутрішня прив'язка x
        (b (baz x x))) ; прив'язка b
    (quux x a)
    (quux y b))
  (baz x a) ; відноситься до зовнішнього x (і a)
  (baz x b)) ; невірно?
```

Коли керування надходить до зовнішнього дозволу, обчислюються початкові значення для змінних. У цьому випадку це лише літеральні значення **10** і **20**. Потім пам'ять виділяється для змінних та ініціалізується цими значеннями. Після того, як це зроблено, значення імен **x** і **a** змінюється - тепер вони посилаються на нове сховище для (зв'язків) змінних **let x** і **a** --- і тоді обчислюються основні вирази. Подібним чином, коли керування надходить до внутрішнього **let**, початкові значення обчислюються за допомогою викликів **bar** і **baz**, а потім пам'ять для **x** і **b** виділяється та ініціалізується цими значеннями. Тоді значення імен **x** і **b** змінюються, щоб посилатися на нове зберігання (прив'язки) цих змінних. внутрішня змінна **let x** затінює зовнішню, всередині тіла **let**. Зовнішній **x** більше не видно, тому що є внутрішній. Коли ми виходимо з **let**, прив'язки, введені **let**, «виходять за межі», тобто більше не видимі. (Наприклад, коли ми обчислюємо вираз **(baz x a)** у тілі зовнішнього **let**, **x** посилається на прив'язку, введену зовнішнім **let** --- **x**, введений внутрішнім **let**, більше не видно. Подібним чином у фрагменті коду прикладу **b** в останньому виразі **(baz x b)** не відноситься до внутрішнього зв'язування **let b**. Якщо немає зв'язування **b** у зовнішній області, тоді це буде помилка.

Діаграма структури блоків для let

Розглянемо фрагмент коду

```
(let ((x 10) ; зовнішня прив'язка x
      (a 20)) ; прив'язка a
  (func x)
  (let ((x (bar)) ; внутрішня прив'язка x
        (b (baz x x))) ; прив'язка b
    (quux x a)
    (quux y b))
  (baz x a) ; відноситься до зовнішнього x (і a)
  (baz x b)) ; невірно?
```

```
(let ((x 10) ; bindings of x
      (a 20)) ; and a
  +-----+
  | (foo x)                                     scope of outer x |
  | (let ((x (bar))                             and a
          (b (baz x x))))
  | +-----+
  | | (quux x a)                               scope of inner x |
  | | (quux y b)                               and b           | )
  | +-----+
  | (baz x a)
  | (baz x b)
  +-----+ )
```

Останній виклик **(baz x b)** не стосується змінної **let b** --- його немає в полі, що відповідає цій змінній. Ми також бачимо, що **x** у цьому виразі відноситься до **зовнішнього x**. Поява **x** у викликах **quux** відноситься до внутрішнього **x**, оскільки вони знаходяться всередині його поля, а внутрішні визначення затінюють зовнішні.

Scheme пропонує два варіанти let, які називаються **let*** і **letrec**.

Стандартна форма let*

let є корисним для більшості локальних змінних, але іноді потрібно створити **кілька локальних змінних послідовно**, маючи значення кожної змінної, доступне для обчислення значення наступної змінної. Наприклад, прийнято «деструктурувати» структуру даних, вилучаючи частину структури, потім частину цієї частини і так далі. Ми могли б зробити це, просто вкладаючи вирази, які виділяють частини, але тоді у нас не було б зрозумілих назв для проміжних результатів вкладених виразів. В інших випадках ми можемо захотіти зробити більше ніж одну дію з результатами одного з вкладених виразів, тому **нам потрібно створити змінну, щоб ми могли посилатися на неї в кількох основних виразах**.

```
(let ((a-structure (some-procedure)))  
  (let ((a-substructure (get-some-subpart a-structure)))  
    (let ((a-subsubstructure (get-another-subpart a-substructure)))  
      (func a-substructure))))
```

Scheme надає зручний синтаксис для такого типу вкладених літер; можна записати як один **let***

Стандартна форма let*

```
(let ((a-structure (some-procedure)))  
  (let ((a-substructure (get-some-subpart a-structure)))  
    (let ((a-subsubstructure (get-another-subpart a-substructure)))  
      (func a-substructure))))
```

```
(let* ((a-structure (some-procedure))  
      (a-substructure (get-some-subpart a-structure))  
      (a-subsubstructure (get-another-subpart a-substructure)))  
  (func a-substructure))
```

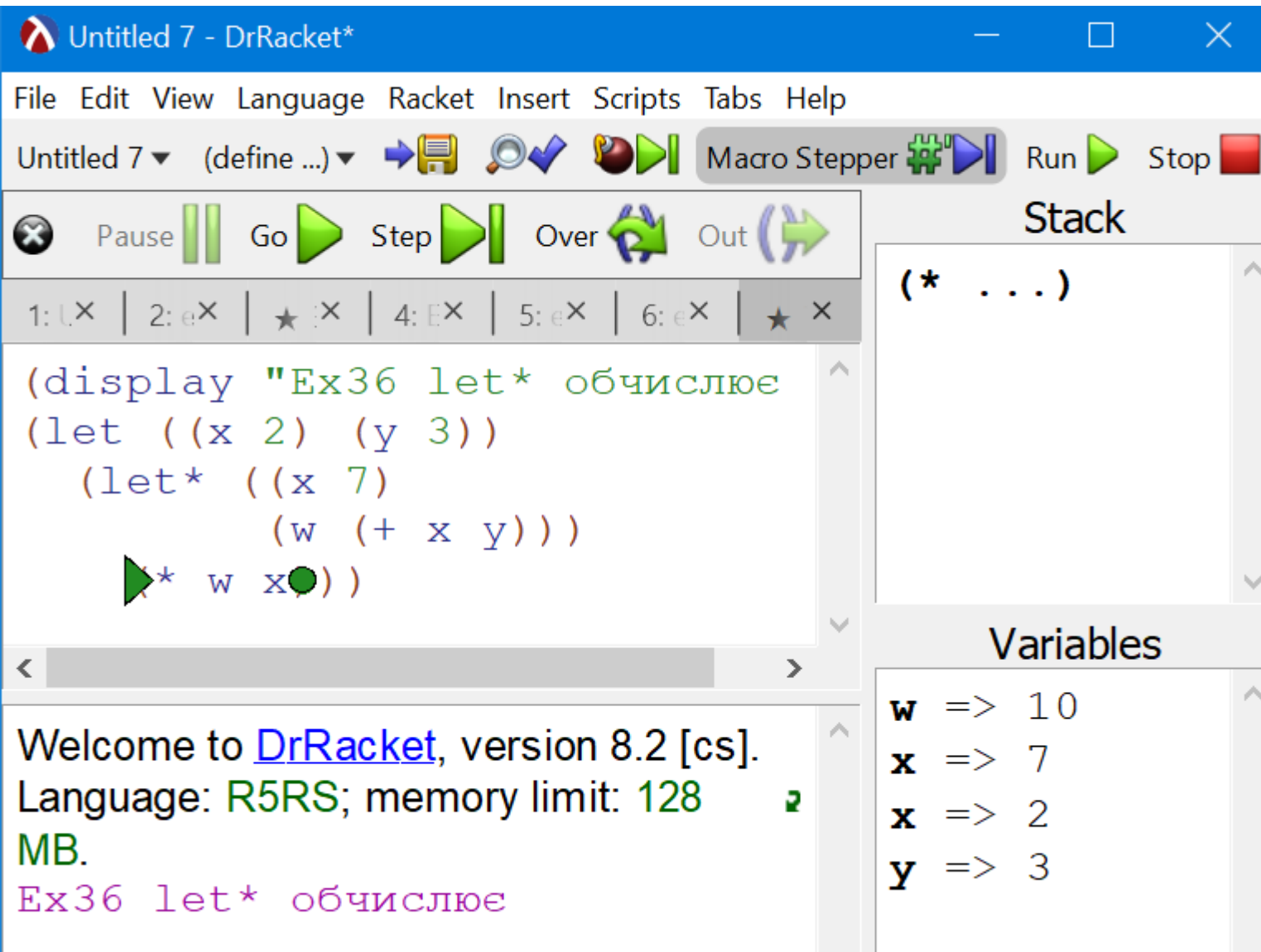
Кожна прив'язка початкового значення знаходиться в області видимості попередньої змінної в **let***.

З вкладеності полів ми бачимо, що прив'язки стають видимими по черзі, тому значення прив'язки можна використовувати для обчислення початкового значення наступної прив'язки.



```
(let* ((a-structure (some-procedure))  
      (a-substructure (get-some-subpart a-structure))  
      (a-subsubstructure (get-another-subpart a-substructure)))  
  (foo a-subsubstructure))
```

Стандартна форма let*



Untitled 7 - DrRacket*

File Edit View Language Racket Insert Scripts Tabs Help

Untitled 7 (define ...) Macro Stepper Run Stop

Pause Go Step Over Out

1: C× | 2: e× | ★ | 4: E× | 5: e× | 6: e× | ★ ×

```
(display "Ex36 let* обчислює")
(let ((x 2) (y 3))
  (let* ((x 7)
         (w (+ x y)))
    (* w x)))
```

Welcome to [DrRacket](#), version 8.2 [cs].
Language: R5RS; memory limit: 128 MB.
Ex36 let* обчислює

Stack

```
(* ...)
```

Variables

```
w => 10
x  => 7
x  => 2
y  => 3
```

```
(display "Ex36 let* обчислює")
(let ((x 2) (y 3))
  (let* ((x 7)
         (w (+ x y)))
    (* w x)))
```

Ex36 let* обчислює 70

Прив'язки виконуються послідовно зліва направо, а область прив'язки, позначена (**<variable>** **<init>**), є частиною виразу **let*** праворуч від прив'язки. Таким чином, друге зв'язування виконується в середовищі, в якому видно перше зв'язування, і так далі

Стандартна форма letrec

У Scheme є інша конструкція локального зв'язування, **letrec**, яка використовується під час створення взаємно рекурсивних локальних процедур.

Синтаксис:

(letrec ((<змінна 1> <init 1>) тіло ...))

тіло має бути послідовністю одного або кількох виразів.

Якщо <змінна> з'являється більше одного разу в списку зв'язаних змінних, **це є помилкою**.

Семантика: <змінні> прив'язані до нових місць, що містять невизначені значення, <init> оцінюються в результуючому середовищі, кожна <змінна> призначається результату відповідного <init>, <тіло> обчислюється в отриманому середовищі, і повертається значення останнього виразу в <тілі>. Кожне прив'язування <змінної> має весь вираз letrec як область, що дає змогу визначати взаємно рекурсивні процедури.

Стандартна форма letrec

letrec

```
(display "ex37 letrec result=")
(letrec ((even?
  (lambda (n)
    (if (zero? n)
      #t
      (odd? (- n 1)))))
  (odd?
  (lambda (n)
    (if (zero? n)
      #f
      (even? (- n 1)))))
  (even? 88)) ;#t
```

ex37 letrec result=#t

Процедури, створені у виразі, що ініціалізує змінні, можуть рекурсивно посилатися на визначені змінні.

Одне обмеження для letrec дуже важливе: має бути можливість оцінити кожен <init> без призначення чи посилання на значення будь-якої <змінної>. Якщо це обмеження порушується, то це є помилкою. Обмеження є необхідним, оскільки Scheme передає аргументи за значенням, а не за назвою.

У найпоширенішому використанні letrec усі <init> є лямбда-виразами, і обмеження задовольняються автоматично.

```
(define (triple x)
  (letrec ((y (+ x 2))
    (f (lambda (xx)
      (+ xx y w x)))
    (w (+ x 7)))
    (f -9)))
```

(triple 2)

6

;ex38

Стандартні процедури для роботи з числами в Scheme

Ціль	Процедура
Базові арифметичні оператори	<code>+, -, *, /, abs, quotient, remainder, modulo, gcd, lcm, expt, sqrt</code>
Дійсні числа	<code>numerator, denominator, rational?, rationalize</code>
Наближення	<code>floor, ceiling, truncate, round</code>
Точність	<code>inexact->exact, exact->inexact, exact?, inexact?</code>
Нерівності	<code><, <=, >, >=, =</code>
Предикати	<code>zero?, negative?, positive?, odd?, even?</code>
Максимум і мінімум	<code>max, min</code>
Тригонометрія	<code>sin, cos, tan, asin, acos, atan</code>
Експоненти	<code>exp, log</code>
Комплексні числа	<code>make-rectangular, make-polar, real-part, imag-part, magnitude, angle, complex?</code>
Перетворення типів	<code>number->string, string->number</code>
Предикат типу	<code>integer?, rational?, real?, complex?, number?</code>

Стандартні предикати для чисел

Предикати

Призначення	Форма
Тест на точність	(exact? z)
Тест на неточність	(inexact? z)
Перевірка на нуль	(zero? z)
Перевірка чи є число додатнім	(positive? x)
Перевірка чи є число від'ємним	(negative? x)
Перевірка чи є число непарним	(odd? n)
Перевірка чи є число парним	(even? n)

Числові типи

Математично, числа можуть бути організовані в вежу підтипів, в якій кожен рівень є підмножиною рівня вище останнього

number

complex

real

rational

integer

Наприклад, число 3 це ціле. Більш того, 3 також є раціональним, дійсним та комплексним. Те саме виконується і для Scheme чисел, які моделюють число 3. Для Scheme чисел ці типи визначаються предикатами **number?**, **complex?**, **real?**, **rational?** та **integer?**

Чисельні операції Scheme обробляють числа, як абстрактні дані, настільки незалежно від їхнього уявлення, наскільки можливо. Хоча реалізації Scheme можуть використовувати **fixnum** (числа з фіксованою комою), **flonum** (числа з плаваючою комою), і можливо інші уявлення для чисел, для звичайного програміста, що пише просту програму, це може бути не очевидно. Однак необхідно розрізняти числа, які видаються точно від тих, які можуть не бути такими. Наприклад, повинні бути точно відомі **індекси в структурах даних**, деякі **коефіцієнти многочлена символічної алгебраїчної системи**. З іншого боку, результати обчислень суттєво неточні, а ірраціональні числа можуть наближатися раціональними і навіть неточними наближеннями. Для того, щоб відловити випадки використання неточних чисел, де необхідні точні, у Scheme є точний поділ **на точні та неточні числа**.

Точність чисел в SCHEME

1. Числа Scheme є або точними, або неточними.
2. Число є точним, якщо воно записане як точна константа або отримано з точних чисел з використанням лише точних операцій.
3. Число є неточним, якщо воно записано як неточна константа і якщо воно отримане з використанням неточних інгредієнтів, або якщо воно отримане з використанням неточних операцій.
4. Якщо дві імплементації видають точні результати для обчислень, які не включають неточних проміжних результатів, два кінцеві результати будуть математично еквівалентними
5. Раціональні операції такі, як повинні завжди видавати точні результати при передачі точних аргументів.
6. Операція може повертати точний результат, якщо може довести, що значення результату не впливає неточність аргументів. Наприклад, множення будь-якого числа на точний нуль може видавати як результат точний нуль, навіть якщо інші аргументи є неточними.

Числові операції в SCHEME

процедура: (number? obj)
процедура: (complex? obj)
процедура: (real? obj)
процедура: (rational? obj)
процедура: (integer? obj)

```
(complex? 3+4i) ==> #t  
(complex? 3) ==> #f  
(real? 3) ==> #t  
(real? -2.5+0.0i) ==> #t  
(real? #e1e10) ==> #t  
(rational? 6/10) ==> #t  
(rational? 6/3) ==> #t  
(integer? 3+0i) ==> #t  
(integer? 3.0) ==> #f  
(integer? 8/4) ==> #t
```

- ❑ Дані предикати числових типів можуть застосовуватися до будь-якого типу аргументів, включаючи ті, які є числами.
- ❑ Вони повертають #t, якщо об'єкт іменованого типу, інакше вони повертають #f.
- ❑ Зазвичай, якщо предикат типу даного числа є істинним, то всі предикати типів, що включають даний тип, є істинними даного числа.
- ❑ Відповідно, якщо предикат типу для числа набуває хибного значення, то всі предикати типу нижнього рівня також набувають значення хибності для даного числа.

Числові операції в SCHEME

процедура: (= z1 z2 z3 ...)
процедура: (< x1 x2 x3 ...)
процедура: (> x1 x2 x3 ...)
процедура: (<= x1 x2 x3 ...)
процедура: (>= x1 x2 x3 ...)

Ці процедури повертають #t, якщо їх аргументи є (відповідно):
рівними, монотоннозростаючими, монотонно спадаючими,
монотонно не спадаючими або монотонно незростаючими.

библіотечная процедура: (zero? z)
библіотечная процедура: (positive? x)
библіотечная процедура: (negative? x)
библіотечная процедура: (odd? n)
библіотечная процедура: (even? n)

Бібліотечна процедура: (max X1 X2 ...)
Бібліотечна процедура: (min x1 x2 ...)

(**max** 3 4 2) ==> 4
(**max** 3.9 4 2) ==> 4.0

Якщо min або max використовується для **порівняння чисел різної точності** і чисельне значення результату не може бути представлене як неточне число без втрати точності, процедура може повідомити про порушення обмеження імплементації.

Числові операції в SCHEME

процедура: (+ z1 ...)

процедура: (* z1 ...)

процедура: (- z1 z2)

процедура: (- z)

Необов'язкова процедура: (- z1 z2 ...)

процедура: (/ z1 z2)

процедура: (/ z)

Необов'язкова процедура: (/ z1 z2 ...)

У випадку двох або більше аргументів ці процедури повертають різницю або частку аргументів, асоціативні зліва.

Якщо передається один аргумент, процедури повертають адитивну чи мультиплікативну інверсію цього аргументу.

(+ 3 4) ==> 7

(+ 3) ==> 3

(+) ==> 0

(* 4) ==> 4

(*) ==> 1

(- 3 4) ==> -1

(- 3 4 5) ==> -6

(- 3) ==> -3

(/ 3 4 5) ==> 3/20

(/ 3) ==> 1/3

Стандартні процедури для обробки чисел

Пошукові процедури та операції над числами

(max x1 x2 ...)	Пошук максимального з чисел
(min x1 x2 ...)	Пошук мінімального з чисел
(abs x)	Абсолютне значення числа

Додаткові операції ділення:

(quotient n1 n2)	Результат ділення $n1/n2$, якщо $n2 \neq 0$
(remainder n1 n2)	Остача від ділення $n1$ на $n2$, знак визначається чисельником
(modulo n1 n2)	Остача від ділення $n1$ на $n2$, знак визначається знаменником

Процедури , що повертають чисельник і знаменник дробу

(numerator q)	чисельник дробу
(denominator q)	знаменник дробу

Стандартні процедури для роботи з числами

Процедури обробки чисел

(floor x)	найбільше ціле число не більше ніж x.
(ceiling x)	найменше ціле число не менше ніж x.
(truncate x)	ціле число, абсолютна величина якого не більше абсолютної величини x
(round x)	ціле число шляхом округлення x

Тригонометричні процедури:

(exp z)	(log z)	
(sin z)	(asin z)	
(cos z)	(acos z)	
(tan z)	(atan z)	(atan y x)

(sqrt z)	корінь квадратний
(expt z1 z2)	зведення в степінь

Приклади використання стандартних процедур для обробки чисел

Для самостійної творчості

Стандартні процедури в Scheme

Призначення	Процедури
Конструкції	vector, make-vector, make-string, list
Предикати еквівалентності	eq? , eqv? , equal? , string=?, string-ci=?, char=?, char-ci=? =
Перетворення типів	vector->list, list->vector, number->string, string->number, symbol->string, string->symbol, char->integer, integer->char, string->list, list->string
Рядки	string?, make-string, string, string-length, string-ref, string-set!, string=?, string-ci=?, string<? string-ci<?, string<=? string-ci<=?, string>? string-ci>?, string>=? string-ci>=?, substring, string-append, string->list, list->string, string-copy, string-fill!
Символи	char?, char=?, char-ci=?, char<? char-ci<?, char<=? char-ci<=?, char>? char-ci>?, char>=? char-ci>=?, char-alphabetic?, char-numeric?, char-whitespace?, char-upper-case?, char-lower-case?, char->integer, integer->char, char-upcase, char-downcase
Вектори	make-vector, vector, vector?, vector-length, vector-ref, vector-set!, vector->list, list->vector, vector-fill!
Symbols	symbol->string, string->symbol, symbol?

Стандартні предикати еквівалентності

- ❑ Предикат — це процедура, яка завжди повертає логічне значення (#t або #f).
- ❑ Предикат еквівалентності є обчислювальним аналогом математичного відношення еквівалентності (воно є симетричним, рефлексивним і транзитивним).
- ❑ З предикатів еквівалентності
 - ✓ eq? є найкращим чи найбільш дискримінаційним
 - ✓ equal? є найгрубішим
 - ✓ eqv? є трохи менш дискримінаційним, ніж eq?

Синтаксис:

eqv? *obj1 obj2*

Предикат eq? повертає #t, якщо obj1 і obj2 зазвичай слід розглядати як той самий об'єкт

https://www.cs.cmu.edu/Groups/AI/html/r4rs/r4rs_8.html

Стандартні предикати еквівалентності

Предикат `eqv?` повертає `#t`, якщо:

- `obj1` і `obj2` обидва `#t` або обидва `#f`.
- `obj1` і `obj2` є символами
- `obj1` і `obj2` — обидва числа, чисельно рівні обидва точні або неточні
- `obj1` і `obj2` є символами і є однаковими символами відповідно до `char=?` процедури
- `obj1` та `obj2` є порожнім списком
- `obj1` і `obj2` — це пари, вектори або рядки, які позначають однакові місця в сховищі
- `obj1` і `obj2` — це процедури, теги розташування яких рівні

Предикат `eqv?` повертає `#f`, якщо:

- `obj1` і `obj2` мають різні типи
- один з `obj1` і `obj2` — `#t`, а інший — `#f`
- `obj1` і `obj2` є символами але предикат перетворення символа в рядок `=>` `#f`
- один з `obj1` і `obj2` є точним числом, а інший є неточним числом
- `obj1` і `obj2` — це числа, для яких процедура `=` повертає `#f`.
- `obj1` і `obj2` є символами, для яких `char=?` процедура повертає `#f`
- один з `obj1` і `obj2` є порожнім списком, а інший ні
- `obj1` і `obj2` — це пари, вектори або рядки, які позначають різні розташування
- `obj1` і `obj2` — це процедури, які поводитимуться по-різному (повертають інше значення або матимуть різні побічні ефекти) для деяких аргументів.

https://www.cs.cmu.edu/Groups/AI/html/r4rs/r4rs_8.html

Стандартні предикати еквівалентності

```
;=====
; ПРЕДИКАТИ ЕКВИВАЛЕНТНОСТІ
;=====
;-----ex 41 eqv? -----
(display "ex41 \n")
(eqv? 'a 'a)      ;=> #t
(eqv? 'a 'b)      ;=> #f
(eqv? 2 2)        ;=> #t
(eqv? '() '())    ;=> #t
(eqv? 100000000 100000000) ;=> #t
(eqv? (cons 1 2) (cons 1 2));=> #f
(eqv? (lambda () 1)
      (lambda () 2)) ;=> #f
(eqv? #f 'nil)     ;=> #f
(let ((p (lambda (x) x)))
  (eqv? p p))      ;=> #t
```

```
(display "ex41 continue \n")
(eqv? "" "")      ;=> unspecified #t
(eqv? '#() '())   ;=> unspecified #t
(eqv? (lambda (x) x)
      (lambda (x) x)) ;=> unspecified #f
(eqv? (lambda (x) x)
      (lambda (y) y)) ;=> unspecified #f
```

Стандартні предикати еквівалентності

```
;=====
; ПРЕДИКАТИ ЕКВИВАЛЕНТНОСТИ
;=====
;-----ex42 -----
(display "ex42 eqv?\n")
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g)) ; => #t
(eqv? (gen-counter) (gen-counter)) ; => #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g)) ; => #t
(eqv? (gen-loser) (gen-loser)) ; => #f
```

```
(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
          (g (lambda () (if (eqv? f g) 'both 'f))))
  (eqv? f g))
; => #f
```

```
(letrec ((f (lambda ()
              (if (eqv? f g) 'f 'both)
              ))
          (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g)) ; => #f
```

Стандартні предикати еквівалентності

eq? *obj1 obj2*

eq? схожий на eqv? за винятком того, що в деяких випадках він здатний розпізнавати відмінності більш тонкі, ніж ті, які виявляє eqv?.

Eq? та eqv? гарантовано мають однакову поведінку щодо символів, логічних значень, порожнього списку, пар і непорожніх рядків і векторів.

Поведінка Eq? щодо чисел і символів залежить від реалізації, але вона завжди повертатиме true або false і повертатиме true лише тоді, коли eqv? також поверне true. eq? також може поводитися інакше, ніж eqv? на порожніх векторах і порожніх рядках.

Зазвичай можна реалізувати eq? набагато ефективніше, ніж eqv?, наприклад, як просте порівняння вказівників замість складнішої операції. Однією з причин є те, що неможливо обчислити eqv? двох чисел за постійний час, тоді як eq? реалізоване як порівняння вказівників завжди завершуватиметься за постійний час.

eq? можна використовувати як eqv? у програмах, які використовують процедури для реалізації об'єктів із станом, оскільки він підкоряється тим самим обмеженням, що й eqv?.

Стандартні предикати еквівалентності

```
(display "ex43 eq? \n")
(eq? 'a 'a)           ;=> #t
(eq? '(a) '(a))       ;=> unspecified
(eq? (list 'a) (list 'a)) ;=> #f
(eq? "a" "a")         ;=> unspecified
(eq? "" "")           ;=> unspecified
(eq? '() '())         ;=> #t
(eq? 2 2)             ;=> unspecified
(eq? #\A #\A)         ;=> unspecified
(eq? car car)         ;=> #t
(let ((n (+ 2 3)))
  (eq? n n))          ;=> unspecified
(let ((x '(a)))
  (eq? x x))          ;=> #t
(let ((x '#()))
  (eq? x x))          ;=> #t
(let ((p (lambda (x) x)))
  (eq? p p))          ;=> #t
```

Стандартні предикати еквівалентності

equal? *obj1 obj2*

equal? рекурсивно порівнює вміст пар, векторів і рядків, застосовуючи **eqv?** на інших об'єктах, таких як числа та символи. Емпіричне правило полягає в тому, що об'єкти загалом **equal?** якщо вони друкують однаково. **equal?** може не завершитися, якщо його аргументи є циклічними структурами даних.

```
;-----ex44 equal?-----  
(equal? 'a 'a)           ;=> #t  
(equal? '(a) '(a))       ;=> #t  
(equal? '(a (b) c)  
  '(a (b) c))           ;=> #t  
(equal? "abc" "abc")     ;=> #t  
(equal? 2 2)             ;=> #t  
(equal? (make-vector 5 'a)  
  (make-vector 5 'a)); => #t  
(equal? (lambda (x) x)  
  (lambda (y) y))       ;=> unspecified
```

Стандартні предикати еквівалентності

```
(display "ex45 equal? \n")  
(= 2 3)    ;=> #f  
(= 2.5 2.5) ;=> #t  
;(= '() '()) ;=> error  
(define x '(2 3))  
(define y '(2 3))  
(eq? x y)   ;=> #f  
(define y x)  
(eq? x y)   ;=> #t  
(eqv? 2 2)  ;=> #t  
(eqv? "a" "a") ;=> залежить від реалізації  
(define x '(2 3))  
(define y '(2 3))  
(equal? x y) ;=> #t  
(eqv? x y)   ;=> #f
```

Загалом:

1. Використовуйте предикат **=**, якщо ви хочете перевірити, чи є два **числа** еквівалентними.
2. Використовувати **eqv?** предикат, коли ви хочете перевірити, чи є два **нечислових** значення еквівалентними.
3. Використовувати **equal?** предикат, коли ви хочете перевірити, чи є **два списки, вектори** тощо еквівалентними.
4. **Не використовуйте eq?** предикат, якщо ви точно не знаєте, що робите.

Стандартні процедури в Scheme

Призначення	Процедури
Пари і списки	pair?, cons, car, cdr, set-car!, set-cdr!, null?, list?, list, length, append, reverse, list-tail, list-ref, memq, memv, member, assq, assv, assoc, list->vector, vector->list, list->string, string->list
Предикати ідентичності	boolean?, pair?, symbol?, number?, char?, string?, vector?, port?, procedure?
Продовження	call-with-current-continuation (call/cc), values, call-with-values, dynamic-wind
Оточення	eval, scheme-report-environment, null-environment, interaction-environment (optional)
Ввід\вивід	display, newline, read, write, read-char, write-char, peek-char, char-ready?, eof-object? open-input-file, open-output-file, close-input-port, close-output-port, input-port?, output-port?, current-input-port, current-output-port, call-with-input-file, call-with-output-file, with-input-from-file(optional), with-output-to-file(optional)
Системний інтерфейс	load (optional), transcript-on (optional), transcript-off (optional)
Вид обчислення	force
Функціональне програмування	procedure?, apply, map, for-each
Булеві змінні	boolean? not

Стандартні процедури перетворення

Процедури перетворення

`(exact-> inexact x)` перетворення точного числа в неточне

`(inexact-> exact x)` перетворення неточного числа в точне

`(string-> number string)` Перетворення рядка в число

`(string-> number string radix)` Перетворення рядка в число

Тут `radix` є основа системи числення (точне ціле число 2, 8, 10 або 16).

Приклад

`(string->number "100" 16)` Результат **256**

Процедури вищого порядку

- ❑ В Scheme багато зумовлених операцій забезпечуються не синтаксисом, а **змінними, значеннями яких є процедури**.
- ❑ Операція +, наприклад, в Scheme є всього лише регулярним ідентифікатором, пов'язаним з процедурою, що додає числові об'єкти.
- ❑ Процедури, по суті, є **абстракціями**, які описують складові операції над числами безвідносно до конкретних значень.
- ❑ **!!!При виконанні різних операцій потрібно будувати процедури, які приймають інші процедури як аргументи або повертають їх як значення.**
- ❑ Процедура, що маніпулює іншими процедурами, називається **процедурою вищого порядку (higher-order procedure)**.

Процедури як аргументи

Розглянемо такі процедури


1. Обчислює суму цілих чисел від a до b:

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
```

2. Обчислює суму кубів цілих чисел в заданому діапазоні:

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

3. Обчислює куб цілого числа



```
(define (cube x)
  (* x (* x x)))
```

4. Обчислює суму послідовності термів в ряді, який сходиться до $\pi/8$:
 $1/(1*3)+1/(5*7)+1/(9*11)+....$

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

Процедури як аргументи

За цими процедурами стоїть одна загальна схема:

- одна функція обчислює терм, що підлягає додаванню,
- інша функція обчислює наступне значення а.

Всі ці процедури можна породити, застосувавши шаблон:

```
(define (<имя> a b)
  (if (> a b)
      0
      (+ (<терм> a)
         (<имя> (<наступний> a) b))))
```

В наведеному шаблоні можна перетворити семантичні означення у формальні параметри:

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b)))))
```

`sum` приймає в якості аргументів нижню, верхню межі `a` і `b` і процедури `term` і `next`.

`sum` можна використовувати так, як будь-яку іншу процедуру.

Процедури як аргументи

Процедура `inc` збільшує аргумент на 1

```
(define (inc n)
  (+ n 1))
```

Процедура `cube` обчислює куб числа

```
(define (cube x)
  (* x (* x x)))
```

Процедура `sum` підсумовує два числа, приймає в якості аргументів нижню, верхню межі `a` і `b` і процедури `term` і `next`.

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

За допомогою `sum` можна визначити `sum-cubes`

```
(define (sum-cubes a b)
  (sum cube a inc b))
```

Скориставшись цим визначенням, можна обчислити суму кубів чисел від 1 до 10 (**виклик процедури**):

```
(sum-cubes 1 10)
3025
```

Процедури як аргументи

Процедура `inc` збільшує аргумент на 1

Процедура ідентичності `identity` повертає значення свого аргументу

Процедура `sum` підсумовує числа, приймає в якості аргументів нижню, верхню межі `a` і `b` і процедури `term` і `next`.

Процедура `sum-integers` підсумовує числа, в діапазоні від нижньої межі `a` до верхньої межі `b`.

Тепер можна скласти цілі числа від 1 до 10 (**виклик процедури**)

За допомогою процедури ідентичності (яка повертає свій аргумент) для обчислення терму, можна визначити `sum-integers` через `sum`:

```
(define (inc n)
  (+ n 1))
```

```
(define (identity x)
  x)
```

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

```
(define (sum-integers a b)
  (sum identity a inc b))
```

```
(sum-integers 1 10)
55
```



Процедури як аргументи

Так само визначається процедура pi-sum:

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a pi-next b))
```



```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

За допомогою цих процедур можна обчислити наближення до π (**виклик процедури**):

```
(* 8 (pi-sum 1 1000))
3.139592655589783
```

```
(* 8 (pi-sum 1 100))
3.1215946525910105
```


Процедури як аргументи

Процедуру `sum` можна використовувати в якості будівельного блоку при формулюванні інших понять.

Наприклад, **визначений інтеграл** функції `f` між межами `a` і `b` для малих `dx` можна чисельно оцінити за допомогою формули:

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] dx$$

```
(define (integral f a b dx)
  (define (add-dx x)
    (+ x dx))
  (* (sum f (+ a (/ dx 2)) add-dx b)
     dx))
```

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```



```
(define (cube x)
  (* x (* x x)))
```

Виклик процедур з різними `dx` для функції куба числа:

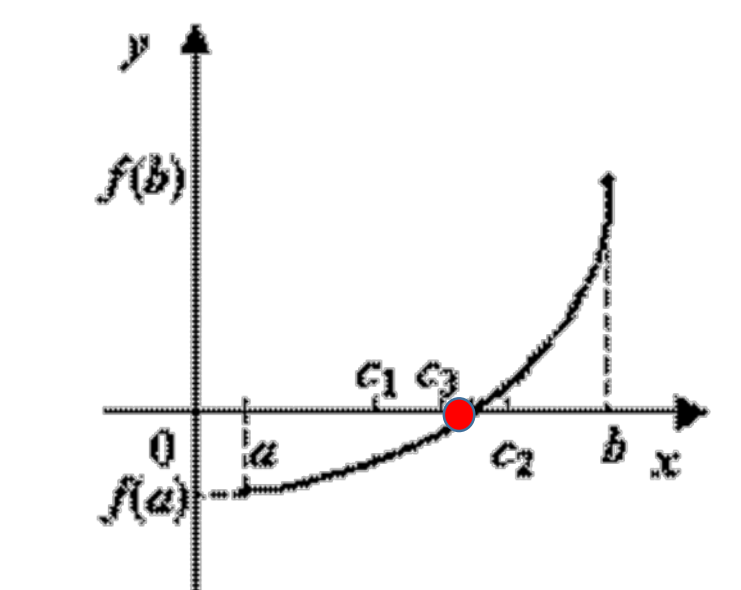


```
(integral cube 0 1 0.01)
.24998750000000042
```

```
(integral cube 0 1 0.001)
.2499998750000001
```

Приклад. Знаходження коренів рівнянь методом половинного ділення

- ❑ Метод половинного ділення (half-interval method) - це простий спосіб знаходження коренів **рівняння $f(x) = 0$** , де f - неперервна функція.
- ❑ Ідея полягає в тому, що якщо є такі точки a і b , що **$f(a) < 0 < f(b)$** , то функція f повинна мати принаймні один нуль на відрізку між a і b .
- ❑ Щоб знайти його, візьмемо x , що дорівнює середньому між a і b , і обчислимо $f(x)$.
- ❑ Якщо **$f(x) > 0$** , то f повинна мати нуль на відрізку між a і x .
- ❑ Якщо **$f(x) < 0$** , то f повинна мати нуль на відрізку між x і b .
- ❑ Продовжуючи таким чином, ми зможемо знаходити все більш вузькі інтервали, на яких f повинна мати нуль.
- ❑ Коли ми дійдемо до точки, де цей інтервал досить малий, процес зупиняється



Приклад. Знаходження коренів рівнянь методом половинного ділення

Процедура, яка реалізує стратегію пошуку методом половинного ділення:

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
```

Приклад. Знаходження коренів рівнянь методом половинного ділення

Процедура, яка реалізує стратегію пошуку методом половинного ділення:

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint)))))))
```

1. Є функція **f** і дві точки, в одній із яких значення функції від'ємне **neg-point**, в іншій додатне **pos-point**.
2. Спочатку обчислюємо середнє між двома краями інтервалу - **average**.
3. Потім ми перевіряємо, чи не є інтервал вже досить малим - **close-enough?**
4. Якщо інтервал між точками малий, повертаємо середню точку як відповідь - **midpoint**.
5. Якщо інтервал ще великий, обчислюємо значення **f** в середній точці - **test-value**.
6. Якщо це значення додатне - **positive?**, продовжуємо процес з інтервалом від вихідної від'ємної точки до середньої точки – **search**.
7. Якщо значення в середній точці від'ємне - **negative?**, ми продовжуємо процес з інтервалом від середньої точки до вихідної додатної точки.
8. Нарешті, існує можливість, що значення в середній точці в точності дорівнює 0, і тоді середня точка і є шуканий корінь..

Приклад. Знаходження коренів рівнянь методом половинного ділення

Перевірка, чи достатньо близькі кінці інтервалу пошуку кореня

```
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
```

Розрахунок середньо арифметичного двох значень

```
(define (average x y)
  (/ (+ x y) 2))
```

Обчислення значення функції в середній точці

```
(define (test-value f midpoint)
  (< розрахунок виразу>))
```

Обчислення модуля числа

```
(define (abs x)
  (if (positive? x )
      x
      (- x)))
```

Приклад. Знаходження коренів рівнянь методом половинного ділення

- ❑ Використовувати процедуру **search** безпосередньо незручно, оскільки випадково можна дати їй точки, в яких значення **f** не мають потрібних знаків, і в цьому випадку отримаємо неправильну відповідь.
- ❑ Замість цього будемо використовувати **search** за допомогою процедури, яка перевіряє, який кінець інтервалу має додатне, а який від'ємне значення, і відповідним чином викличе **search**.
- ❑ Якщо на обох кінцях інтервалу функція має однаковий знак, метод половинного ділення використовувати не можна, і тоді процедура повідомляє **про помилку**.

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "У аргументов не різні знаки " a b)))))
```

Виклик процедури для пошуку кореня рівняння **$\sin x = 0$** , що лежить між 2 та 4:



Результат



```
(half-interval-method sin 2.0 4.0)
3.14111328125
```

Приклад. Знаходження нерухомих точок функцій

Число x називається нерухомою (фіксованою) точкою (fixed point) функції f , якщо воно задовольняє рівнянню $f(x) = x$.

Для деяких функцій f можна знайти нерухому точку, почавши з якогось значення і застосовуючи f багаторазово:

$f(x), f(f(x)), f(f(f(x))), \dots$

поки значення не перестане сильно змінюватися.

За допомогою цієї ідеї можна скласти процедуру **fixed-point**, яка в якості аргументів приймає функцію і початкове значення і виробляє наближення до нерухомої точки функції. Багато разів застосовуємо функцію, поки не знайдеться два послідовних значення, різниця між якими менше деякої заданої чутливості:

```
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

```
(define tolerance 0.00001)
```

```
(fixed-point cos 1.0)
.7390822985224023
```

Приклад. Знаходження нерухомих точок функцій

```
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Форма зв'язування імені функції f з параметром `first-guess`

Форма зв'язування імені функції f з параметром `first-guess`

Процедури як значення, що повертаються

Ідея – створити процедури, які повертають значення у вигляді процедур

Розглянемо **приклад процедури обчислення квадратного кореня** \sqrt{x} як пошук нерухомої точки, вважаючи, що \sqrt{x} є нерухома точка функції $y = x / y$.

Потім використовуємо гальмування усередненням, щоб змусити наближення сходитися. При цьому, отримавши функцію **f**, повертаємо функцію, значення якої в точці **x** є середнє арифметичне між **x** і **f(x)**

Процедура, що реалізує Ідею гальмування усередненням

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
```

average-damp - це процедура, яка бере в якості аргументу процедуру **f** і повертає в якості значення процедуру (отриману за допомогою **lambda**), яка, будучи застосована до числа **x**, повертає середнє між **x** і **(f x)**.

```
((average-damp square) 10)
55
```

Застосування **average-damp** до процедури **square** отримує процедуру, значенням якої для деякого числа **x** буде середнє між **x** і **x²**.

Процедури як значення, що повертаються

Використовуючи **average-damp**, ми можемо переформулювати процедуру обчислення квадратного кореня наступним чином:

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
    1.0))
```

Можна узагальнити процедуру пошуку квадратного кореня так, щоб вона отримувала кубічні корені

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
    1.0))
```

Приклад. Процедури як значення, що повертаються

Розглянемо поняття **похідної**. Взяття похідної, подібно до гальмування усередненням, трансформує одну функцію в іншу.

Наприклад, похідна функції x^3 є функція $3x^2$.

У загальному випадку, якщо g є функція, а dx - маленьке число, то похідна Dg функції g є функція, значення якої в кожній точці x описується формулою при dx , яка прагне до нуля:

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

```
(define dx 0.00001)

(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
        dx)))
```

Процедура `deriv` бере процедуру в якості аргументу і повертає процедуру як значення.

Наприклад, щоб знайти наближене значення похідної x^3 в точці 5 :

```
(define (cube x) (* x x x))
```

```
((deriv cube) 5)
```

```
75.00014999664018
```

Література з програмування на Scheme

1. Навчальні матеріали Ковалюк Т.В. <https://github.com/tkovalyuk/>
2. Стандарт Scheme, версія 6.
http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-2.html#node_toc_start
3. Стандарт Scheme, версія 7. Revised7 Report on the Algorithmic Language Scheme.
<http://www.larcenists.org/Documentation/Documentation0.98/r7rs.pdf>
4. Абельсон Гарольд, Сассман Джеральд Джей, Сассман Джули. Структура и интерпретация компьютерных программ. <https://www.twirpx.com/file/81061/>
<https://library.kre.dp.ua/Books/2-4%20kurs/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%B8%20%D1%96%20%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8%20%D0%BE%D0%B1%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D1%8C/%D0%94%D0%BE%D0%B4%D0%B0%D1%82%D0%BA%D0%BE%D0%B2%D1%96%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D1%96%D0%B0%D0%BB%D0%B8/%D0%90%D0%B1%D0%B5%D0%BB%D1%8C%D1%81%D0%BE%D0%BD%2C%20%D0%A1%D0%B0%D1%81%D1%81%D0%BC%D0%B0%D0%BD%20-%20%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0%20%D0%B8%20%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%86%D0%B8%D1%8F%20%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D1%8B%D1%85%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC.pdf>
5. R. Kent Dybvig. The Scheme Programming Language. <https://www.scheme.com/tspl4/>
6. Кристиан Кеннек. Интерпретация Лиспа и Scheme. <http://blog.ilammy.net/lisp/index.html>
7. Майлингова О. Л., Манжелей С. Г., Соловская Л. Б. Прототипирование программ на языке Scheme. <https://docplayer.ru/71381060-Prototipirovanie-programm-na-yazyke-scheme-metodicheskoe-posobie-po-praktikumu.html>

Джерела

1. Harold Abelson, Gerald Jay Sussman, Julie Sussman. Structure and Interpretation of Computer Programs. The MIT Press. 2005 (Харольд Абельсон, Джеральд Джей Сассман, Джули Сассман. Структура и интерпретация компьютерных программ. «Добросвет», 2006)
2. Филд. А., Харрисон П. Функциональное программирование. –М.: «Мир», 1993
3. Городня Л. Введение программирование на языке Лисп.
http://ict.edu.ru/ft/005133/prog_lisp.pdf
4. Хювенен Ё. Сеппянен И. Мир Лиспа. Т.1. Введение в Лисп и функциональное программирование. 1990 bydlokoder.ru/index.php?p=books_LISP
5. Кристиан Кеннек. Интерпретация Лиспа и Scheme. Электронный ресурс. Режим доступа: <http://blog.ilammy.net/lisp/>
6. **An Introduction to Scheme and its Implementation**
https://www.cs.utexas.edu/ftp/garbage/cs345/schintro-v14/schintro_toc.html
7. **Revised(4) Report on the Algorithmic Language Scheme**
https://www.cs.cmu.edu/Groups/AI/html/r4rs/r4rs_toc.html

Література з програмування на Haskell, Lisp, Common Lisp, ML

Інші мови функціонального програмування

1. Антон Холомьёв. Учебник по Haskell.
<https://docplayer.ru/25937980-Uchebnik-po-haskell-anton-holomyov.html>
2. John Harrison. Введение в функциональное программирование.
<https://nsu.ru/xmlui/bitstream/handle/nsu/8874/Harrison.pdf;jsessionid=7BDBFCF0EA05BFD026052B868E6DAEDF?sequence=1>
3. Лидия Городняя. Введение в программирование на языке Лисп.
http://window.edu.ru/resource/684/41684/files/prog_lisp.pdf
4. Практический Common Lisp. <http://lisper.ru/pcl/pcl.pdf>



Дякую за увагу

Доц. кафедри ПСТ,
к.т.н. Ковалюк Т.В.

tkovalyuk@ukr.net

<https://github.com/tkovalyuk/functional-program>