

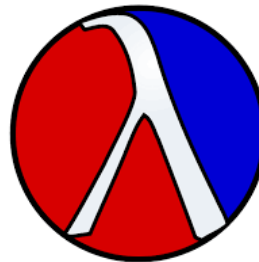
Функціональне програмування

*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 повераються.

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	<code>define-syntax</code> , <code>let-syntax</code> , <code>letrec-syntax</code> , <code>syntax-rules (R5RS)</code> , <code>syntax-case (R6RS)</code>
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	do
Конструкції прив'язки	let, let*, letrec
Умовні обчислення	cond, case
Умовні обчислення	and, or
Ітерації	named let
Відкладені обчислення	delay
Послідовні обчислення	begin

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 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)
Відобчислення	force
Функціональне програмування	procedure?, apply, map, for-each
Булеві змінні	boolean? not

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>
Ввід\вивід	<code>number->string, string->number</code>
Предикат типу	<code>integer?, rational?, real?, complex?, number?</code>

Вбудовані предикати

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 не парним	(odd? n)
Перевірка чи є число парним	(even? n)

Вбудовані процедури

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

Вбудовані 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

Вбудовані процедур перетворення

Процедури перетворення

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

`(inexact-> exact z)` 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**

Умовні вирази

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.

Умовні вирази

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.

Умовні вирази

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

Приклад умовних виразів

```
(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
```


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:

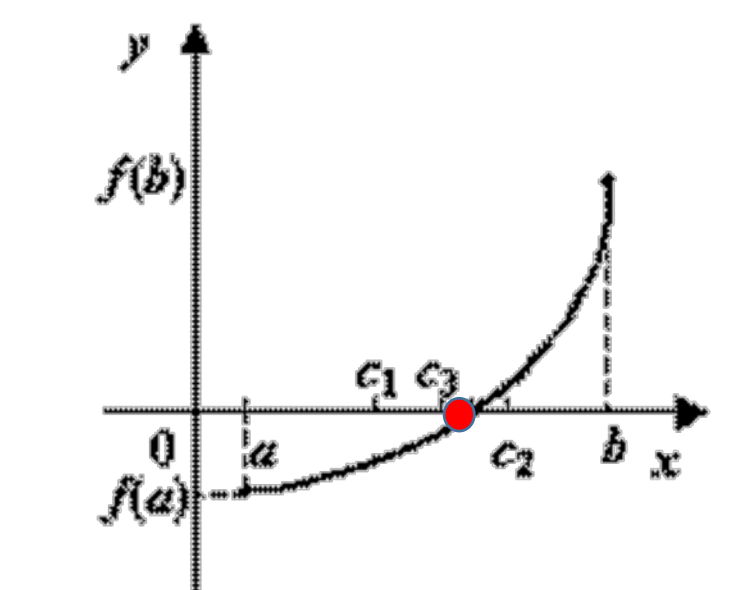


```
(integral cube 0 1 0.01)
.249987500000000042
```

```
(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 сходиться. 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/>

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>