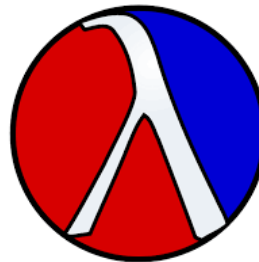


# Функціональне програмування

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

tkovalyuk@ukr.net  
[https://github.com/tkovalyuk/functional program](https://github.com/tkovalyuk/functional_program)

# Лінійні та рекурсивні процеси обчислення в SCHEME/Lisp/...



# План лекції 3

1. Форми
2. Процедури
3. Лінійні рекурсія і ітерація
  - 3.1. Лінійно-рекурсивний процес обчислень
  - 3.2. Лінійно-ітеративний процес обчислення
4. Особливості реалізації рекурсій
5. Деревоподібна рекурсія
6. Приклад рекурсії.
  - 6.1. Зведення в степінь
  - 6.2. Знаходження найбільшого спільного дільника
8.  $\lambda$  - форма (lambda-форма)
9. Створення локальних змінних за допомогою форми let
  - 9.1 Знаходження коренів рівнянь методом половинного ділення

# Форми

Хоча визначення не є виразами, складові виразу і визначення мають схожу синтаксичну структуру:

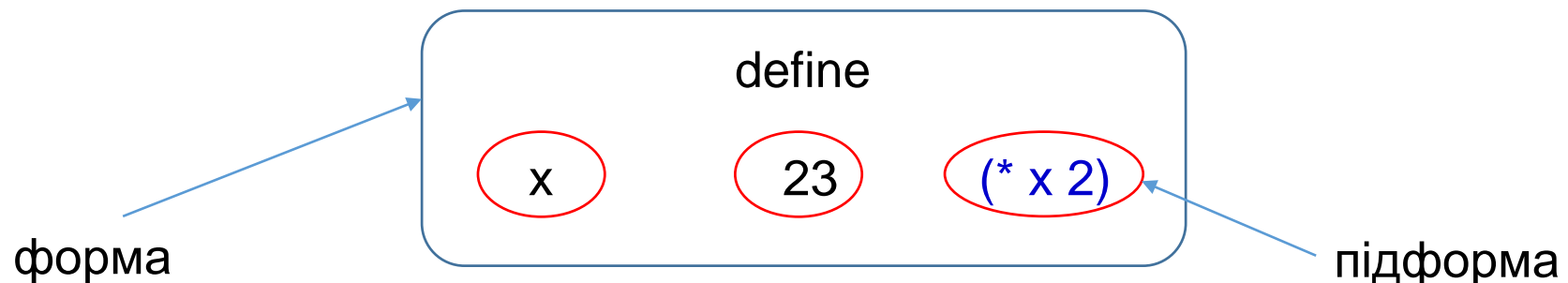
```
(define x 23)
(* x 2)
```

При цьому перший рядок містить **визначення**, а наступний рядок – **вираз**.

Дана відмінність ґрунтується на зв'язуванні означень **define** та **\***.

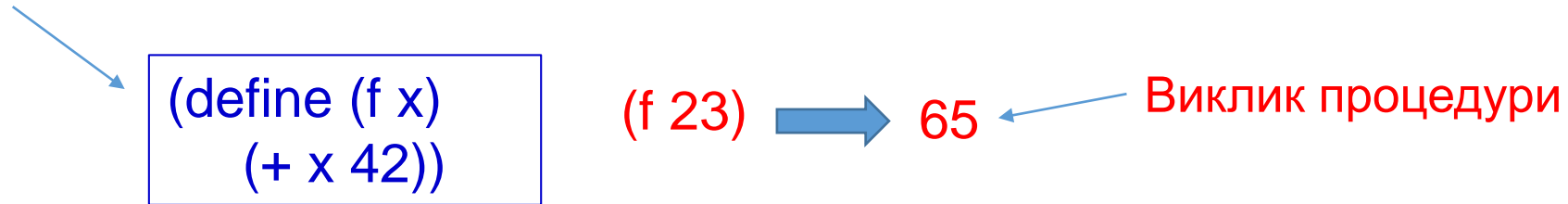
На чисто синтаксичному рівні обидві є **формами**, а форма є узагальненою назвою **синтаксичних частин програми** Scheme.

Зокрема, **23** є підформою форми **(define x 23)**



# Процедури

Визначення можуть також використовуватися для побудови процедур.

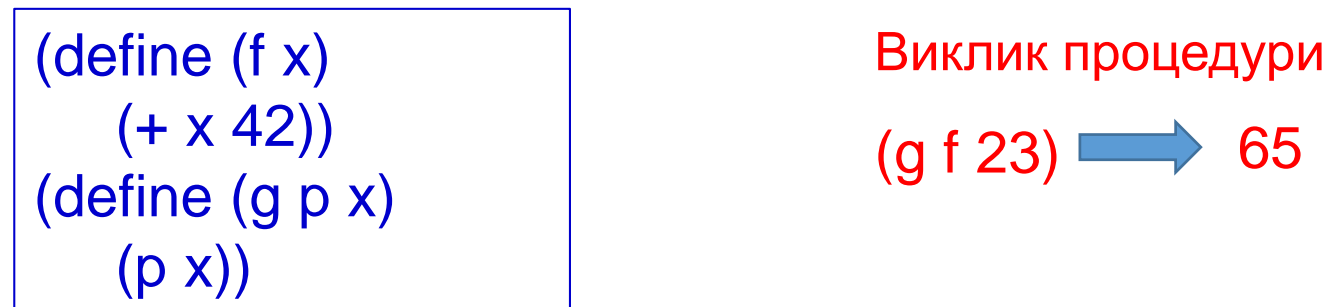


Процедура є абстракцією виразу за допомогою об'єктів.

Круглі дужки навколо **f x** позначають визначення процедури.

Вираз `(f 23)` є викликом процедури, і означає "вирахувати `(+ x 42)` (тіло процедури) з `x`, прив'язаним до `23`".

Оскільки процедури є об'єктами, їх можна передавати в інші процедури:



# Процедури

- ❑ Фактично багато зумовлених операцій Scheme забезпечуються не синтаксисом, а змінними, значеннями яких є процедури.
- ❑ Операція **+**, наприклад, набуває спеціального синтаксичного трактування в багатьох інших мовах, в Scheme є всього лише **регулярним ідентифікатором, пов'язаним з процедурою**, відповідною числовому об'єкту.
- ❑ Те саме стосується і **\***, і багатьох інших:

# Лінійні рекурсія і ітерація

Розглянемо функцію факторіал, яка визначається рівнянням

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

Існує безліч способів обчислювати факторіали. Один з них полягає в тому, що  $n!$  для будь-якого додатнього цілого числа  $n$  дорівнює  $n$ , помноженому на  $(n - 1)!$ :

$$n! = n \cdot [(n - 1) \cdot (n - 2) \cdot \dots \cdot 3 \cdot 2 \cdot 1] = n \cdot (n - 1)!$$

Таким чином, можна обчислити  $n!$ , обчисливши спочатку  $(n - 1)!$ , а потім помноживши його на  $n$ .

Якщо додати умову, що  $1!$  дорівнює  $1$ , можна записати процедуру

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

# Лінійно-рекурсивний процес обчислень

- ❑ Підставкова модель показує спочатку серію **розширень**, а потім **стиснення**.
- ❑ Розширення відбувається по мірі того, як процес будує ланцюжок відкладених операцій (deferred operations), в даному випадку ланцюжок множень.
- ❑ Стиснення відбувається тоді, коли виконуються ці відкладені операції.
- ❑ **Такий тип процесу, який характеризується ланцюжком відкладених операцій, називається рекурсивним процесом (recursive process).**
- ❑ Виконання цього процесу вимагає, щоб інтерпретатор запам'ятовував, які операції він повинен виконати згодом.
- ❑ При обчисленні  **$n!$**  довжина ланцюжка відкладених множень, а отже, і обсяг даних, яких потрібно зберегти, зростає **лінійно** з ростом  **$n$**  (пропорційний  $n$ ), як і число кроків.
- ❑ Такий процес називається **лінійно рекурсивним процесом (linear recursive process).**



# Лінійно рекурсивний процес для обчислення 6!

```
(factorial 6)  
(* 6 (factorial 5))  
(* 6 (* 5 (factorial 4)))  
(* 6 (* 5 (* 4 (factorial 3))))  
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))  


---

(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1)))))  
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))  
(* 6 (* 5 (* 4 (* 3 2))))  
(* 6 (* 5 (* 4 6)))  
(* 6 (* 5 24))  
(* 6 120)  
720
```

Процес розширення  
(занурення)

Процес стиснення

# Лінійні рекурсія і ітерація

Якщо описати це обчислення через лічильник і добуток, які з кожним кроком одночасно змінюються згідно з правилом:

**добуток = лічильник × добуток**  
**лічильник = лічильник + 1**

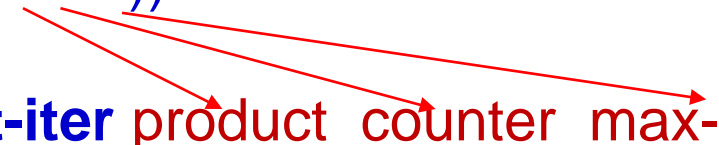
і додавши умову, що

**$n!$  - це значення добутку в той момент, коли лічильник стає більше, ніж  $n$ ,**  
можна записати процедуру обчислення факторіала так:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

```
(define (factorial n)
  (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                  (+ counter 1)
                  max-count)))
```

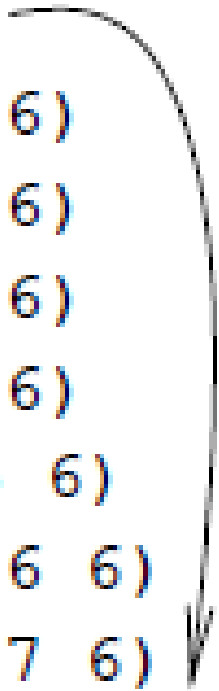


# Лінійно-ітеративний процес обчислень

- ❑ Цей процес не росте і не стискається.
- ❑ На кожному кроці при будь-якому значенні  $n$  необхідно пам'ятати лише поточні значення змінних **product**, **counter** і **max-count**.
- ❑ Такий процес називається **ітеративним** (iterative process).
- ❑ У загальному випадку, **ітеративний процес - це такий процес**,
  1. стан якого можна описати кінцевою кількістю змінних стану (state variables)
  2. заздалегідь заданим правилом, що визначає, як ці змінні стану змінюються від кроку до кроку,
  3. (можливо) тест на завершення, який визначає умови, за яких процес повинен закінчити роботу.
- ❑ При обчисленні  $n!$  кількість кроків **лінійно** зростає з ростом  $n$ .
- ❑ Такий процес називається **лінійно ітеративним процесом** (linear iterative process).

# Лінійно-ітеративний процес обчислення для 6!

```
(factorial 6)  
(fact-iter 1 1 6)  
(fact-iter 1 2 6)  
(fact-iter 2 3 6)  
(fact-iter 6 4 6)  
(fact-iter 24 5 6)  
(fact-iter 120 6 6)  
(fact-iter 720 7 6)  
720
```



# Особливості реалізації рекурсій

- Більшість реалізацій звичайних мов (включаючи процедурний C++) побудовані так, що інтерпретація рекурсивної процедури поглинає обсяг пам'яті, лінійно зростаючий пропорційно кількості викликів процедури, навіть якщо процес, що описаний, ітеративний.
- Як наслідок, ці мови здатні описувати ітеративні процеси тільки за допомогою спеціальних **циклічних конструкцій** на зразок **do**, **for** і **while**.
- Реалізація Scheme вільна від цього недоліку. Вона буде виконувати ітеративний процес, використовуючи **фіксований обсяг пам'яті**, навіть якщо він описується рекурсивною процедурою. Така властивість реалізації мови називається **підтримкою хвостової рекурсії (tail recursion)**.
- **Якщо реалізація мови підтримує хвостову рекурсію, то ітерацію можна виразити за допомогою звичайного механізму виклику функцій.**

# Деревоподібна рекурсія

Існує ще одна схема обчислень, яка називається **деревоподібна рекурсія (tree recursion)**. Як приклад розглянемо обчислення послідовності чисел Фібоначчі, в якій кожне число є сумою двох попередніх:

**0, 1, 1, 2, 3, 5, 8, 13, 21, ...**

Рекурентне означення для чисел Фібоначчі можна сформулювати так:

$$\text{Fib}(n) = \begin{cases} 0 & \text{если } n = 0 \\ 1 & \text{если } n = 1 \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{в остальных случаях} \end{cases}$$

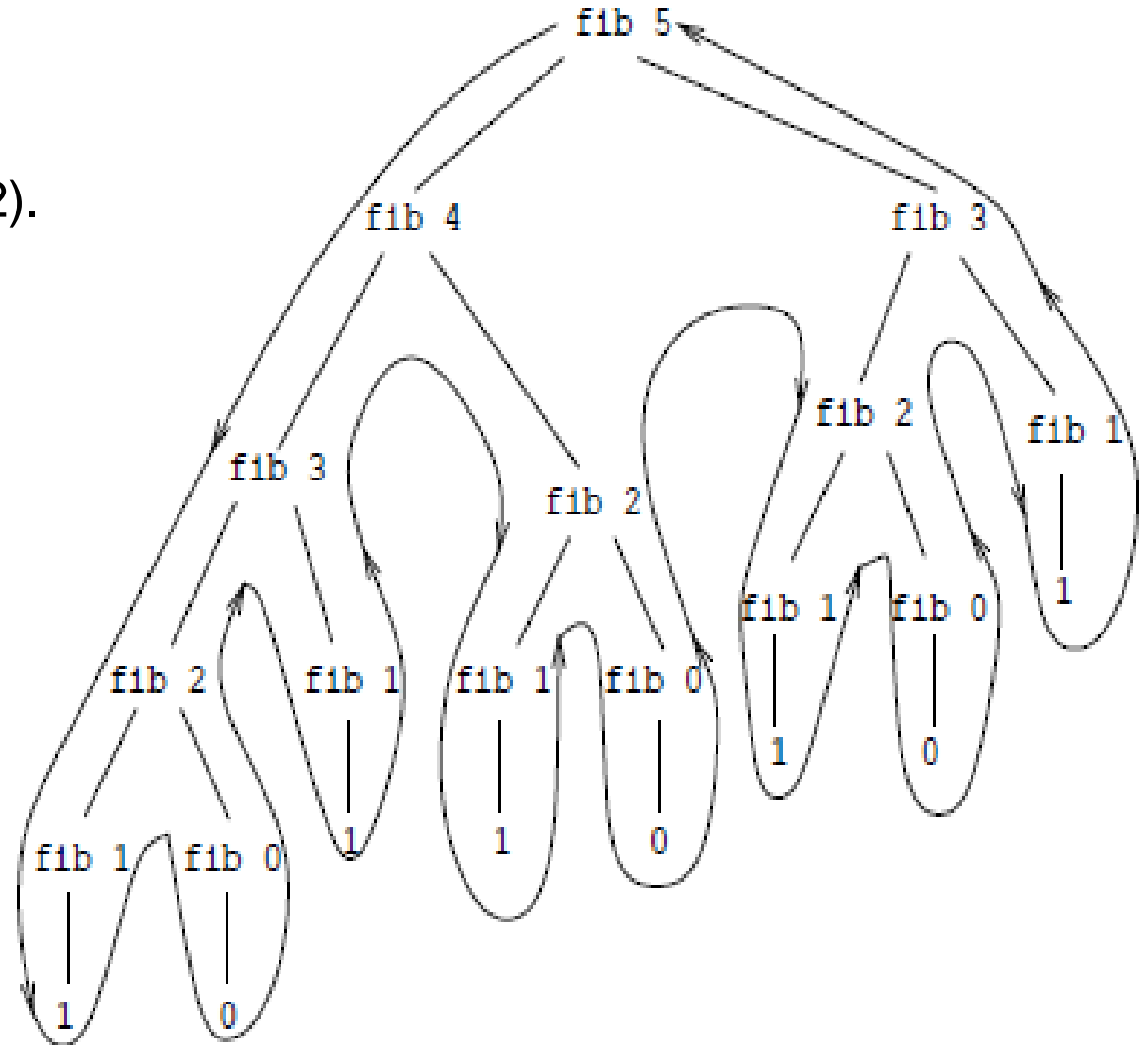
```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

# Деревоподібна рекурсія

**Розглянемо схему цього обчислення.**

- ❑ Щоб обчислити (fib 5), спочатку обчислюємо (fib 4) і (fib 3).
- ❑ Щоб обчислити (fib 4), обчислюємо (fib 3) і (fib 2).
- ❑ Загалом, виходить процес схожий на дерево

У загальному випадку число кроків, необхідних деревовидно-рекурсивним процесам, пропорційно числу вершин дерева, а необхідний обсяг пам'яті пропорційний максимальній глибині дерева.



# Ітеративна процедура обчислення чисел Фібоначчі

Для отримання чисел Фібоначчі ми можемо сформулювати **ітеративний процес**.

Для цього використовуємо пару цілих **a** і **b**, яким на початку даються значення **Fib (1) = 1** і **Fib (0) = 0**, і на кожному кроці застосовуємо одночасну трансформацію

$$a \leftarrow a + b$$

Після того, як виконана ця трансформація **n** разів, **a** і **b** будуть відповідно рівні **Fib (n + 1)** і **Fib (n)**.

Таким чином, можна **ітеративно** обчислювати числа Фібоначчі за допомогою процедури

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```



# Приклад рекурсії. Зведення в степінь рекурсивно

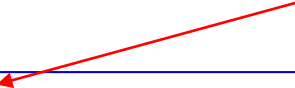
Розглянемо задачу зведення числа в степінь.

Для цього потрібна процедура, яка, прийнявши в якості аргументу основу **b** і додатне ціле значення степеня **n**, повертає **b<sup>n</sup>**.

Один із способів отримати значення – використати рекурсивне визначення

$$b^n = b \times b^{n-1}$$
$$b^0 = 1$$

яке переводиться в процедуру:



```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

Ім'я функції

Це **лінійно рекурсивний процес**, що вимагає **O(n)** кроків і O(n) пам'яті.

# Зведення в степінь ітеративно

Можна сформулювати **еквівалентну лінійну ітерацію**:

```
(define (expt b n)
  (expt-iter b n 1))

(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                  (- counter 1)
                  (* b product))))
```

Ця версія потребує  $O(n)$  кроків та  $O(n)$  пам'яті.

# Зведення в степінь

Враховуючи правило:

$$\begin{cases} b^n = (b^{n/2})^2 & \text{якщо } n \text{ парне} \\ b^n = b \times b^{n-1} & \text{якщо } n \text{ непарне} \end{cases}$$

можна отримати процедуру:

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

де **предикат**, що перевіряє ціле число на парність, визначений через елементарну процедуру **remainder** – остача від ділення двох чисел:


```
(define (even? n)
  (= (remainder n 2) 0))
```

# Приклад рекурсії. Знаходження найбільшого спільного дільника

За визначенням, найбільший спільний дільник (НСД) двох цілих чисел **a** і **b** - це найбільше ціле число, на яке і **a**, і **b** діляться без залишку.

Наприклад, НСД 16 і 28 дорівнює 4.

Існує **алгоритм Евкліда**, який заснований на тому, що якщо **r** є **залишок** від ділення **a** на **b**, то загальні дільники **a** і **b** в точності ті самі, що і загальні дільники **b** і **r**. Таким чином, **можна скористатися рівнянням**

$$\text{НСД}(a, b) = \text{НСД}(b, r)$$


щоб послідовно звести задачу знаходження НСД до задачі знаходження НСД все менших і менших пар цілих чисел.

Наприклад,

$$\begin{aligned}\text{НСД}(206, 40) &= \text{НСД}(40, 6) \\ &= \text{НСД}(6, 4) \\ &= \text{НСД}(4, 2) \\ &= \text{НСД}(2, 0) \\ &= 2\end{aligned}$$

зводить НСД (206, 40) до НСД (2, 0), що дорівнює двом.

# Приклад рекурсії. Знаходження найбільшого спільного дільника

Алгоритм Евкліда у вигляді Scheme процедури

```
(define (nod a b)
  (if (= b 0)
      a
      (nod b (remainder a b))))
```

Процедура породжує ітеративний процес, кількість кроків якого росте пропорційно логарифму чисел-аргументів.

# λ - форма (lambda)

Процедури можна визначити формою `lambda`.

За словом `lambda` слідує список аргументів, після нього - послідовність виразів, які описують власне обчислення (тіло) функції.

У загальному випадку, `lambda` використовується для створення процедур так само, як `define`, тільки **ніякого імені для процедури не вказується**:

**(lambda (<формальні-параметри>) <тіло>)**

Виходить така сама повноцінна процедура, як і за допомогою `define`. Єдина різниця полягає в тому, що вона **не пов'язана з жодним ім'ям в оточенні**.

# λ - форма (lambda)

(lambda (<формальні-параметри>) <тіло>)

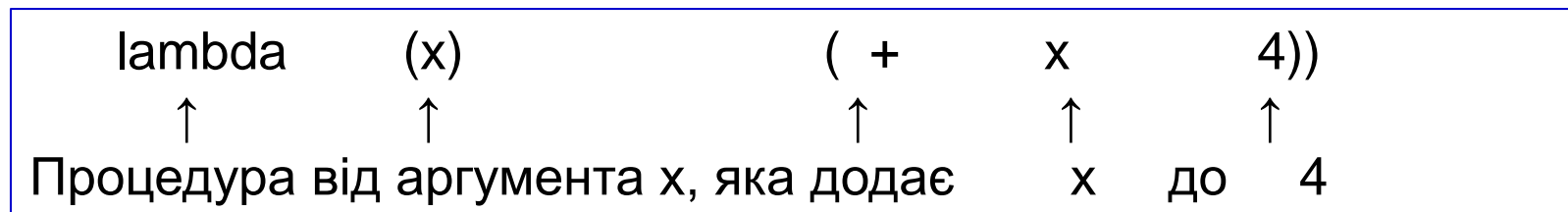
Насправді

(define (plus4 x) (+ x 4))

еквівалентно

(define plus4 (lambda (x) (+ x 4)))

Можна читати **вираз lambda** так:



# λ - форма (lambda)

Приклад процедури, яка повертає свій аргумент плюс 4

```
(lambda (x)  
  (+ x 4))
```

Приклад процедури, яка обчислює число,  
зворотне добутку аргумента і аргумента плюс 2

```
(lambda (x)  
  (/ 1.0 (* x (+ x 2))))
```

Тогда процедуру **pi-sum** можна описати без допоміжних процедур:

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



# λ - форма (lambda)

За допомогою lambda можна записати процедуру **integral**, не визначаючи допоміжну процедуру **add-dx**:

```
(define (integral f a b dx)
  (* (sum f
        (+ a (/ dx 2.0))
        (lambda (x) (+ x dx))
        b)
     dx))
```

Подібно будь-якому виразу, значенням якого є процедура, вираз з **lambda** можна використовувати як **оператор в комбінації**, наприклад:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
12
```

# Створення локальних змінних за допомогою форми let

Ще одне застосування `lambda` полягає у введенні **локальних змінних**.

Часто в процедурі бувають потрібні **локальні змінні** крім тих, що пов'язані формальними параметрами. Наприклад, треба обчислити функцію

$$f(x, y) = x(1 + xy)^3 + y(1 - y) + (1 + xy)(1 - y)$$

яку також можна виразити як:

$$a = 1 + xy$$

$$b = 1 - y$$

$$f(x, y) = xa^2 + yb + ab$$

Для обчислення `f` хотілося б мати як локальні змінні не тільки `x` і `y`, а й імена для проміжних результатів `a` і `b`. Можна зробити це за допомогою допоміжної процедури, яка пов'язує локальні змінні:

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
      (* y b)
      (* a b)))

  (f-helper (+ 1 (* x y))
    (- 1 y)))
```

# Створення локальних змінних за допомогою форми let

Безіменну процедуру для зв'язування локальних змінних можна записати через **lambda-вираз**. При цьому тіло **f** виявляється просто викликом цієї процедури.

```
(define (f x y)
  ((lambda (a b)
    (+ (* x (square a))
      (* y b)
      (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

Така конструкція настільки корисна, що є особлива форма під назвою **let**, яка робить її більш зручною. З використанням **let** процедуру **f** можна записати так:

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
      (* y b)
      (* a b))))
```

# Створення локальних змінних за допомогою форми let

Загальна форма виразу з **let** така:

```
(let ((<змінна1> <вираз1>)
      (<змінна2> <вираз2>))
  ...
  (<зміннаn> <виразn>))
<тіло>
```

Це можна розуміти так:

Нехай **<змінна1>** має значення **<вираз1>** і  
**<змінна2>** має значення **<вираз2>**

...  
і **<зміннаn>** має значення **<виразn>** в **<тілі>**

# Створення локальних змінних за допомогою форми let

- ❑ Перша частина **let**-висловлювання є список пар виду **ім'я-значення**.
- ❑ Коли **let** обчислюється, кожне **ім'я пов'язується зі значенням** відповідного виразу.
- ❑ Потім обчислюється **тіло let**, причому ці імена пов'язані як локальні змінні.
- ❑ Відбувається це так: **вираз let інтерпретується як альтернативна форма для:**

```
((lambda (<змінна1> ... <зміннаn>)
  <тіло>)
<вираз1> ... <виразn>)
```

Отже, вираз **let** - це всього лише синтаксична форма для виклику **lambda**

```
(let ((x π1)) π2) ≡ ((lambda(x) π2) π1)
```

# Створення локальних змінних за допомогою форми let

З цієї еквівалентності видно, що **область визначення змінної, введенної в let-виразі - тіло let.**

Звідси слідує що:

1. **let** дозволяє пов'язувати змінні як завгодно близько до того місця, де вони використовуються.
2. Значення змінних обчислюються за межами **let**. Це істотно, коли вирази, що дають значення локальних змінних, залежать від змінних, які мають ті самі імена, що й самі локальні змінні.

```
(+ (let ((x 3))  
    (+ x (* x 10)))  
x)
```

При  $x=5$  значення виразу  $=38$ .

Значення  $x$  в тілі **let** дорівнює 3, так що значення **let**-вирази одно 33.

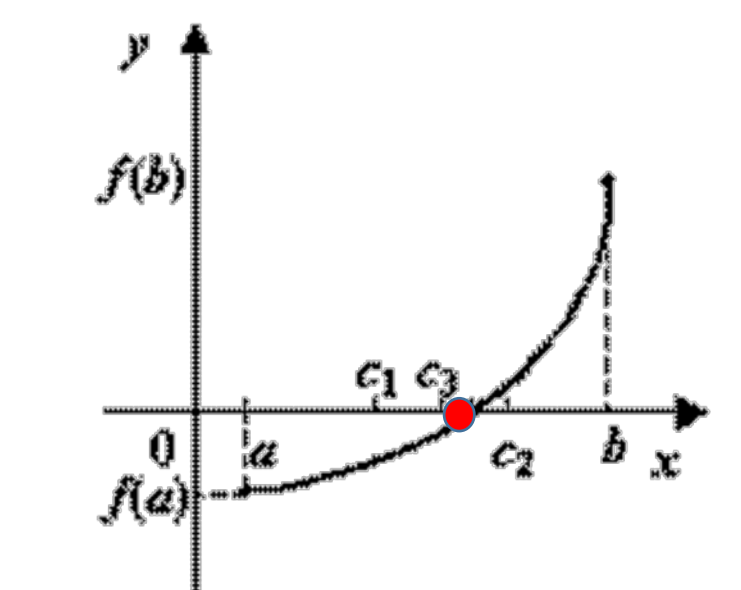
З іншого боку,  $x$  як другий аргумент до зовнішнього  $+$  як і раніше дорівнює 5

```
(let ((x 3)  
      (y (+ x 2)))  
    (* x y))
```

Якщо значення  $x = 2$ , вираз буде мати значення 12, оскільки всередині тіла **let**  $x$  дорівнюватиме 3, а  $y = 4$  (що дорівнює зовнішньому  $x$  плюс 2).

# 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:



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

Результат



# 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  $\text{first-guess}$

Форма зв'язування імені функції  $f$  з параметром  $\text{first-guess}$

# 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](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>
8. Как реализовать оптимизацию хвостовой рекурсии с помощью Common Lisp.  
<https://russianblogs.com/article/72331609404/>

# 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](http://ict.edu.ru/ft/005133/prog_lisp.pdf)
4. Хювенен Е. Сеппянен И. Мир Лиспа. Т.1. Введение в Лисп и функциональное программирование. 1990 [bydlokoder.ru/index.php?p=books\\_LISP](http://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](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](mailto:tkovalyuk@ukr.net)

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

