

# Функціональне програмування

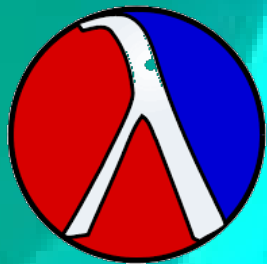
*Лектор Ковалюк Тетяна Володимирівна  
к.т.н., доцент*

`tkovalyuk@ukr.net`  
`https://github.com/tkovalyuk/funcprogram`

# Лекція 5

## Введення в абстракцію даних.

### Списки та дерева (діалект Scheme)

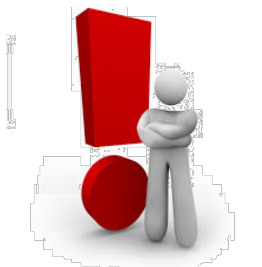


# Зміст

1. Ведення в абстракцію даних
2. Пари
3. Ієрархічні дані і властивість замикання
4. Подання послідовностей
5. Операції зі списками
6. Дерева як Ієрархічні структури
7. Подання дерев

# Поняття абстракції даних

- ❑ Часто доводиться будувати обчислювальні об'єкти, що складаються з декількох частин, щоб змодельовати багатосторонні явища реального світу.
- ❑ **Складені обчислювальні об'єкти** представляють комбінацію об'єктів даних, що використовується для створення **абстракцій даних**.
- ❑ Програма поводить ся зі складеними об'єктами даних (compound data object) **як з єдиним поняттям**, не вирізняючи складові частини такого об'єкта даних
- ❑ Використання складених даних дозволяє збільшити **модульність** програми, надаючи можливість відокремити частину програми, що працює зі складеними об'єктами даних, від деталей представлення цих об'єктів даних.



**Методологія проектування програм, в якій частина програми, яка має справу з поданням об'єктів даних, відділена від тих частин, де ці об'єкти даних використовуються, називається абстракцією даних**

# Поняття абстракції даних

- ❑ **Абстракція даних** - це методологія, яка дозволяє відокремити спосіб використання складеного об'єкта даних від деталей того, як він складений з елементарних даних.
- ❑ **Конкретне уявлення даних** визначається незалежно від програм, які ці дані використовують.
- ❑ **Інтерфейсом** між двома цими частинами системи (абстрактними даними і конкретним уявленням складових даних) служить набір процедур, які називаються **селекторами (selectors)** і **конструкторами (constructors)**, що реалізують абстрактні дані в термінах конкретного уявлення.

# Приклад створення складених даних

Розглянемо раціональну арифметику і реалізуємо операції:

- додавання,
- віднімання,
- множення,
- ділення раціональних чисел,
- перевірки, чи рівні два раціональних числа один одному.

Припустимо, що існує спосіб побудувати раціональне число з його чисельника і знаменника (тобто існує **конструктор**).

Припустимо, що маючи раціональне число, можна отримати його чисельник і знаменник (тобто здійснити **селекцію за допомогою селектора**).

Припустимо також, що конструктор і два селектора доступні у вигляді процедур:

- ❑ **(make-rat *n d*)** повертає раціональне число, чисельник якого ціле  $\langle n \rangle$ , а знаменник - ціле  $\langle d \rangle$ .
- ❑ **(numer *x*)** повертає чисельник раціонального числа  $\langle x \rangle$ .
- ❑ **(denom *x*)** повертає знаменник раціонального числа  $\langle x \rangle$ .

Поки що невідомо, як подати раціональне число і як повинні реалізовуватися процедури **numer**, **denom** і **make-rat**. Проте, якби ці процедури були, можна було б складати, віднімати, множити, ділити і перевіряти на рівність раціональні числа за допомогою наступних правил:

# Приклад створення складених даних

Арифметика раціональних чисел:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1 d_2}{d_1 n_2}$$

$$\frac{n_1}{d_1} = \frac{n_2}{d_2} \text{ тогдa и только тогдa,}$$

когда  $n_1 d_2 = n_2 d_1$

numer, denom – селектори,  
make-rat - конструктор

Арифметика раціональних чисел в процедурах:

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

```
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

```
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

```
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
```

```
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

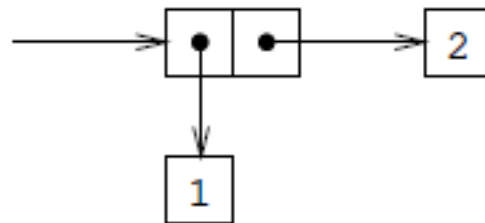
# Пара (pair)

Для реалізації конкретного рівня абстракції даних є складова структура, що називається **парою** (**pair**), і вона створюється за допомогою елементарної процедури-конструктора **cons**.

Процедура **cons** приймає два аргументи і повертає об'єкт даних, який містить ці два аргументи як елементи.

Маючи пару, можна отримати її частини за допомогою елементарних процедур **car** і **cdr**. Використовують **cons**, **car** і **cdr** так:

```
(define x (cons 1 2))  
(car x)  
1  
(cdr x)  
2
```



(cons 1 2) у вигляді  
**стрілочної діаграми**

Пара є об'єктом, якому можна дати ім'я і працювати з ним, подібно до елементарного об'єкту даних. Можна використовувати **cons** для створення пар, елементи яких самі пари:

```
(define x (cons 1 2))  
(define y (cons 3 4))  
(define z (cons x y))  
(car (car z))  
1  
(car (cdr z))  
3
```

Об'єкти даних, складені з пар, називаються дані зі **списковою структурою** (List-structured data) .



# Конструктори та селектори пари

## Конструктор

`(cons x y)`

## Селектори

`(car p)` — перший елемент, head Contents of the Address part of Register

`(cdr p)` — другий елемент, tail Contents of the Decrement part of Register

## Закони типу

$(\text{car } (\text{cons } X \ Y)) = X$

$(\text{cdr } (\text{cons } X \ Y)) = Y$

# Приклад використання пар

Пари дозволяють природним чином завершити побудову системи раціональних чисел. Будемо просто представляти раціональне число у вигляді пари двох цілих чисел: чисельника і знаменника.

Тоді `make-rat`, `numer` і `denom` реалізуються так:

```
(define (make-rat n d) (cons n d))  
(define (numer x) (car x))  
(define (denom x) (cdr x))
```

Коли потрібно виводити результати обчислень, друкуватиме раціональне число, спочатку виводячи його чисельник, потім косу риску і потім знаменник:

```
(define (print-rat x)  
  (newline)  
  (display (numer x))  
  (display "/")  
  (display (denom x)))
```

**display, newline** – елементарні процедури SCHEME

# Приклад арифметики раціональних чисел

```
(define (numer x) (car x)) ;чисельник
(define (denom x) (cdr x)) ;знаменник
(define (make-rat n d) (cons n d)) ; створення пари
(define (print-rat x); друк пари
  (newline)
  (display (numer x))
  (display "/" )
  (display (denom x)))

(print-rat (make-rat 5 6)) ;виклик друку пари у вигляді дробу
```

Welcome to DrRacket, version 7.8 [3m].  
Language: R5RS; memory limit: 128 MB.

5/6

# Приклад арифметики раціональних чисел

```
;=====
;операції з дробами
;=====
(define fraction1 (make-rat 4 7));створення дробу 1
(define fraction2 (make-rat 5 9));створення дробу 2
;=====додавання=====
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x))
              )
            (* (denom x) (denom y))
  )
)

(print-rat (add-rat fraction1 fraction2))
```

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

$$\frac{4 \times 9 + 5 \times 7}{7 \times 9} = \frac{71}{63}$$



71/63

# Приклад арифметики раціональних чисел

=====ВІДНІМАННЯ=====

```
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
                (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(print-rat (sub-rat fraction1 fraction2))
```

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$$

=====division=====

```
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
```

$$\frac{n_1/d_1}{n_2/d_2} = \frac{n_1 d_2}{d_1 n_2}$$

```
(print-rat (div-rat fraction1 fraction2))
```

=====МНОЖЕННЯ=====

```
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

```
(print-rat (mul-rat fraction1 fraction2))
```

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$$

=====порівняння=====

```
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

```
(newline)
```

```
(equal-rat? fraction1 fraction2)
```

$$n_1 d_2 = n_2 d_1$$

1/63

36/35

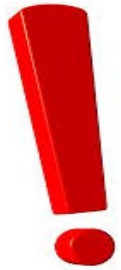
20/63

#f

# Абстракція даних

Основна **ідея абстракції** даних полягає в тому, щоб:

1. визначити для кожного типу об'єктів даних **набір базових операцій**, через які будуть виражатися всі дії з об'єктами цього типу,
2. при роботі з даними **використовувати тільки визначений раніше набір операцій**.



Можна вважати, що дані - це те, що визначається деяким набором **селекторів і конструкторів**, а також деякими **умовами**, яким ці процедури повинні задовольняти, щоб бути правильним поданням

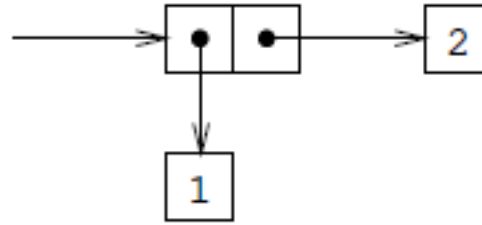
Для побудови пари склеюємо два об'єкти за допомогою **cons**, а за допомогою **car** і **cdr** можемо отримати самі об'єкти з набору даних. Тобто ці операції задовольняють умові, що для будь-яких об'єктів **x** і **y**, якщо

$z \in (\text{cons } x \ y), \text{ то } (\text{car } z) \in x, \text{ а } (\text{cdr } z) \in y.$

Здатність працювати **з процедурами як з об'єктами** автоматично надає можливість представляти складені дані.

# Ієрархічні дані і властивість замикання

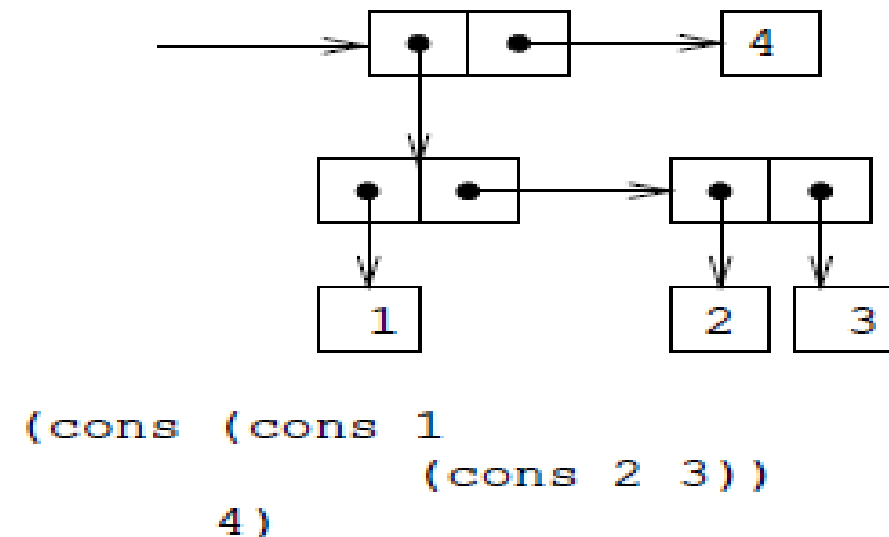
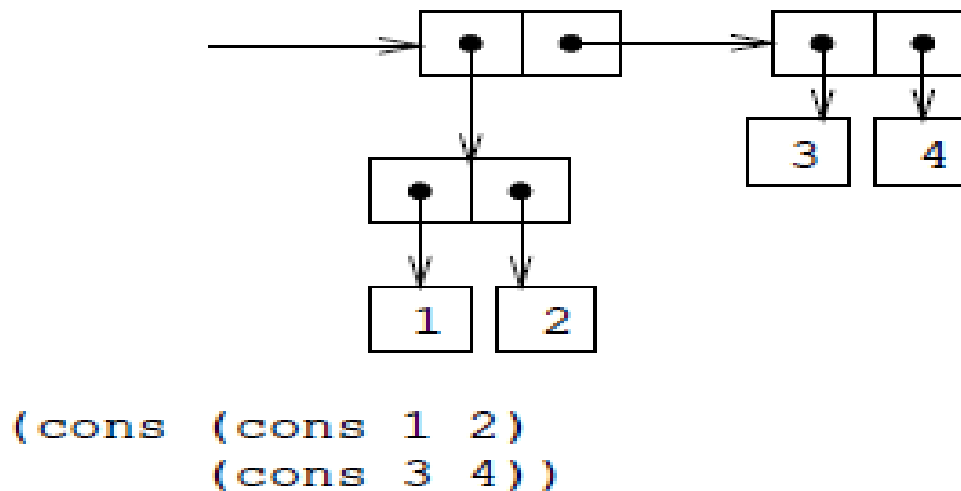
Подання пари (`cons 1 2`) у вигляді стрілочної діаграми означає:



1. Кожен об'єкт зображується у вигляді стрілки (pointer), що вказує на якусь комірку.
2. Комірка, що зображає елементарний об'єкт, містить уявлення цього об'єкта. Наприклад, комірка, відповідна числу, містить числову константу.
3. Зображення пари складається з двох комірок, ліва з них містить покажчик на `car` цієї пари, а права - її `cdr`.

# Ієрархічні дані і властивість замикання

- ❑ Можливість створювати пари, елементи яких самі є парами, визначає **спискову структуру** як засіб представлення даних.
- ❑ Ця можливість називається **властивістю замикання** (closure property) для **cons**.
- ❑ У загальному випадку, операція **комбінування об'єктів** даних має **властивість замикання** в тому випадку, якщо результати з'єднання об'єктів за допомогою цієї операції самі можуть з'єднуватися цією самою операцією.
- ❑ **Замикання** дозволяє будувати **ієрархічні структури**, тобто структури, які складені з частин, які самі складені з частин, і так далі.



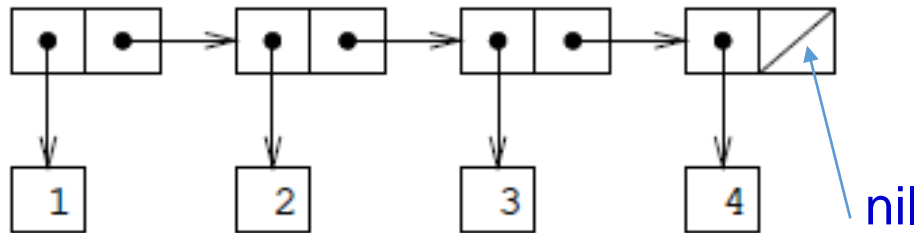


# Подання послідовностей

Одна з корисних структур, які можна побудувати за допомогою пар - це **послідовність (sequence)**, тобто впорядкована сукупність об'єктів даних.

Послідовність представлена як **ланцюжок пар**:

- ❑ У кожній парі **car** - це відповідний член ланцюжка,
- ❑ У кожній парі **cdr** - наступна пара ланцюжка.
- ❑ **cdr** останньої пари вказує на особливе значення **nil**, яке не є парою.



```
(cons 1  
  (cons 2  
    (cons 3  
      (cons 4 ))))
```

# Подання послідовностей

- ❑ Послідовність пар, породжувана вкладеними **cons**, називається **СПИСКОМ** (list).
- ❑ У Scheme є примітив **list**, який допомагає будувати списки.
- ❑ Вищевказану послідовність можна було б отримати за допомогою виразу **(list 1 2 3 4)**

В загальному випадку вираз

**(list <a1> <a2> ... <an>)**

еквівалентний виразу

**(cons <a1> (cons <a2> (cons ... (cons <an> nil) ... )))**

**(define one-through-four (list 1 2 3 4))**

**one-through-four** ; виклик форми  
**(1 2 3 4)**

Тут **(list 1 2 3 4)** - вираз,  
**(1 2 3 4)** - результат обчислення цього виразу



# Подання послідовностей. Конструктор списку

Процедура **car** вибирає перший елемент зі списку, а **cdr** повертає підсписок, що складається з усіх елементів, крім першого.

Вкладені застосування **car** і **cdr** можуть вибрати другий, третій і наступні елементи списку. Конструктор **cons** породжує список, подібний вихідному, але з додатковим елементом на початку.

```
(define one-through-four (list 1 2 3 4))
```

```
(car one-through-four)
```

1

```
(cdr one-through-four)
```

(2 3 4)

```
(car (cdr one-through-four))
```

2

```
(cons 10 one-through-four)
```

(10 1 2 3 4)

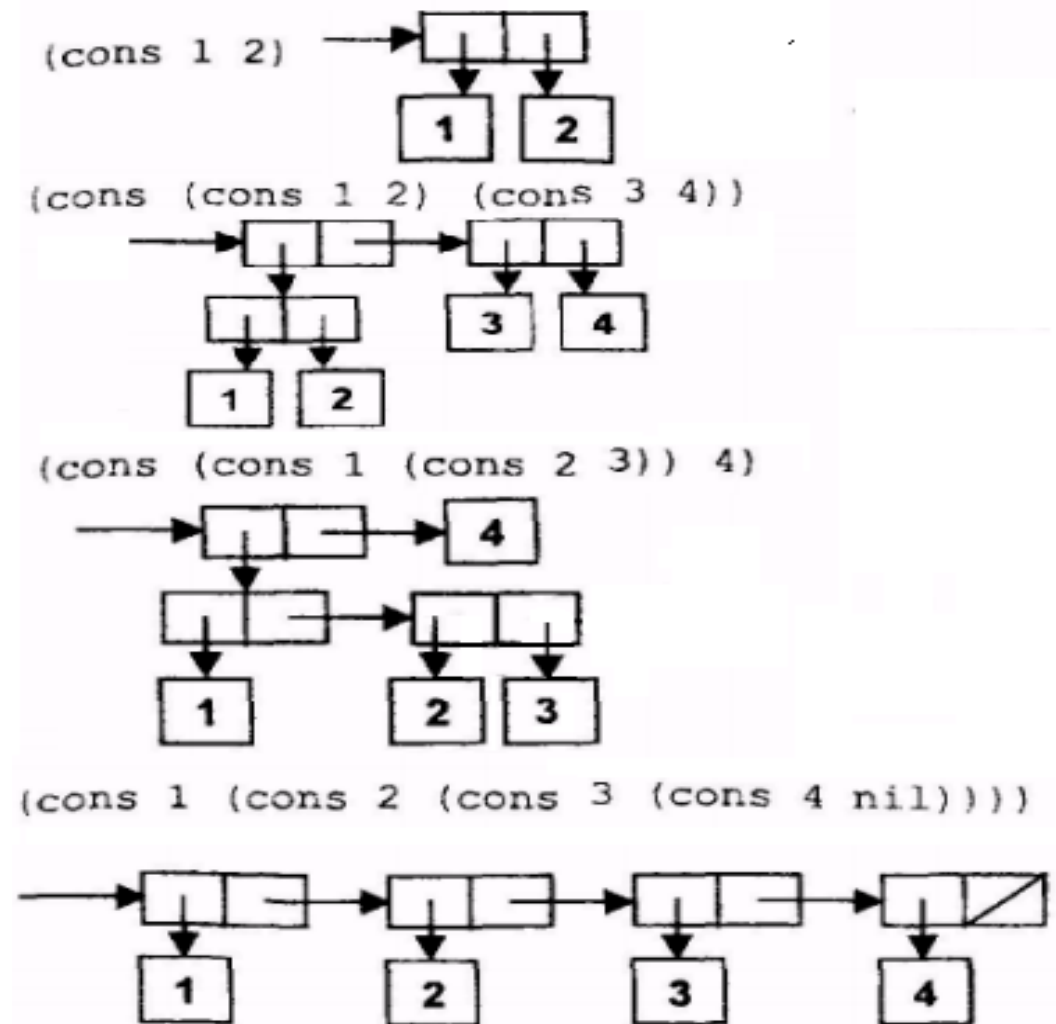
```
(cons 5 one-through-four)
```

(5 1 2 3 4)

Значення **nil**, яким завершується ланцюжок пар, можна розглядати як послідовність без елементів, тобто порожній список (**empty list**).

# Подання послідовностей

- ❑ Об'єкти можна представляти у вигляді покажчика на деякий бокс.
- ❑ Бокс простого об'єкта містить текстове представлення цього об'єкта.
- ❑ Бокс для складеного об'єкта (типу пара) – подвійний:
  - Ліва частина містить покажчик на перший елемент пари (**car пари**),
  - В правій частині знаходиться покажчик на другий елемент пари (**cdr пари**).



# Точкові пари

Символьні вирази складаються з елементів, які називаються **атомами**, що об'єдні в список за допомогою **дужок**.

Якщо змінити положення і кількість дужок, то разом з цим зміниться структура і зміст виразу.

**Атоми** бувають **символьні** і **числові**.

**Логічні константи** також вважаються **атомами**.

Символьний атом розглядається як одне неподільне ціле. До символьним атомам застосовується тільки одна операція - **порівняння**.

## Загальні правила побудови символьних виразів

1. Атом є символьний вираз.
2. Послідовність символьних виразів, укладена в дужки, є символьним виразом.

У загальному випадку, другим аргументом процедури **cons** може бути як **список**, так і просто **атом**.

І в першому і в другому випадку отримуємо об'єкт типу **точкова пара**.

**Список є окремим випадком пари, в якій самий правий елемент є атомом nil.**

# Точкові пари

(cons 'a 3) -> (a . 3)  
(cons ' (a b) 'c) -> ((a b) . c)  
'abc -> abc  
'a -> a

‘идентификатор => символи ідентифікатора

Значення виразу (**cons e1 e2**) представляється точкової парою (**v1.v2**), де **v1** і **v2** є символьними виразами для значень **e1** і **e2**, незалежно від того, якого типу ці значення.

Правила запису точкових пар дозволяють також використовувати і спискову нотацію:

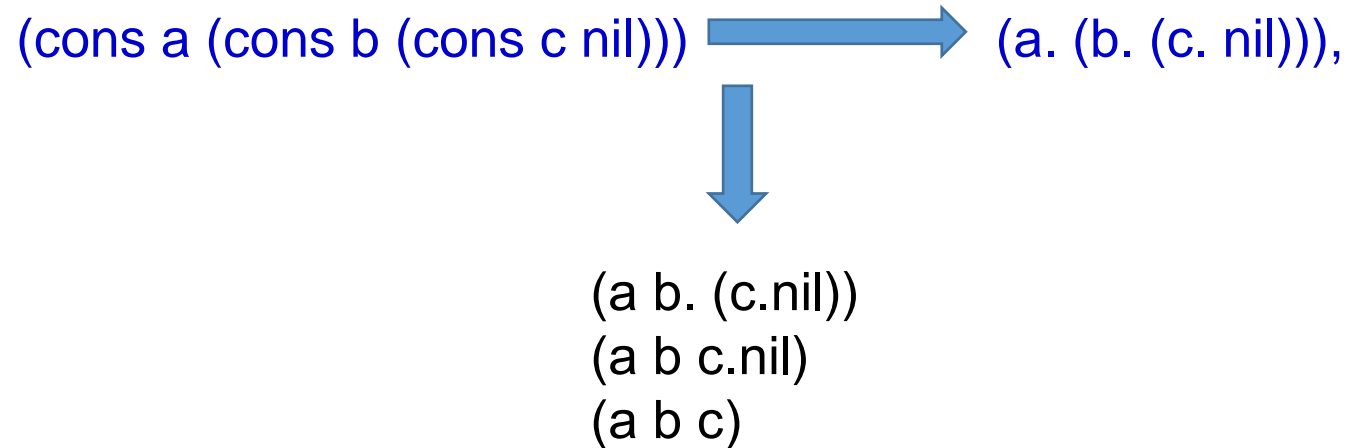
- ❑ точка, за якою безпосередньо слідує дужка, що відкривається, може бути опущена, так само опускаються і дужки:

(**v1. (v2. (v3 ... (v<sub>k</sub>. v<sub>k+1</sub>)...)**) записується у вигляді (**v1 v2 ...v<sub>k</sub>.v<sub>k+1</sub>**)

- ❑ точка, за якою безпосередньо слідує атом **nil**, також може бути опущена, сам атом nil теж можна опустити:

(**v1 v2 ... v<sub>k</sub>.nil**) записується у вигляді (**v1 v2 ... v<sub>k</sub>**)

# Точкові пари



Точкова запис використовується в тих випадках, коли, наприклад, необхідно послатися на перші два елементи списку довільної довжини.

Якщо є запис **(x1 x2.y)**,

**x1** позначає перший елемент,

**x2** - другий,

**y** – список, що залишився

# Процедури для робот з точковими парами та списками

№	Процедура	Приклад	Результат	Примітка
1	Створення точкової пари	(cons 1 2)	(1 . 2)	Другий елемент пари - не список
2	Створення списку	(list 1 2 3)	(1 2 3)	
3	Створення списку	'(1 2)	(1 2)	короткий запис
4	Доступ до першого елемента пари (списку)	(car (cons 1 2))	1	
5	Доступ до другого елемента пари	(cdr (cons 1 2))	2	
6	Доступ до хвоста списку	(cdr (list 1 2))	(2)	cdr для списку повертає список
7	Пустий список?	(null? '())	#t	
8	Пошук елемента в списку	(member 2 (list 1 2 3))	(2 3 4)	повертає список від знайденого елемента, інакше #f



# Процедури для робот з точковими парами та списками

№	Процедура	Приклад	Результат	Примітка
9	Перестановка (інвертування) списку	<code>(reverse '(2 3 4))</code>	<code>(4 3 2)</code>	
10	Злиття списків	<code>(append '(1 2) '(3 4))</code>	<code>(1 2 3 4)</code>	
11	Застосування процедури до списку	<code>(map (lambda (x) (+ x 1)) '(1 2))</code>	<code>(2 3)</code>	для кожного елемента списку викликається процедура
12	Сортування елементів списку	<code>(sort '(3 6 2) &lt;)</code>	<code>(2 3 6)</code>	Другий параметр - критерій сортування

# Операції зі списками. Доступ до елементів

Розглянемо приклад процедури **list-ref**, яка бере в якості аргументів список і число **n** і повертає **n**-й елемент списку. Зазвичай елементи списку нумерують, починаючи з 0.

## Метод обчислення list-reference :

- ❑ Якщо **n = 0**, **list-ref** повинна повернути **car** списку.
- ❑ В інших випадках **list-ref** повернутає **(n - 1)** -й елемент від **cdr** списку.

```
;створення списку  
(define squares (list 1 4 9 16 25))  
  
;вибір n-го елементу списку  
(define (list-reference items n)  
  (if (= n 0)  
      (car items)  
      (list-reference (cdr items) (- n 1))))
```

```
(list-reference squares 3)  
16
```

# Операції зі списками. Кількість елементів. Пустий список

1. Перевірити, чи є аргумент **пустим списком** – предикат **null?**
2. Визначення **кількості елементів** списку – процедура **len**

Процедура **length** реалізує просту рекурсивну схему. Крок редукції такий:

- ❑ Довжина будь-якого списку дорівнює 1 плюс довжина **cdr** цього списку
- ❑ Цей крок послідовно застосовується, поки не досягнуто базового випадку:
- ❑ Довжина порожнього списку дорівнює 0.

```
(define (len items)
  (if (null? items) ; якщо список пустий
      0
      (+ 1 (len (cdr items)))) ; порахувати кількість
```

```
(define odds (list 1 3 5 7)) ; створити список
```

Виклик процедури

```
(len odds)
4
```

# Операції зі списками. З'єднання списків

**Задача:** створити список з елементів двох інших списків, використавши їх в якості аргументів.

## Алгоритм з'єднання списків **list1** і **list2**:

1. Якщо список **list1** порожній, то результатом є список **list2**.
2. В іншому випадку, потрібно сконструювати новий список, додавши на початок другого списку елементи першого:
  - 2.1. Для цього за допомогою **cons** конструюється список,
  - 2.2. Вибирається перший елемент **list1** за допомогою **car**
  - 2.3. Додається елементи, починаючи з другого до **list2** за допомогою **cdr** від **list1**

```
(define odds (list 1 3 5 7)) ; створити списки  
(define squares (list 1 4 9 16 25))
```

```
(define (append list1 list2)  
  (if (null? list1)  
      list2  
      (cons (car list1)  
            (append (cdr list1) list2))))
```

Виклик процедури **append** :

```
(append squares odds)  
(1 4 9 16 25 1 3 5 7)  
(append odds squares)  
(1 3 5 7 1 4 9 16 25)
```

В **DrRacket** замість **append** слід записати іншу назву, наприклад, **app**.

# Операції зі списками. Перетворення списків

**Задача.** Заданий список. Необхідно застосувати перетворення до кожного елементу списку і отримати новий список результатів.

Наприклад, наступна процедура **множить кожен елемент списку на задане число.**

```
(define (scale-list items factor)
  (if (null? items)
      0
      (cons (* (car items) factor)
            (scale-list (cdr items)
                        factor)))))
```


Виклик процедури scale-list

```
(scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

# Операції зі списками. Перетворення списків

## Застосування процедури вищого порядку

Процедура вищого порядку **map** бере в якості аргументів процедуру від одного аргументу і список, а повертає список результатів, застосувачи процедуру від одного аргументу до кожного елементу списку:



```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
              (map proc (cdr items))))))
```

В DrRacket замість **map**  
слід писати **mmap**,  
Замість **nil** слід писати **0**

Виклик процедури **map**

```
(map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)

(map (lambda (x) (* x x))
     (list 1 2 3 4))
(1 4 9 16); 1^2 2^2 3^2 4^2
```

Виклик процедури **map**

```
(map + (list 1 2 3) (list 40 50 60) (list 700 800 900))
(741 852 963)

(map (lambda (x y) (+ x (* 2 y)))
     (list 1 2 3)
     (list 4 5 6))
(9 12 15); 1+4*2 2+5*2 3+6*2
```

# Операції зі списками. Перетворення списків

## Застосування процедури вищого порядку.

Процедура вищого порядку **map** бере в якості аргументів процедуру від одного аргументу і список, а повертає список результатів, застосувавши процедуру до кожного елементу списку:

```
(define (map proc items)
  (if (null? items)
      nil
      (cons (proc (car items))
              (map proc (cdr items)))))
```

Можна дати нове визначення **scale-list** через **map**:

```
(define (scale-list items factor)
  (if (null? items)
      nil
      (cons (* (car items) factor)
              (scale-list (cdr items)
                           factor)))))
```

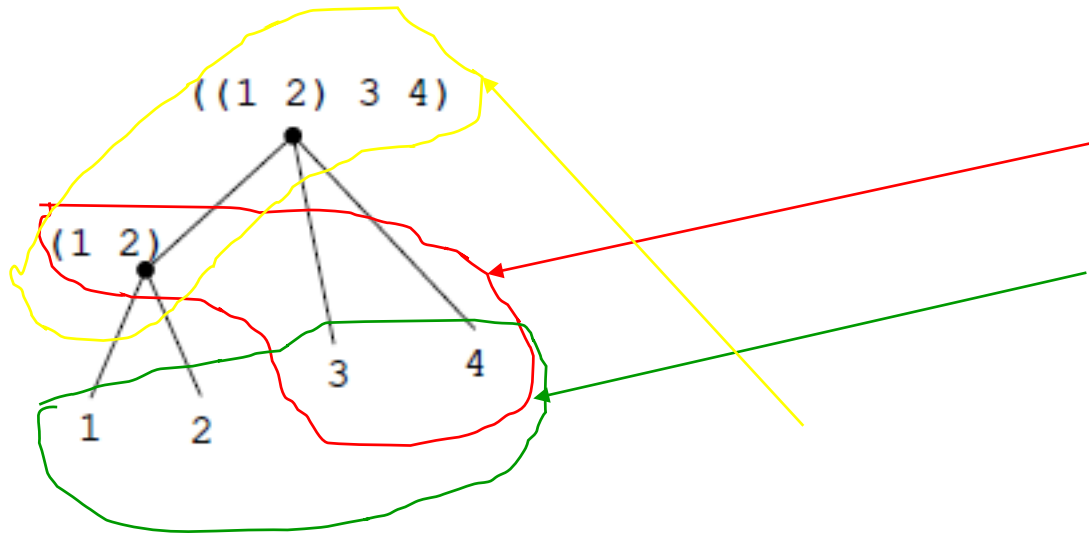


```
(define (scale-list items factor)
  (map (lambda (x)
        (* x factor))
        items))
```

# Ієрархічні структури - дерева

Подання послідовностей у вигляді списків природно поширити на послідовності, елементи яких самі можуть бути послідовностями.

Наприклад, можна розглядати об'єкт `((1 2) 3 4)`, отриманий за допомогою `(cons (list 1 2) (list 3 4))` як список з трьома членами, перший з яких сам є списком



```
(define x (cons (list 1 2) (list 3 4)))
```

```
(length x)
```

3

```
(count-leaves x)
```

4

```
(list x x)
```

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

```
(length (list x x))
```

2

```
(count-leaves (list x x))
```

8

Можна побудувати послідовності послідовностей - **дерева (trees)**.

Елементи послідовності є **гілками** дерева, а елементи, які самі по собі послідовності - **піддеревами**



# Ієрархічні структури - дерева

```
(define x (cons (list 1 2)
                (list 3 4)))
```

```
(length x)
```

```
3
```

```
(count-leaves x)
```

```
4
```

```
(list x x)
```

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

```
(length (list x x))
```

```
2
```

```
(count-leaves (list x x))
```

```
8
```

Рекурсивна схема обчислення довжини дерева **length**:

- ❑ Довжина списку  $x$  є 1 плюс довжина  $\text{cdr}$  від  $x$ .
- ❑ Довжина порожнього списку є 0.

Рекурсивна схема обчислення кількості лисків дерева **count-leaves**

має врахувати, що **car** сам по собі може бути деревом, листя якого потрібно порахувати.

Таким чином, крок редукції такий :

- ❑ Значення для порожнього списку **count-leaves** дорівнює 0.
- ❑ **count-leaves** від дерева  $x$  є **count-leaves** від  $(\text{car } x)$  плюс **count-leaves** від  $(\text{cdr } x)$ .
- ❑ обчислюючи **car**, досягаємо листя, для якого базовий випадок **count-leaves** від листа дорівнює 1.

# Ієрархічні структури - дерева

Елементарний предикат `pair?` перевіряє, чи є його аргумент парою.  
Розрахунок кількості листків дерева з предикатом `pair?` записаний так:

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                  (count-leaves (cdr x))))))
```

# Подання дерев

Подібно до того, як `map` може служити абстракцією для роботи з послідовностями, `map`, **поєднана з рекурсією, служить абстракцією для роботи з деревами.**

Наприклад, процедура `scale-tree` приймає в якості аргументу числовий множник і дерево, листям якого є числа. Вона повертає дерево тієї самої форми, де кожне число помножене на множник. Рекурсивна схема `scale-tree` схожа на схему `count-leaves`

```
(define (scale-tree tree factor)
  (cond ((null? tree) 0)
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                      (scale-tree (cdr tree) factor))))))
```

Виклик процедури

```
(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)) 10)
(10 (20 (30 40 . 0) 50 . 0) (60 70 . 0) . 0)
```

# Подання дерев

Інший спосіб реалізації `scale-tree` полягає в тому, щоб розглядати дерево як послідовність піддерев і використовувати `map`.

Ми відображаємо послідовність, масштабуючи по черзі кожне піддерево, і повертаємо список результатів.

У базовому випадку, коли дерево є листом, ми просто множимо:

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
        (if (pair? sub-tree)
            (scale-tree sub-tree factor)
            (* sub-tree factor)))
       tree))
```

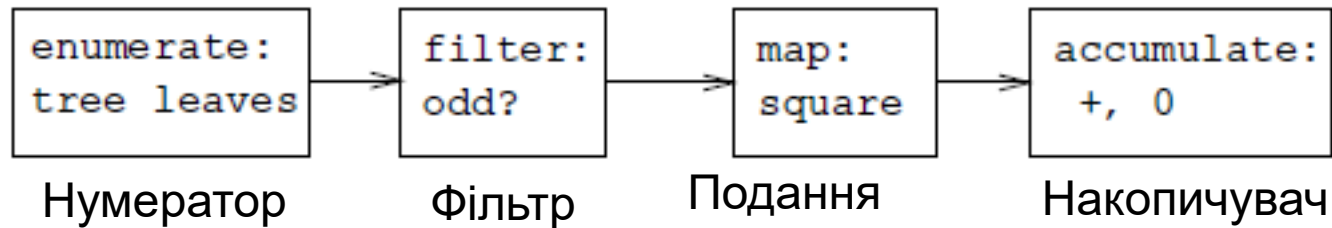
Виклик `scale-tree`:

```
(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7)) 10)
(10 (20 (30 40) 50) (60 70))
```

# Послідовності як стандартні інтерфейси

Розглянемо процедуру, яка приймає в якості аргументу дерево і обчислює суму квадратів тих з його листків, які є непарними числами:

## Діаграма потоку сигналів



## Алгоритм процедури :

1. Перерахувати листя дерева;
2. Просіяти їх, відбираючи непарні;
3. Звести в квадрат кожне з відібраних чисел;
4. Підсумувати квадрати чисел за допомогою процедури `+`, починаючи з початкової суми `= 0`.

# Послідовності як стандартні інтерфейси

## Алгоритм процедури :

1. Перерахувати листя дерева;
2. Просіяти їх, відбираючи непарні;
3. Звести в квадрат кожне з відібраних чисел;
4. Підсумувати квадрати чисел за допомогою процедури +, починаючи з 0.

Підрахунок листків в дереві здійснюється процедурою:

1. Кількість листів в порожньому дереві дорівнює нулю.
2. Кількість листів в дереві, що складається з єдиного атома, дорівнює одиниці.
3. Кількість листів в будь-якому іншому дереві може бути отримано як сума листя правого (**car**) і лівого (**cdr**) піддерев.

```
(define (enumerate-tree tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+(enumerate-tree (car tree))
                  (enumerate-tree (cdr tree))))))
```

## Виклик процедури

```
(enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

# Послідовності як стандартні інтерфейси

## Алгоритм процедури :

1. Перерахувати листя дерева;
2. **Просіяти листя дерева, відбираючи непарні;**
3. Звести в квадрат кожне з відібраних чисел;
4. Підсумувати квадрати чисел за допомогою процедури +, починаючи з 0.

**Просіювання** послідовності, що вибирає тільки ті елементи, які задовольняють даному предикату, можна записати за допомогою процедури:

```
(define (filter predicate sequence)
  (cond ((null? sequence) 0)
        ((predicate (car sequence))
         (cons (car sequence)
                (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

Виклик процедури

```
(filter odd? (list 1 2 3 4 5))
(1 3 5 . 0)
```

# Послідовності як стандартні інтерфейси

## Алгоритм процедури :

1. Перерахувати листя дерева;
2. Просіяти їх, відбираючи непарні;
3. **Звести в квадрат кожне з відібраних чисел;**
4. Підсумувати квадрати чисел за допомогою процедури +, починаючи з 0.

Відображення послідовності відібраних елементів

```
(define (square x)
  (* x x))

(define (mmap proc items)
  (if (null? items)
      0
      (cons (proc (car items))
            (mmap proc (cdr items)))))
```

mmap для DrRacket

## Виклик процедури

```
(mmap square (list 1 2 3 4 5))
(1 4 9 16 25 . 0)
```



# Послідовності як стандартні інтерфейси

## Алгоритм процедури :

1. Перерахувати листя дерева;
2. Просіяти їх, відбираючи непарні;
3. Звести в квадрат кожне з відібраних чисел;
4. Підсумувати квадрати чисел за допомогою процедури **+**, починаючи з **0**.

Накопичення можна записати процедурою:

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence))))))
```

Виклик процедури:

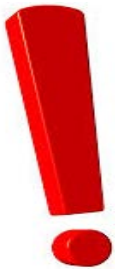
```
(accumulate + 0 (list 1 2 3 4 5))
15
(accumulate * 1 (list 1 2 3 4 5))
120
(accumulate cons nil (list 1 2 3 4 5))
(1 2 3 4 5)
```

для DrRacket замість nil  
писати 0

# Послідовності як стандартні інтерфейси. Операції

Переформулювати **sum-odd-squares** відповідно до діаграми потоку сигналів

```
(define (sum-odd-squares tree)
  (accumulate +
              0
              (map square
                    (filter odd?
                          (enumerate-tree tree))))))
```



Користь від виразу програм у вигляді операцій над послідовностями складається в тому, що ця стратегія допомагає будувати модульні проекти програм, тобто проекти, які виходять шляхом складання з відносно незалежних частин

# 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-Z-H-2.html#node\\_toc\\_start](http://www.r6rs.org/final/html/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%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](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](http://bydlokoder.ru/index.php?p=books_LISP)
5. *Кристиан Кеннек*. Интерпретация Лиспа и Scheme. Электронный ресурс. Режим доступа: <http://blog.ilammy.net/lisp/>

## 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>