ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

Лекція 8

РЕКУРСІЯ

2020

Повний текст лекції буде розміщений на сайті baklaniv.at.ua

Рейтінг THIOBE

30	Kotlin
31	Logo
32	Lisp
33	F#
34	Fortran
35	Lua
36	Ada
37	Prolog
38	VBScript
39	LabVIEW
40	Apex
41	Haskell
42	TypeScript
43	PowerShell
44	ML
45	RPG
46	Scheme
47	Erlang
48	Groovy
49	Bash
50	Julia

Apr 2020	Apr 2019	Change	Programming Language
1	1		Java
2	2		C
3	4	^	Python
4	3	v	C++
5	6	^	C#
6	5	v	Visual Basic
7	7		JavaScript
8	9	^	PHP
9	8	v	SQL
10	16	^^	R
11	19	^^	Swift
12	18	^^	Go
13	13		Ruby
14	10	vv	Assembly language
15	22	^^	PL/SQL
16	14	v	Perl
17	11	vv	Objective-C
18	12	vv	MATLAB
19	17	v	Classic Visual Basic
20	27	^^	Scratch

Position	Programming Language
21	SAS
22	Delphi
23	Dart
24	Transact-SQL
25	D
26	COBOL
27	Rust
28	Scala
29	ABAP

Проста рекурсія

Незважаючи на те, що в ліспоподібних мовах є механізми для організації циклічних процесів, все ж основний метод вирішення завдань залишається метод з використанням рекурсії, тобто із застосуванням рекурсивних функції.

Функція є рекурсивною, якщо в її визначенні міститься виклик цієї ж функції. Рекурсія є простою, якщо виклик функції зустрічається в деякій гілці лише один раз.

Простій рекурсії в процедурному програмуванні відповідає звичайний цикл. Наприклад, завдання знаходження значення факторіала n! зводиться до знаходження значення факторіала (n-1)! і множення знайденого значення на n.

Приклад: знаходження значення факторіала n!.

```
> (defun factorial (n)
        (cond
;факториал 0! равен 1
                ((= n 0) 1)
;факториал n! равен (n-1)!*n(
                (t (* (factorial (- n 1)) n))))
FACTORIAL
>(factorial 3)
6
```

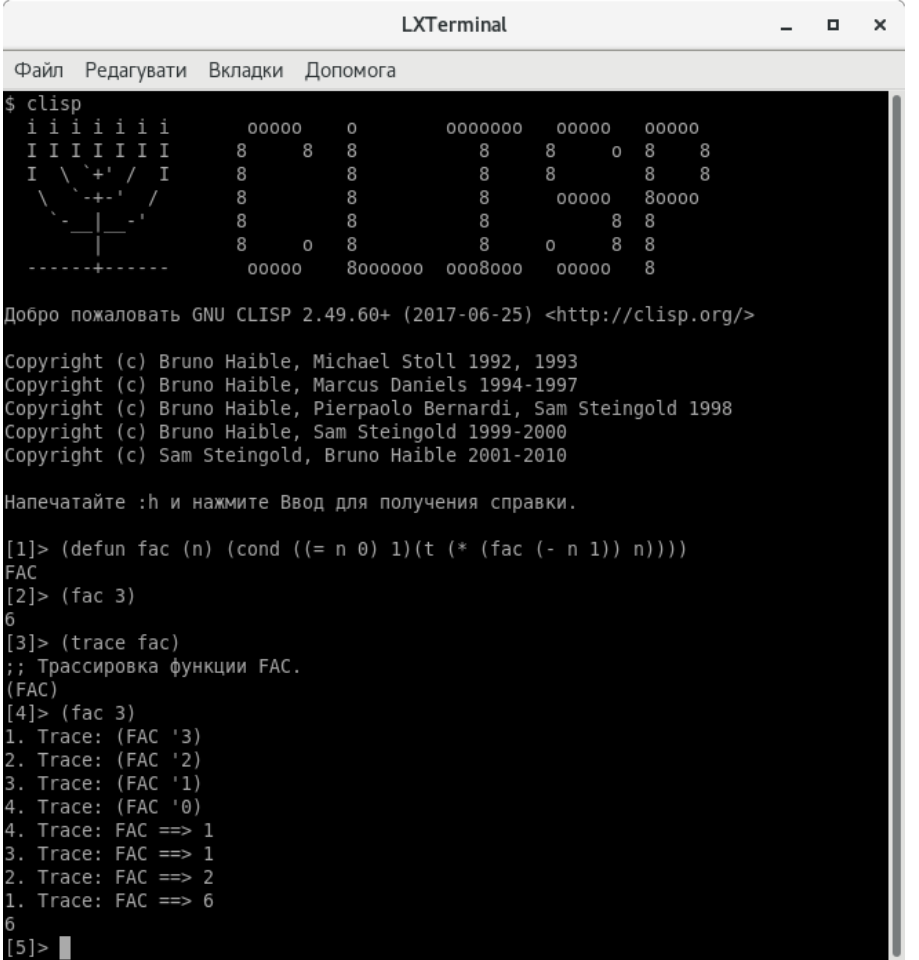
Для налагодження програми можна скористатися наявними можливостями трасування. Трасування дозволяє простежити процес знаходження рішення.

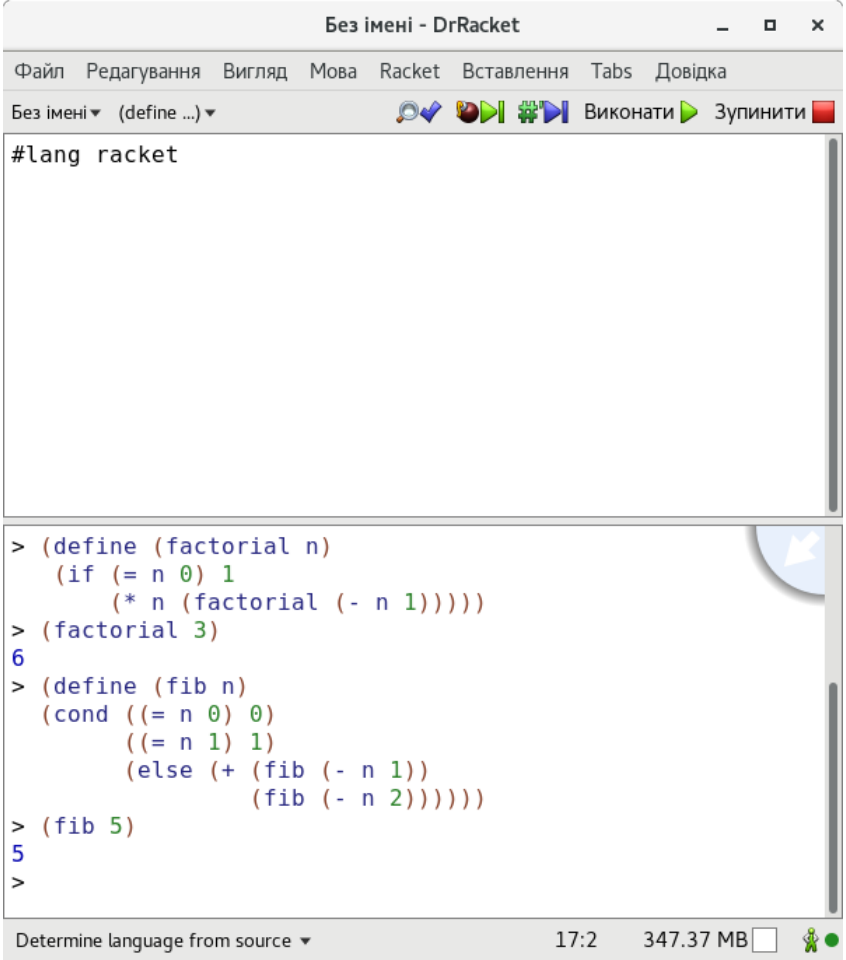
Для того щоб включити трасування, можна скористатися функцією trace:

Наприклад:

```
> (trace factorial)
(FACTORIAL)

> (factorial 3)
Entering: FACTORIAL, Argument list: (3)
        Entering: FACTORIAL, Argument list: (2)
                Entering: FACTORIAL, Argument list: (1)
                        Entering: FACTORIAL, Argument
list: (0)
                        Exiting: FACTORIAL, Value: 1
                Exiting: FACTORIAL, Value: 1
        Exiting: FACTORIAL, Value: 2
Exiting: FACTORIAL, Value: 6
6
```


LXTerminal
Файл Редагувати Вкладки Допомога
$ clisp
i i i i i i i ooooo o ooooooo ooooo ooooo
I I I I I I I 8 8 8 8 8 o 8 8
I \ `+' / I 8 8 8 8 8 8
\ `-+-' / 8 8 8 ooooo 8oooo
`-__|__-' 8 8 8 8 8
| 8 o 8 8 o 8 8
------+------ ooooo 8oooooo ooo8ooo ooooo 8
Добро пожаловать GNU CLISP 2.49.60+ (2017-06-25) <http://clisp.org/>
Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010
Напечатайте :h и нажмите Ввод для получения справки.
[1]> (defun fac (n) (cond ((= n 0) 1)(t (* (fac (- n 1)) n))))
FAC
[2]> (fac 3)
6
[3]> (trace fac)
;; Трассировка функции FAC.
(FAC)
[4]> (fac 3)
1. Trace: (FAC '3)
2. Trace: (FAC '2)
3. Trace: (FAC '1)
4. Trace: (FAC '0)
4. Trace: FAC ==> 1
3. Trace: FAC ==> 1
2. Trace: FAC ==> 2
1. Trace: FAC ==> 6
6
[5]>


Без імені - DrRacket
Файл Редагування Вигляд Мова Racket Вставлення Tabs Довідка
Без імені (define ...)
Виконати
Зупинити
#lang racket
> (define (factorial n)
(if (= n 0) 1
(* n (factorial (- n 1)))))
> (factorial 3)
6
> (define (fib n)
(cond ((= n 0) 0)
((= n 1) 1)
(else (+ (fib (- n 1))
(fib (- n 2))))))
> (fib 5)
5
>
Determine language from source
17:2
347.37 MB

Для відключення трасування можна скористатися функцією untrace:

Наприклад:

```
> (untrace factorial)
NIL
```

Трасування в мові Racket представлене на роліку

https://www.youtube.com/watch?v=-yNJZnZ6Nnw

Можна говорити про два види рекурсії: *рекурсія за значенням* і *рекурсія по аргументу*. *Рекурсія за значенням* визначається в разі, якщо рекурсивний виклик є виразом, що визначає результат функції. *Рекурсія по аргументу* існує в функції, повернене значення якої формує деяка нерекурсивна функція, в якості аргументу якої використовується рекурсивний виклик.

Наведений вище приклад рекурсивної функції обчислення факторіала є прикладом рекурсії по аргументу, так як повертається результат формує функція множення, як аргумент якої використовується рекурсивний виклик.

Ось кілька прикладів простої рекурсії.

Зведення числа x в ступінь n за допомогою множення (рекурсія по аргументу):

```
> (defun power (x n)
        (cond
;x0=1 (будь-яке число в нульовому ступені дорівнює
1)
                ((= n 0) 1)
;xn=x(n-1)*n (значення x в ступені n обчислюється
зведенням x в ступінь n-1
;й множення результату на n)
                (t (* (power (- n 1)) n))))
> (power 2 3)
8
> (power 10 0)
1
```

Копіювання списку (рекурсія по аргументу):

```
> (defun copy_list (list)
        (cond
; копією порожнього списку є порожній список
                ((null list) nil)
;копією непорожньої списку є список, отриманий з
голови і копії
;хвоста вхідного списку
                (t (cons (car list) (copy_list
(cdr list))))))
COPY_LIST
>(copy_list '(1 2 3))
(1 2 3)
>(copy_list ())
NIL
```

Визначення приналежності елемента списку (рекурсія за значенням):

```
> (defun member (el list)
        (cond
;список переглянутий до кінця, елемент не знайдений
                ((null list) nil)
;чергова голова списку дорівнює шуканого елементу, елемент знайдений
                ((equal el (car list)) t)
;якщо елемент не знайдений, продовжити його пошук в хвості списку
                (t (member el (cdr list)))))
MEMBER
```

```
> (member 2 '(1 2 3))
T
> (member 22 '(1 2 3))
NIL
```

З'єднання двох списків (рекурсія по аргументу):

```
> (defun append (list1 list2)
        (cond
;з'єднання порожнього списку і непорожньої дає
непорожній список
                ((null list1) list2)
;з'єднати голову першого списку і хвіст першого
списку,
;з'єднаний з другим списком
                (t (cons (car list1) (append (cdr
list1) list2)))))
APPEND
```

```
> (append '(1 2) '(3 4))
(1 2 3 4)
> (append '(1 2) ())
(1 2)
> (append () '(3 4))
(3 4)
> (append () ())
NIL
```

Реверс списку (рекурсія по аргументу):

```
> (defun reverse (list)
        (cond
;реверс порожнього списку дає порожній список
                ((null list) nil)
;з'єднати реверсувати хвіст списку і голову списку
                (t (append (reverse (cdr list))
(cons (car list) nil)))))
REVERSE
> (reverse '(one two three))
(THREE TWO ONE)
> (reverse ())
NIL
```

Інші види рекурсії

Рекурсію можна назвати простою, якщо в функції присутній лише один рекурсивний виклик. Таку рекурсию можна назвати ще рекурсією першого порядку. Але рекурсивний виклик може з'являтися в функції більш, ніж один раз.

У таких випадках можна виділити наступні види рекурсії:

1. паралельна рекурсія - тіло визначення функції function_1 містить виклик деякої функції function_2, кілька аргументів якої є рекурсивними викликами функції function_1.
   (defun function_1 … (function_2 … (function_1 …) … (function_1 …) …) …)

2.взаємна рекурсія - в тілі визначення функції function_1 викликається деяка функції function_2, яка, в свою чергу, містить виклик функції function_1.

(defun function_1 … (function_2 …) …) (defun function_2 … (function_1 …) …)

3.рекурсія вищого порядку - в тілі визначення функції аргументом рекурсивного виклику є рекурсивний виклик.

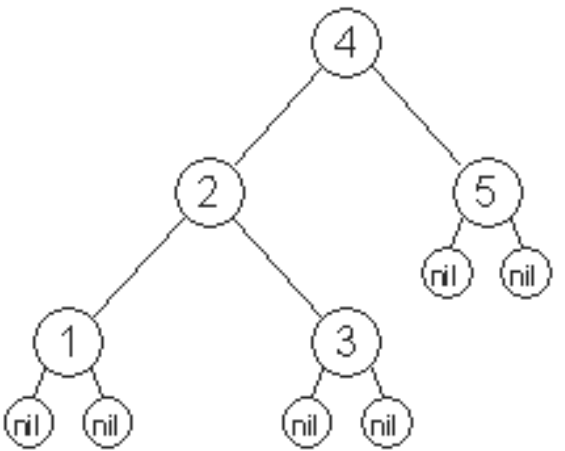
(defun function_1 … (function_1 … (function_1 …) …) …)

Розглянемо приклади паралельної рекурсії. У розділі, присвяченому простий рекурсії, вже розглядався приклад копіювання списку (функція copy_list), але ця функція не виконує копіювання елементів списку в разі, якщо вони є, в свою чергу також списками. Для запису функції, яка буде копіювати список в глибину, доведеться скористатися паралельної рекурсією.

```
> (defun full_copy_list (list)
        (cond
;копією порожнього списку є порожній
список
                ((null list) nil)
;копією елемента-атома є елемент-атом
                ((atom list) list)
; копією непорожньої списку є список,
отриманий з копії голови
; і копії хвоста вхідного списку
```

```
                    (t (cons
(full_copy_list (car list))
(full_copy_list (cdr list))))))
FULL_COPY_LIST
> (full_copy_list '(((1) 2) 3))
(((1) 2) 3)
> (full_copy_list ())
NIL
```

Не обійтися без паралельної рекурсії при роботі с бінарними деревами. Бінарне дерево, як і всі інші дані, представляються в Lisp'е у вигляді списків. Наприклад, впорядковане бінарне дерево



можна представити у вигляді списку

(4 (2 (1 nil nil) (3 nil nil)) (5 nil nil)).

Константи nil представляють порожні дерева. У такому поданні перший елемент списку - це вузол дерева, другий елемент списку - ліве піддерево і третій елемент списку - праве піддерево.

Інший варіант подання дерева -

(((nil 1 nil) 2 (nil 3 nil)) 4 (nil 5 nil)).

У такому поданні перший елемент списку - це ліве піддерево, другий елемент списку - вузол дерева і третій елемент списку - праве піддерево.

Можна використовувати і інші варіанти представлення дерев. Розглянемо простий приклад роботи з бінарним деревом - обхід дерева і підрахунок числа вузлів дерева. Для роботи з елементами дерева, які є, по суті, елементами списку, дуже зручно використовувати стандартні функції Lisp'a, для отримання першого, другого і третього елементів списку - fist, second і third, відповідно.

```
> (defun node_counter (tree)
        (cond
;кількість вузлів порожнього дерева дорівнює 0
                    ((null tree) 0)
; кількість вузлів непорожньої дерева складається з: одного кореня,
; кількості вузлів лівого піддерева і кількості вузлів правого піддерева
                    (t (+ 1 (node_counter (second tree)) (node_counter (third tree))))))
NODE_COUNTER
```

```
> (node_counter '(4 (2 (1 nil nil) (3 nil nil)) (5 nil nil)))
5
> (node_counter nil)
0
```

Приклад *взаємної рекурсії* - реверс списку. Так як рекурсія взаємна, в прикладі визначені дві функції: reverse и rearrange. Функція rearrange рекурсивна сама по собі.

```
> (defun reverse (list)
        (cond
                ((atom list) list)
                (t (rearrange list
nil)))))
REVERSE
```

```
> (defun rearrange (list result)
        (cond
                ((null list) result)
                (t (rearrange (cdr
list) (cons (reverse (car list))
result)))))
REARRANGE
> (reverse '(((1 2 3) 4 5) 6 7))
(7 6 (5 4 (3 2 1)))
```

Приклад рекурсії більш високого порядку - другого. Класичний приклад *функції з рекурсією другого порядку* - функція Аккермана.

Функція Аккермана визначається наступним чином:

B (0, n) = n+1

B (m, 0) = B (m-1, 0)

B (m, n) = B (m-1, B (m, n-1))

где m>=0 и n>=0.

```
> (defun ackerman
        (cond
                ((= n 0) (+ n 1))
                ((= m 0) (ackerman (-
m 1) 1))
                (t (ackerman (- m 1)
(ackerman m (- n 1))))))
ACKERMAN
```

```
> (ackerman 2 2)
7
> (ackerman 2 3)
9
> (ackerman 3 2)
29
```