Курси з мови програмування Lisp

ЛЕКЦІЯ 2

Види рекурсії

Проста рекурсія

```
(defun factorial (n)
  "Calculate factorial of n"
  (if (<= n 1)
      1
      (* n (factorial (1- n)))))
```

Паралельна рекурсія

```
(defun foo (...)
    (bar ... (foo ...) ... (foo ...) ...) ...)
```

Види рекурсії

Взаємна рекусія

```
(defun foo (...)
    (bar ...)  ...)
(defun bar (...)
    (foo ...)  ...)
```

Рекурсія вищого порядку

```
(defun foo (...)
    (foo ... (foo ...) ...)  ...)
```

Види рекурсії

Не хвостова рекурсія

```
(defun my-copy-list (lst)
   (when lst
      (cons (first lst) (my-copy-list (rest lst)))))
```

Види рекурсії

Хвостова рекурсія

```
(defun my-copy-list (lst)
  (when (typep lst 'list)
    (when lst
      (copy-list-inner (rest lst)
                       (cons (first lst) nil)))))
(defun copy-list-inner (lst res)
  (if lst
      (copy-list-inner
        (rest lst)
        (nconc res (cons (first lst) nil)))
      res))
```

Символ-значення і символ-функція

(SYMBOL-VALUE s-вираз-аргумент) -> s-вираз — значення, асоційоване з символом, що є результатом виконання s-вираз-аргумент

(set 'lst '(a b c))

(set 'b '(1 2 3))

(symbol-value (second lst)) -> (1 2 3)

(BOUNDP символ) -> T / NIL

(boundp 'w) -> NIL

(boubdp t) == (boundp 't) -> T

(SYMBOL-FUNCTION s-вираз) -> function

(symbol-function 'my-list) -> #<FUNCTION MY-LIST>

(FBOUNDP s-вираз) -> T / NIL

(fboundp 'my-list) -> T

Символ-значення і символ-функція

```
(defun my-list (x y)
      (cons x (cons y nil)))

(set 'my-list '(1))

(symbol-value 'my-list) -> (1)

(symbol-function 'my-list) -> #<FUNCTION MY-LIST>

(my-list my-list 2) -> ((1) 2)
```

Функціональний об’єкт і лямбда-вираз

defun — поєднує символ з певним функціональним об’єктом.

Неіменований функціональний об’єкт можна створити за допомогою lambda-виразу:

(LAMBDA (аргументи)

форма*)

(lamba (x) (1+ x)) -> #<FUNCTION (LAMBDA ...>

Застосування lambda-виразу з фактичними аргументами:

(lambda-вираз фактичні-параметри)

((lambda (a b) (cons a (cons b nil))) 1 2) ==> (1 2)

((lambda (a) ((lambda (b) (cons a (cons b nil))) 2)) 1) ==> (1 2)

Символ-значення і символ-функція

(SETF s-вираз s-вираз) ;; SETF (SET Field) — псевдофункція

(setq x y) == (setf x y)

(set x y) == (setf (symbol-value x) y)

(setf (symbol-value 'my-list) '(2))

(setf (symbol-function 'my-list) (lambda (x) (cons x my-list)))

(my-list my-list) -> ((2) 2)

Замикання

Замикання (clojure) — це функція і контекст її визначення.

```
(let ((b 5)
      (c 10))
   (defun adder (arg)
      (+ arg b c)))
```

(adder 5) -> 20

Написати код, який збереже у змінній calc-line-fn замикання, що приймає один аргумент x і повертає відповідне йому значення на прямій kx+b. k та b визначаються в контексті let.

Функції вищих порядків

Оскільки функція представлена функціональним об’єктом, вона може бути використана як дані й передана як аргумент в іншу функцію.

(FUNCTION eql) == #'eql

(member '(1) '((2) (1)) :test #'equal) -> ((1))

(member 1 '((2) (1)) :key #'first) -> ((1))

Визначити, чи є у списку підсписок, голова якого більша за нуль і

1) повернути його;

2) повернути список, з цим елементом у голові;

3) повернути список без таких елементів.

Вхідний список — ((-1) (0) (1)) — всі елементи — підсписки.

Функції вищих порядків

(MEMBER-IF / MEMER-IF-NOT функція-предикат список)

(REMOVE-IF / REMOVE-IF-NOT функція-предикат список)

(FIND-IF / FIND-IF-NOT функція-предикат список)

(COUNT-IF / COUNT-IF-NOT функція-предикат список)

(POSITION-IF / POSITION-IF-NOT функція-предикат список)

Перепишіть одне з попередніх завдань з використанням *-IF функції.

Виклик функції за допомогою функціонального об’єкта

(FUNCALL функція s-expr*) -> результат виконання функції-аргументу

(funcall #'+ 1 2 3) -> 6

(funcall calc-line-fn 1) -> 9

(APPLY функція s-expr+) -> результат виконання функції-аргументу. Останній з s-expr+ - завжди список.

(apply #'max 3 2 '(4 1)) -> 4

Як можна переписати суматор із домашнього завдання за допомогою apply?

MAP* функції

(MAPCAR/MAPLIST функція список [список+]) -> новий список

```
(mapcar #'1+ '(1 2 3)) -> (2 3 4)
(maplist #'identity '(1 2 3))-> ((1 2 3) (2 3) (3))
```

(MAPC/MAPL функція список [список+]) -> старий список

```
(mapc #'1+ '(1 2 3)) -> (1 2 3)
```

MAP* функції

(MAPCAN/MAPCON функція список [список+]) -> список

```
(mapcan (lambda (x) (and (numberp x) (list x)))
        '(a 1 b c 3 4 d 5))
=> (1 3 4 5)

(setq lst '((1) (2) (3)))
(mapcan #'identity lst)
lst -> ((1 2 3) (2 3) (3))
```

Предикати вищого порядку

(EVERY функція список+) -> T/NIL

(every #'numberp '(1 2 3)) -> T (every #'numberp '(1 a 3)) -> NIL

(SOME **функція** список+) -> <результат виконання **функція** відмінний від NIL>/NIL

(some #'> '(1 2 3) '(1 2 3)) -> NIL

```
(some (lambda (x y)
        (or (> x y) (= y 3)))
      '(1 2 3) '(1 2 3)) -> T
```

(NOTANY функція список+)

Згортання списку

Згортання використовується для перетворення списку в один елемент (який може бути іншим списком).

(REDUCE функція список &key from-end initial-value ...)

функція – з двома аргументами — поточний елемент і акумулятор, порядок яких залежить від параметру from-end.

:from-end – T/NIL — прапорець обходу з кінця

:initial-value — початкове значення акумулятора. Якщо не вказане, за початкове береться перший або останній елемент списку, в залежності від напрямку обходу елементів.

Згортання списку

```
(reduce #'* '(1 2 3 4 5)) =>  120
(reduce #'append '((1) (2)) :initial-value '(i n i t)) =>  (I N I T 1 2)
(reduce (lambda (elem acc)
            (if (evenp elem)
                (cons (1+ elem) acc)
                acc))
        '(1 3 6 10 15)
        :initial-value nil
        :from-end t)
--> (7 11)
```

Локальні функції

(FLET (визначення-функції*) форма*) -> результат виконання останньої форми

```
(flet ((%inc (arg)
         (1+ arg)))
  (print 1)
  (print (%inc 1)))
```

(LABELS (визначення-функції*) форма*) -> результат виконання останньої форми.

Різниця між FLET та LABELS полягає у тому, що функції, визначені в LABELS можуть викликати одна одну (в т. ч. рекурсивно), чого не можуть функції, визначені у FLET.

Кілька результатів функції

(MULTIPLE-VALUE-BIND (символ*) форма-виклик
    форма*) -> результат виконання останньої з форм
(MULTIPLE-VALUE-LIST форма-виклик) -> список результатів
(VALUES s-expr+) -> кілька результатів

```
(defun split-odd-even (lst)
   (when lst
      (multiple-value-bind (odd even) (split-odd-even (rest lst))
         (if (oddp (first lst))
            (values (cons (first lst) odd) even)
            (values odd (cons (first lst) even))))))
```

Кілька результатів функції

```
(multiple-value-bind (odd even)
        (split-odd-even '(1 2 3 4 5 6 7))
    (format t "odd: ~A~%" odd)
    (format t "even: ~A~%" even)
    (multiple-value-list (split-odd-even '(1 2 3 4 5 6 7))))
--> ((1 3 5 7) (2 4 6))
```

Інші функції

(CONSTANTLY s-expr) -> функція

(COMPLEMENT функція) -> функція

(IDENTITY s-expr) -> результат s-expr (повертає аргумент)

Інші функції

Дестурктивні функції

(PUSH форма символ) -> список

(POP символ) -> s-вираз

(INCF s-вираз) -> нове значення

(DECF s-вираз) -> нове значення

(SETF (CAR список) s-вираз) -> результат виконання s-виразу

Інші функції

```
(let ((lst ()))
    (push 1 lst)                ;; -> (1)     ;; lst == (1)
    (push 2 lst)                ;; -> (2 1)   ;; lst == (2 1)
    (incf (second lst))         ;; -> 2       ;; lst == (2 2)
    (decf (first lst))          ;; -> 1       ;; lst == (1 2)
    (setf (car lst) (list 3 4)) ;; -> (3 4)   ;; lst == ((3 4) 2)
    (pop lst))                  ;; -> (3 4)   ;; lst == (2)
```

Цикл LOOP

Макрос LOOP

```
(let ((lst  '(a b c d)))
    (loop :for elem :in lst
          :for idx :from 1 :to 10
          :do (format t "~A -> ~A~%" idx elem)))
```

Завдання

1. Написати функцію compose, яка приймає на вхід список функцій і повертає функцію, що послідовно виконує кожну з функцій над результатом попередньої. Для першої функції аргумент передає користувач.

(setq composed-fn (compose '(1+ 1+)))

(funcall composed-fn 1) -> 3

2. Написати функцію my-flatten, яка перетворює список з підсписками у просто список з елементами на одному рівні.

(my-flatten '(1 (2) (3 (4)) 5)) -> (1 2 3 4 5)