Курси з мови програмування Lisp

ЛЕКЦІЯ 1

Особливості функціонального програмування і мови Lisp

- чистота функцій
- незмінюваність
- безтиповість
- простий синтаксис
- композиція і рекурсія

та інші.

Символи, числа і константи (атоми)

a, symbol, column12, ret-val

+ - * / @ $ % ^ & _ \ < > ~ ! ? [ ] .

|a b|, a\ b

1, 2, 3.0, #c(3, 4), 4/5

T, NIL

PI

Списки

(a b 3) ;; список з трьох атомів — двох символів і одного числа

(+ 2 (- 3 1)) ;; список з трьох елементів — символа, числа і одного
;; підсписку, що складається з одного символу і двох чисел

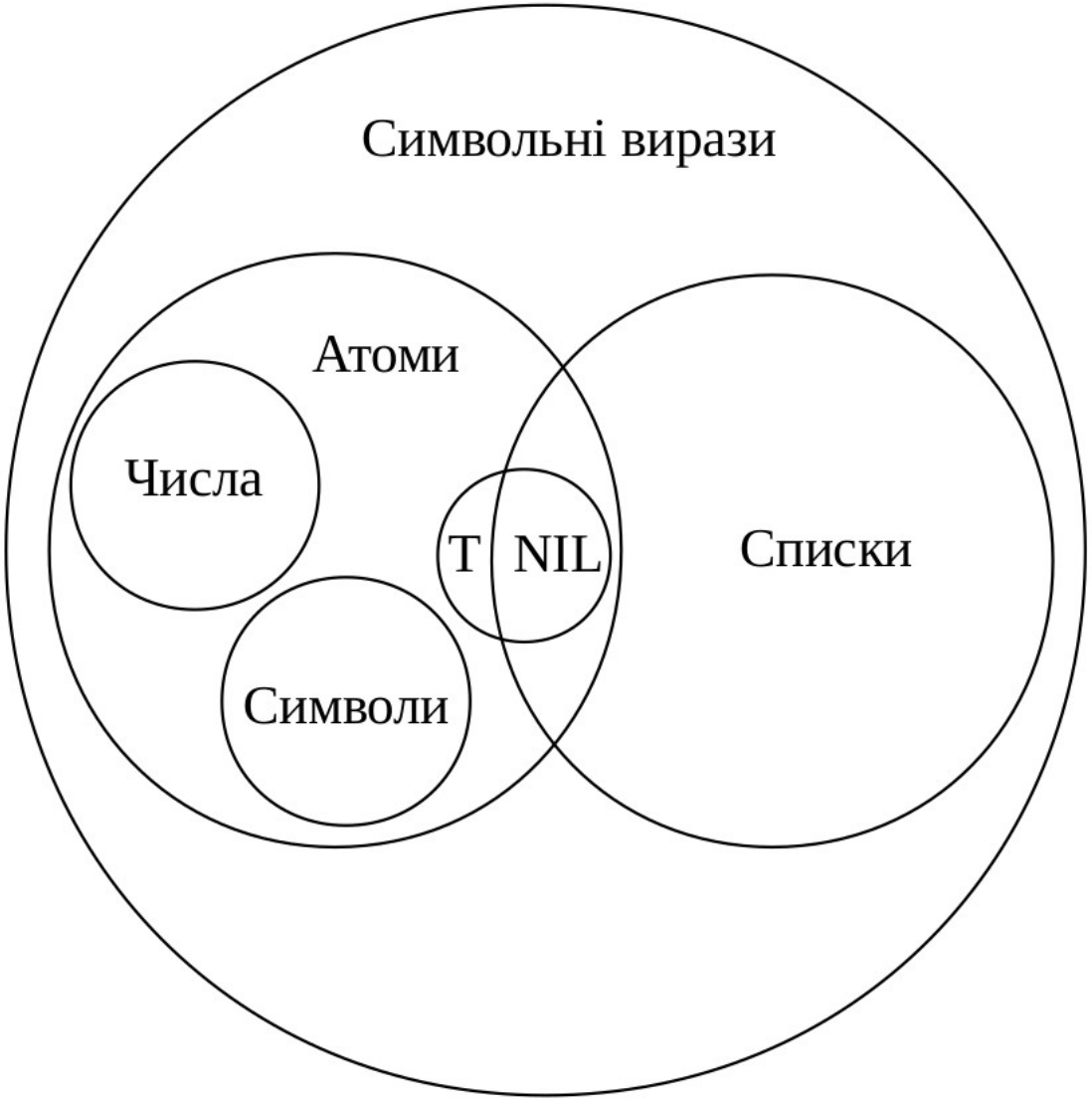
(((one) 2) three)

(), nil, NIL ;; різні записи одного і того ж символу nil

(NIL) ;; це список, що має один елемент — підсписок, який є пустим.

(NIL NIL T (NIL))

Символьні вирази (s-вираз , s-expression)



Дані у вигляді списку

```
(computer
    (processor
        (manufacturer Intel)
        (speed 5.0) ;; GHz
        (dimensions
            (height 2.91) ;; in
            (width 4.41) ;; in
            ...))
    (ram ...
```

Інтерпретація викликів

Алгоритм роботи інтерпретатора:

1. Введення s-виразу.
2. Виконання введеного s-виразу.
3. Виведення результату.

Інтерпретація викликів

Функції у Lisp

Java: Math.sqrt(9)

Lisp: (sqrt 9)

Java: 1 * 2 + 2 * 3

Lisp: (+ (* 1 2) (* 2 3))

Відміна обчислень

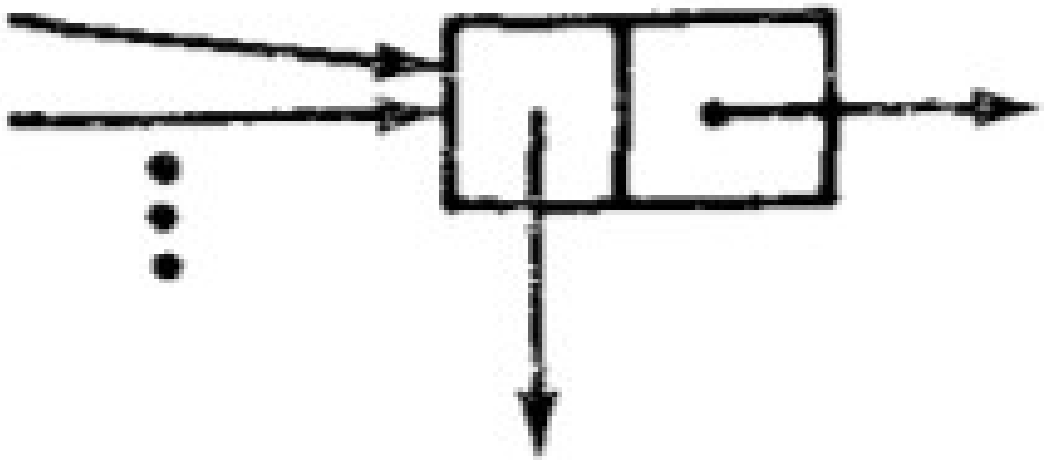
'(+ (* 1 2) (* 2 3)) <----> (quote (+ (* 1 2) (* 2 3)))

(+ '2 3) <----> (+ (quote 2) 3) <----> (+ 2 3) -> 5

Внутрішнє представлення списків

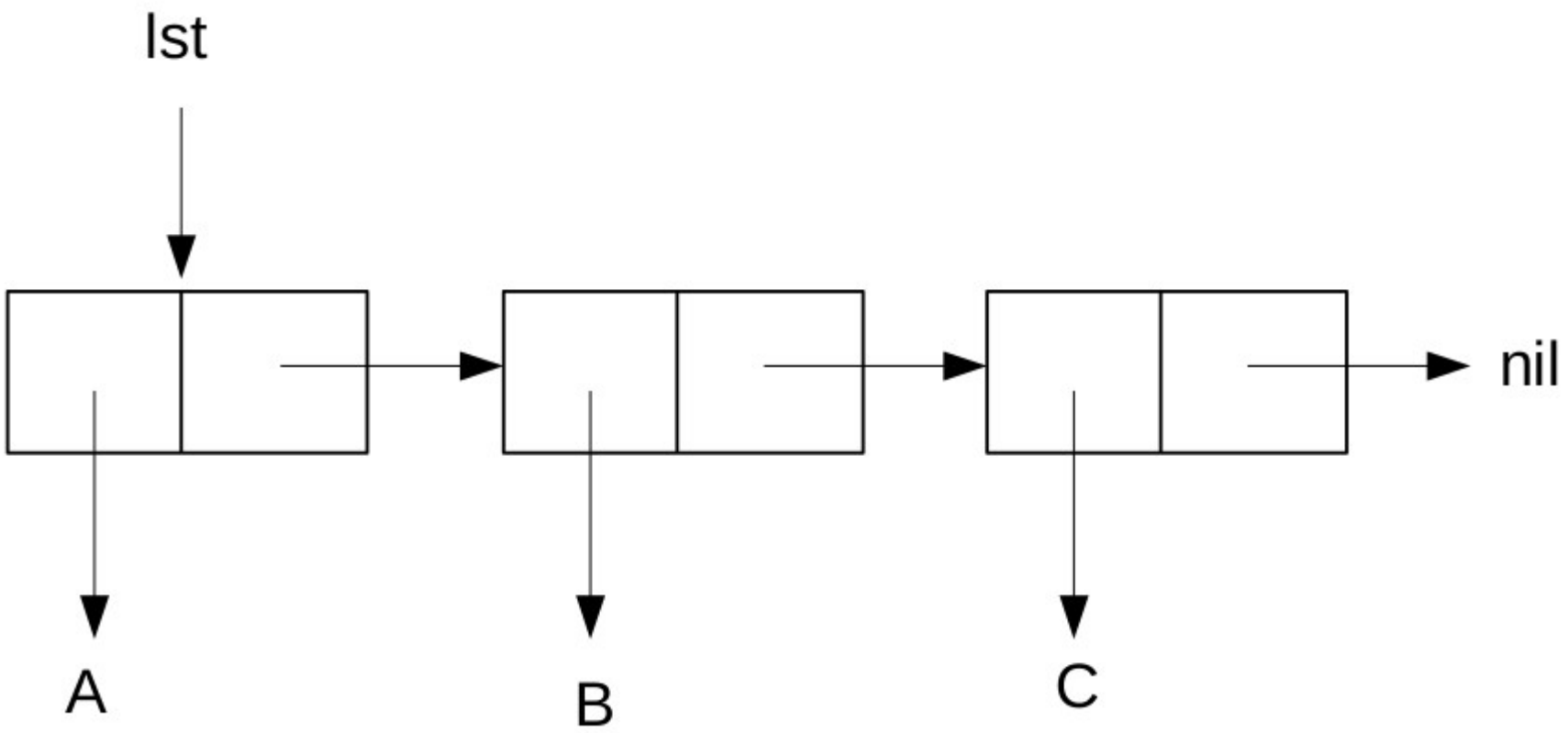
Спискова комірка

CAR CDR



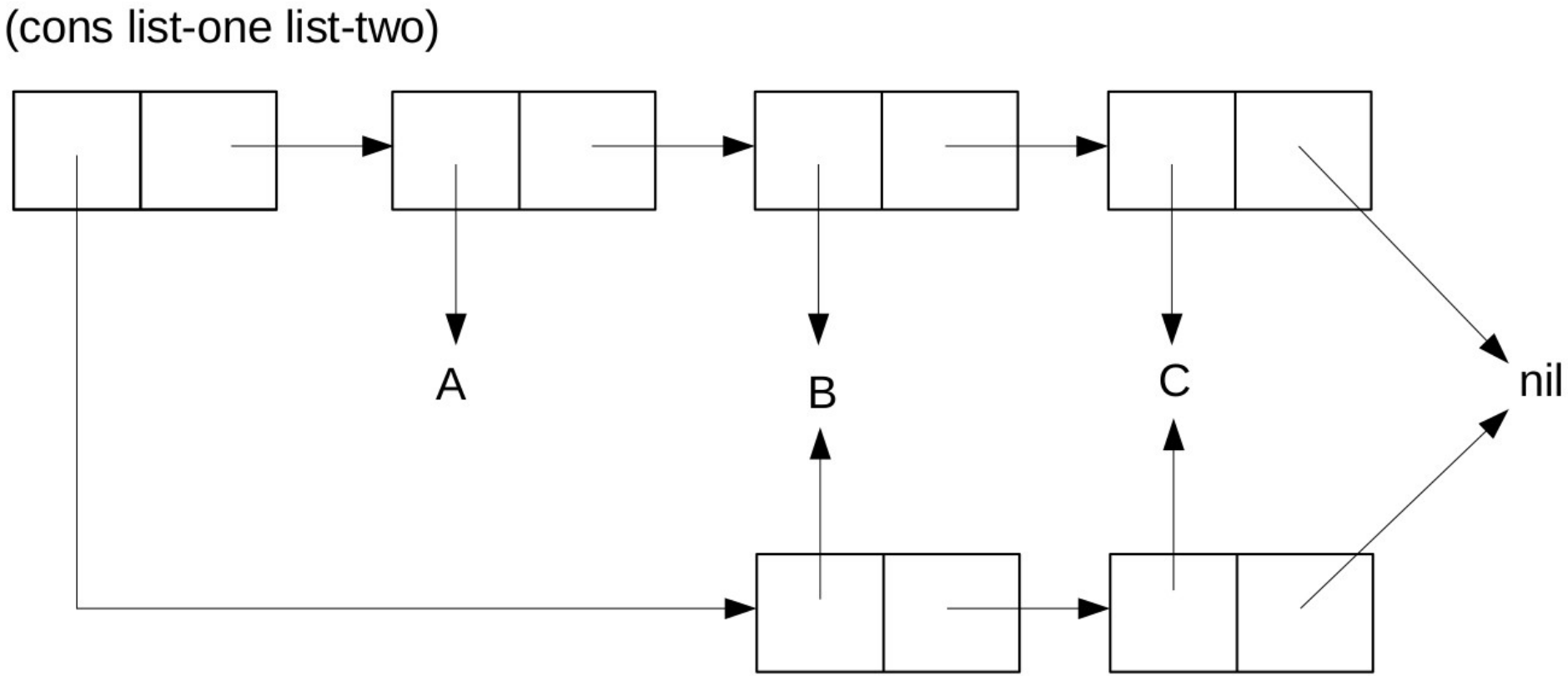
Внутрішнє представлення списків

lst → (a b c)



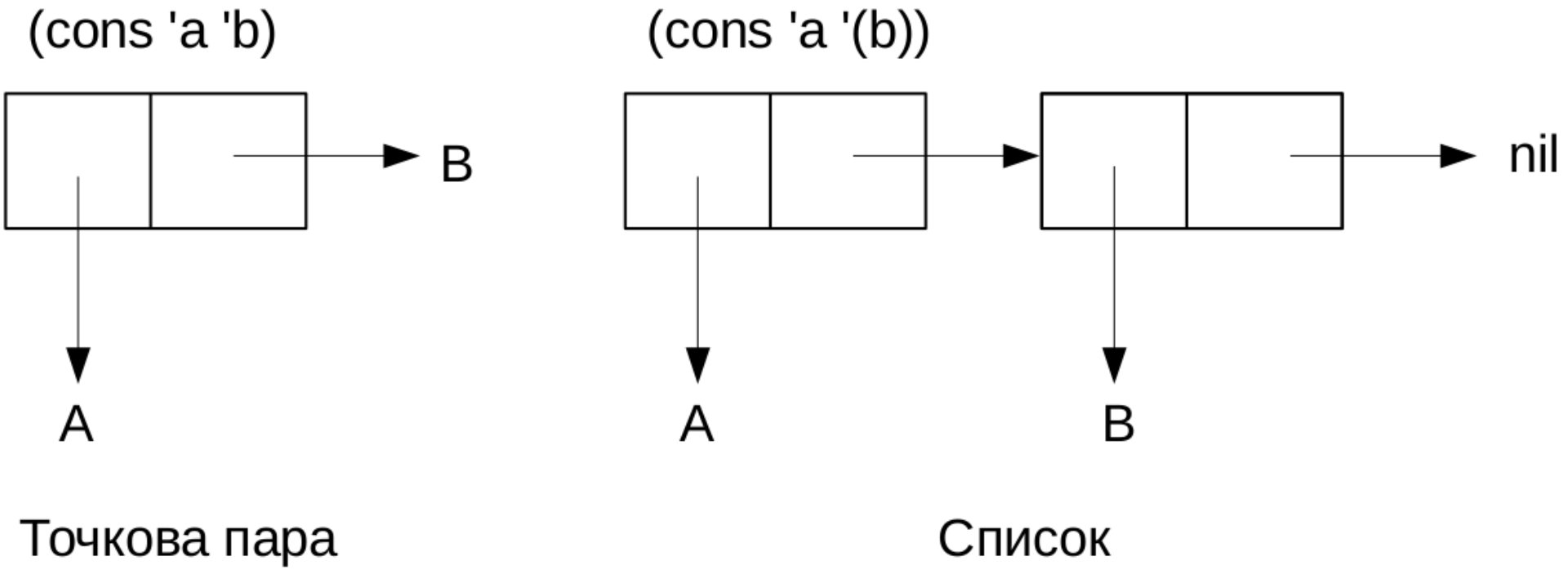
Внутрішнє представлення списків

lst → ((b c) a b c), де list-one → (b c), list-two → (a b c)



Точкова пара

(cons 'a 'b) → (A . B)



Базові функції роботи зі списками і предикати

(CAR список) -> s-вираз

(CDR список) -> список

(CONS s-вираз список) -> список

(ATOM s-вираз) -> T/NIL

(EQ символ символ) -> T/NIL

(EQ 1 1) --> T / NIL

(CAR '(1 2)) --> 1

(CDR '(1 2)) --> (2)

(CONS 1 '(2)) --> (1 2)

(ATOM 1) --> T

(EQ 'a 'a) --> T

(EQ 'a 'b) --> NIL

(car (cdr '(1 2 3))) → ?

(cons '(a) '(a b)) → ?

(cons nil nil) → ?

(cdr (car (cdr '(1 2 3)))) → ?

(atom (car '(1 2 3))) → ?

(eq (cdr (cdr '(1 2 3))) 3) → ?

Базові функції роботи зі списками і предикати

(car (cdr (cdr some-list))) == (caddr some-list)

В SBCL є всі варіації з 2-5 символами між c і r.

CAR == FIRST CADR == SECOND

CADDR == THIRD CDR == REST

(NTH число список) --> s-вираз

(NTHCDR число список) --> список

(second (rest '(a b c))) → ?

(rest (rest '(1 (2)))) → ?

(car (cddr '(a b c))) → ?

(nth (second '(a 2)) '(1 2 3 4)) → ?

(nthcdr 1 (cons 1 '(2 3))) → ?

(cadar '(((1) (2) (3)) (4 (5)))) → ?

Інші предикати та примітиви

Порівняння Lisp-об’єктів

(eql 3.0 3.0) -> T

(eql 3 3.0) -> NIL

(= 3 3.0) -> T

(numberp 1) -> T

(numberp t) -> NIL

(equal '(a b c) '(a b c)) -> T

(equal '(a b) '(c d)) -> NIL

(equal 3 3.0) -> NIL

(equalp 3 3.0) -> T

(eq 3 (cdr '(1 3))) → ?

(eql 2 (/ 6.0 3.0)) → ?

(eq 3 (cdr (cons 1 3))) → ?

(eql 2 (/ 6 3)) → ?

Інші предикати та примітиви

Логічні операції та робота зі списками

(not (numberp 1)) -> NIL

(not (not t)) -> T

(and (numberp 1) (listp nil)) -> T

(or a b)

(null nil) -> T

(null '(1)) -> NIL

(last '(a b c)) -> (c)

(list 'a 'b 'c) -> (a b c)

(list 'a (+ 1 2) '(+ 1 2)) -> (a 3 (+ 1 2))

Інші предикати та примітиви

Об’єднання списків

(append '(1 2) '(3) '(4 5)) -> (1 2 3 4 5)

append копіює всі списки-аргументи, окрім останнього.

Арифметичні операції

<, >, =, /=, <=, >=, +, -, *, /, rem

1+, 1- ;; інкремент, декремент

zerop, plusp, minusp

evenp, oddp

Завдання

1) записати у вигляді s-виразу (s-expression) наступний вираз і виконати його у інтерпретаторі:

1 + (4 / 2) – cos(π) * |-3| * 4

Примітка: для визначення модуля у Lisp є функція abs.

2) виконати в інтепретаторі записану в завданні 1 формулу попередньо її заквотувавши (відмінивши обчислення).

Значення символів

(SET s-вираз1 s-вираз2) -> s-вираз — результат виконання s-вираз2

(set 'a (+ 1 2)) -> 3

(set (car '(a b)) 'b) -> b ;; в символ a запишеться значення — символ b

(SETQ символ s-вираз) -> s-вираз

(setq a 1) == (set (quote a) 1) == (set 'a 1) -> 1

SET, SETQ (SET Quoted) — псевдофункції

Інтерпретація викликів

(eval '(+ 1 2)) --> 3

(eval (list 'print "Boo!")) --> "Boo!"

> "Boo!"

Завдання

1) Визначити змінні x, y і z, надавши їм значення 4, 3, 4 (з попереднього завдання). Обчислити формулу з попереднього завдання:

1 + (x / 2) – cos(π) * |-y| * z

Написати код, який перевірить результати виконання обох формул — з числами і змінними.

2) Написати код, що конструює список з трьох довільних елементів.

Перевірити на рівність список, отриманий таким конструюванням та список, який буде сконструйований таким самим s-виразом, але за допомогою eval.

Підказка: порівняти результат s-виразу з результатом (eval 's-вираз)

Конструктивні/деструктивні функції

Конструктивні функції:

cons, list, append

Деструктивні функції:

rplca, rplcd, nconc

Перерва 20 хвилин

Визначення функції

(DEFUN символ список-аргументів s-вираз*) -> символ

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

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

Визначте функції:

(double <x>) -> 2 * <x>

(negate <x>) -> -<x>

(square <x>) -> $<x>^2$

Передача параметрів у функції і область їх дії

Вільні змінні:

(setq x 100)

(defun f1 (x) (f2 x))

(defun f2 (y) (list x y))

(f1 2) -> (100 2)

Передача параметрів у функції і область їх дії

Глобальні та динамічні змінні:

(DEFVAR символ s-вираз)

(DEFPARAMETER символ s-вираз)

(defvar x 100)

(defun f1 (x) (f2 x))

(defun f2 (y) (list x y))

(f1 2) -> (2 2)

Правила іменування глобальних/динамічних змінних:

<ім’я> --> *x*

Робота з контекстом

(LET ((символ | (символ s-вираз))*)
    форма*)

```
(let (var
        (var-2 15))
  (print (null var))
  (print var-2))
```

Визначте значення виразу, встановивши змінні x, y за допомогою let:

$x^2 + y^2$

Робота з контекстом

```
(LET* ( ( символ | (символ s-вираз) )* )
    форма*)
(let* ((var 0)
          (var-2 (1+ var))
  (print var)
  (print var-2))
```

Визначте значення виразу, встановивши змінні x, y за допомогою let*:

$y^2 + 2y + 1$ де x = calc-x(), y = calc-y(x)

calc-x(): return 1; calc-y(x): return x+1

Для обчислень використовуйте визначені функції double та square.

Послідовне виконання

(PROG1 форма*) -> результат виконання 1 форми

(PROG2 форма*) -> результат виконання 2 форми

(PROGN форма*) -> результат виконання останньої форми

```
(progn
    (+ 1 2 3)
    (- 5 2)
    (* 2 2)) -> 4
```

Гілкування обчислень

(IF форма-умова T-форма [NIL-форма]) -> результат T-форми / NIL-форми в залежності від форми-умови

(if (numberp 1) 0)

(if (numberp a) 0 -1)

(if (numberp 'a) 0 -1)

```
(defun first-number-p (lst)
  (if (listp lst)
      (numberp (first lst))))
```

Гілкування обчислень

(WHEN форма-умова форма*) -> результат останньої форми або NIL

(UNLESS форма-умова форма*) -> аналогічно з WHEN

```
(when (eq a b)
   (print "1"))
(unless (not (eq a b))
   (print "ok"))
```

```
(defun first-number-p (lst)
  (when (listp lst)
    (numberp (first lst))))
```

Гілкування обчислень

(COND (форма-умова форма*)*) -> результат виконання останньої форми тієї гілки, для якої форма-умова повернула не NIL. Якщо форма окрім умови нема — повернеться результат виконання форми-умови.

```
(cond
    ((eql a 1) "match 1")
    (b)
    (t "default"))
```

Гілкування обчислень

(CASE форма ((значення/список-значень) форма*)* [(otherwise форма*)]) -> результат виконання останньої форми тієї гілки, для якої форма-умова повернула не NIL. Якщо форм окрім умови нема — повернеться NIL. Умова за замовчанням — символ otherwise / t.

```
(case a
    (2 "ok")
    (a)
    ((t nil) "t/nil")
    (otherwise "ok-default"))
```

Завдання

Написати функцію my-last, яка працює аналогічно до функції last і повертає список з останнім елементом списку-аргументу.

(defun my-last (lst) ...)

(my-last (list 1 2 3)) -> (3)

(my-last ()) -> NIL

Завдання

Написати функцію is-in-list, яка перевіряє чи є заданий елемент у списку і повертає T або NIL.

(defun is-in-list (elem lst) ...)

(is-in-list 'a '(b c d)) -> NIL

(is-in-list '2 (list 1 2 3)) -> T

Завдання

Написати функцію factorial, яка рахує факторіал числа.

(defun factorial (n) ...)

(factorial 1) -> 1

(factorial 2) -> 2

(factorial 3) -> 6

Цикли

```
(DO ((символ|(символ форма-значення [форма-крок]))*)
        (форма-умова форма*)
    форма*)
(do ((idx 0 (1+ idx)))
       ((>= idx 5))
   (format t "~A" idx))
```

DO* - агалогічно з LET*

Цикли

```
(DOLIST (символ-елемент форма-список [форма-результат])
    форма*)
(dolist (idx '(0 1 2 3 4))
    (format t "Idx: ~A~%" idx))

(DOTIMES (символ-елемент форма-кількість [форма-результат])
    Форма*)
(let (seq)
    (dotimes (idx 10 (reverse seq))
        (push idx seq)))
```

Інші функції

(LENGTH список) -> довжина списку, число

(length '(1 2 3)) -> 3 (length ()) -> 0

(REVERSE список) -> список

(reverse '(1 2 3)) -> (3 2 1)

(MEMBER s-expr список) -> список

(member b '(a b c)) -> (b c)

(REMOVE s-expr список) -> список

(remove nil '(1 t nil (nil))) -> (1 t (nil))

(REMOVE-DUPLICATES список) -> список

(remove-duplicates '(1 2 2 3 3 3 1)) -> (1 2 3)

Інші функції

(FIND s-expr список) -> елемент / NIL

(find 'c '(a b c)) -> c

(find nil '(a b c)) -> NIL

(find nil '(a nil c)) -> NIL ;; Інколи використання find м.б. неправильним

(COUNT s-expr список) -> число

(count 2 '(a 2 c 2 5 2)) -> 3

(POSITION s-expr список) -> індекс / NIL

(position t '(a b c)) -> NIL

(position 'a '(a b c)) -> 0

Інші функції

(BLOCK символ форма*)

(RETURN-FROM символ [форма])

(RETURN [форма])

```
(block foo
    (if (> x 0)
        (progn (when y (return-from foo "ok"))
                (setq x 0))
         y))
```

Домашнє завдання

1. Написати функцію sum, яка рахує суму чисел зі списку-аргументу.

(sum '(1 2 3)) -> 6

2. Написати функцію inc-lst, яка приймає на вхід список чисел і повертає новий список з числами, більшими на 1.

(inc-list '(1 2 3)) -> (2 3 4)

3. Написати функцію my-reverse, яка розвертає список у зворотному порядку (створює новий список).

(my-reverse '(1 2 3)) -> (3 2 1)

4. Написати функцію delete-repeats, яка видаляє дублікати елементів зі списку-аргументу. Елементи — числа і символи.

(delete-repeats '(1 1 2 3 3 3 a a b)) -> (1 2 3 a b)