

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Запорізький національний технічний університет

КОНСПЕКТ ЛЕКЦІЙ

з «Функціонального програмування»

частина 2

**для дисципліни «Функціональне та логічне
програмування»**

**призначено студентам професійного напрямку
6.050103 «Програмна інженерія»**

2016

Конспект лекцій з «Функціонального програмування» частина 2 для дисципліни «Функціональне та логічне програмування» призначено студентам професійного напрямку 6.050103 «Програмна інженерія» /Укл.: І.В. Левада, Л.Ю. Дейнега – Запоріжжя: ЗНТУ, 2016. – 98 с.

Укладачі: старший викладач І. В. Левада
старший викладач Л. Ю. Дейнега

Рецензент: проф., к. т. н. Г.В. Табунщик

Відповідальний за випуск: старший викладач І. В. Левада

Затверджено на
засіданні кафедри
Протокол № 2 від
4 березня 2016 р.

ЗМІСТ

ПЕРЕДМОВА	5
ТЕМА1: Послідовності.....	6
1.1 Функції роботи із послідовностями	6
1.1.1 Функції загального характеру	7
1.1.2 Функції відбору	10
1.1.3 Функції конструктори	18
1.1.4 Функції модифікатори послідовностей	20
ТЕМА2: Функції та макроси роботи із списками	25
2.1 Функції відбору.....	25
2.2 Функції конструктори	26
2.3 Функції модифікатори.....	30
ТЕМА3:Вектори	36
3.1 Загальні відомості про вектори	36
3.2 Спеціалізовані вектори - рядки. Функції роботи з рядками.....	38
3.2.1 Функції відбору знаків та рядків.....	38
3.2.2 Функції конструктори рядків	41
3.2.3 Функції перетворення рядків та знаків	42
3.2.4 Функції порівняння рядків та знаків	43
ТЕМА4: Символи. Застосування символів для обробки текстів	44
4.1 Чому слова подаються символами?	44
4.1 Символ. Властивості символів	45
ТЕМА 5: Числа.....	49
5.1 Класифікація чисел.....	49
5.2 Арифметичні функції	50
5.3 Функції транзитивного порівняння.....	52
5.3 Макроси модифікації чисел за місцем	53
5.4 Ірраціональні та трансцендентні функції	54
5.5 Функції округлення та усікання	55
ТЕМА 6: Змінні	59
6.1 Поняття змінної	59
6.2 Механізми дії лексичних змінних	61

6.3	Замикання.....	64
6.4	Динамічні змінні.....	67
6.5	Константи.....	70
ТЕМА 7: Керування роботою програми		71
7.1	Макроси керування роботою програми.....	71
7.2	Блочна структура функцій.....	74
8	Функції вводу виводу.....	75
8.1	Універсальні функції вводу-виводу.....	75
8.2	Функції вводу.....	79
8.3	Функції виводу.....	81
9	Файли та потокова організація роботи з ними.....	84
9.1	Поняття потоку.....	84
9.2	Робота з файлами через потоки.....	85
9.2.1	Увід-вивід через стандартні потоки.....	85
9.2.2	Функції обробки файлів.....	86
9.2.3	Робота над файлами.....	92
10	Функціонали.....	94
10.1	Застосовні функціонали.....	94
10.2	Відображуючі функціонали.....	95
10.3	Предикати планування.....	98

ПЕРЕДМОВА

Мова програмування PRACTICAL COMMON LISP один із діалектів мови програмування COMMON LISP. Через велику кількість діалектів мови COMMON LISP та їх реалізацій в цій мові виникло багато розбіжностей. Все це стимулювало роботу над створенням стандарту мови Common Lisp.

Перша гіпертекстова версія ANSI стандарту мови Common Lisp Hyper Spec (CLHS) для застосування з WEB-браузерами була підготовлена редактором комітету NCITS/J13 Кентом Пітманом у 1996 році. Для створення гіпертекстової версії стандарту він написав програму на мові Common Lisp. Гіпертекстова версія містить також інші відомості по стандартизації Common Lisp.

Наступна версія стандарту була переглянута Кентом Пітманом у 2005 році. Прагнення розроблювачів діалектів мови зробити їх конкурентоспроможними, примусило розвивати мову до мови загального застосування.

В серпні 1915 року групою розроблювачів GNU COMMON LISP була зроблена заява про випуск стабільної реалізації стандарту ANSI INCITS 226-1994 (R2004) промислової мови програмування загального призначення.

Сучасна загальна мова Common LISP підтримує функціональну та імперативні парадигми традиційно та об'єктна-орієнтовану парадигми. Вона має засоби для реалізації графічного інтерфейсу, роботи з мережами, базами даних, XML.

Створення конспекту лекцій з функціонального програмування частини 2 пов'язано з доповненням до лекцій сучасного матеріалу, розвиненої мови Common LISP. Конспект відповідає вмісту лекцій, що читаються у дисципліні «Функціональне та логічне програмування» і доповнює матеріал конспекту лекцій частина 1.

Сьогодні діалекти мови Common Лісп використовують для розробки програмного забезпечення на підприємствах, при дипломуванні, захисті кандидатських дисертацій.

Створення конспекту лекцій актуально як для студентів денного, так і для студентів заочного відділення.

ТЕМА1: Послідовності

1.1 Функції роботи із послідовностями

Стандартні типи даних мови CLISP являють собою ієрархічну структуру. Фрагмент структури подано на рисунку 1.1.

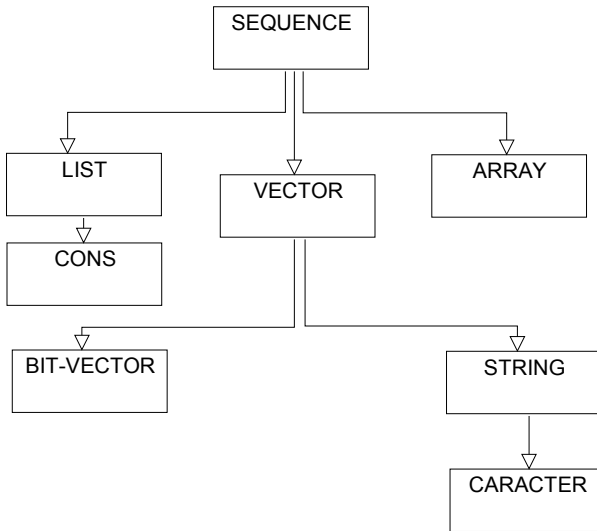


Рисунок 1.1-Фрагмент структури типів даних

Функції класифікуються за даними певного типу. Структуризація типів даних впливає на застосування функцій до даних певного типу. Функції типів можна застосовувати до даних підтипів.

Розглянемо тип SEQUENS. Тип SEQUENS має підтипи LIST, VECTOR, ARRAY. В той же час типи BIT-VECTOR, STRING є підтипами типу VECTOR. Внаслідок чого ці типи є також підтипами типу SEQUENS. Тобто, функції для роботи з послідовностями можна застосовувати для роботи із підтипами послідовностей.

Кожна група функцій, що класифікована за своїм типом даних, розподіляється на підгрупи за діями. Функції відбору одержують елементи даних за тестами, не змінюючи дані. Функції конструктори

будують нові дані на основі вхідних даних. Макроси - модифікатори змінюють існуючі дані.

1.1.1 Функції загального характеру

Тип послідовність (SEQUENCE) відноситься до абстрактних типів. Абстрактний тип дозволяє створити групу функцій спільну для його підтипів. Розглянемо такі функції.

Функція **LENGTH** повертає довжину даного, типу послідовність.

(LENGTH послідовність)

Приклад для списку:

```
CL-USER> (LENGTH '(1 3 6 8))
```

4

Приклад для рядка:

```
CL-USER> (LENGTH "common")
```

6

Приклад для вектора

```
CL-USER> (LENGTH #(2 E 6))
```

3

Приклад для бітового вектора:

```
CL-USER> (LENGTH #*10011110)
```

8

Функція **ELT** повертає елемент послідовності за індексом. Індекс рахується з нуля.

(ELT послідовність індекс)

Приклад для списку:

```
CL-USER> (ELT '(1 3 6 8) 2)
```

6

Приклад для рядку:

```
CL-USER> (ELT "common" 0)
```

#\с

Приклад для вектору:

CL-USER> (ELT #(2 Т 6) 1)

Т

Приклад для бінарного вектору:

CL-USER> (ELT #*10011110 7)

0

Макрос SETF записує елемент у послідовність за формою або заміщує послідовність на елемент і повертає цей елемент.

(SETF форма елемент)

Форма задається символом (ім'ям даного) або функцією, що застосовує символ. Важливо пам'ятати, що форма повертає посилку на дане. Тому замість форми можливо вказувати посилку на дане.

Приклад1: Посилка на список задається формою (ELT L 2). Заміщується елемент за індексом 2.

CL-USER> (SETQ L '(1 3 6 8))

(1 3 6 8)

CL-USER> (SETF (ELT L 2) 5)

5

CL-USER>L

(1 3 5 8)

Приклад2: Посилка на список задається формою (NTH 3 L). На місце 3-ого елементу записується число 10.

CL-USER>(SETQ L '(1 3 5 8))

(1 3 5 8)

CL-USER> (SETF (NTH 3 L) 10)

10

CL-USER>L

(1 3 5 10)

Приклад3: Посилка на рядок задається символом.

CL-USER> (SETQ L "data")

"data"

CL-USER> (SETF L 10)

10

Замість всього рядка записується число 10.

CL-USER>L

10

Приклад4: Рядок – вектор фіксованої довжини, тому не можна задавати індекс більший за довжину рядка. У прикладі довжина рядка 5, максимальний індекс 4, а пропонується записати за індексом 5.

CL-USER> (SETQ L "data1")

"data1"

CL-USER> (SETF (CHAR L 5) #3)

ERROR – індекс більше довжини рядка.

Для роботи із послідовностями важливим є функціонал **REDUCE**. Функціонали мають одним із своїх параметрів функцію, яку застосовують до інших своїх аргументів. Функціонал REDUCE застосовує вказану функцію до кожного елементу послідовності та попереднього результату і повертає останній результат.

Функціонал REDUCE має наступні ключі: INITIAL-VALUE (замовчання перший елемент послідовності), :START, :END, :KEY.

(REDUCE функція послідовність ключі із значеннями)

Приклад 1: Одержимо суму елементів списку.

CL-USER> (REDUCE '+ '(1 4 8))

13

Приклад 2: Треба злити внутрішні списки списку L..

```
CL-USER> (SETQ L '((1 3) (9 4) (6)))
((1 3) (9 4) (6))
```

```
CL-USER> (REDUCE 'APPEND L)
(1 3 9 4 6)
```

Функція APPEND зливає списки:

```
(APPEND (1 3) (9 4)) ==> (1 3 9 4)
```

```
(APPEND (1 3 9 4) (6)) ==> (1 3 9 4 6)
```

Приклад 3: Перетворимо рядок на список рядків, в кожному рядку одна буква. Застосуємо для цього свою функцію LST.

```
CL-USER> (DEFUN LST (A) (LIST (STRING A)))
LST
```

Функція LST одержує один знак з рядку і перетворює його на список з рядка. Функціонал REDUCE застосовує функцію LST до кожного знака рядка і зливає одержані списки функцією APPEND.

```
CL-USER> (REDUCE 'APPEND "ASDF" :KEY 'LST)
("A" "S" "D" "F")
```

1.1.2 Функції відбору

Функції відбору дозволяють виконувати дії над послідовностями без застосування рекурсії або циклів, що значно зменшує об'єм програми.

У параграфі розглядаються функції 2-х типів.

1. Елементи послідовності порівнюються за тестом з певним об'єктом.
2. Елементи тестуються за певним типом.

Важливою особливістю функцій мови CLISP у порівнянні із іншими мовами та ранніми діалектами мови LISP є розширення можливостей роботи стандартних функцій за рахунок застосування ключів. Ключі можуть змінити дії функції, діапазони пошуку тощо. Розглянемо дію ключів на прикладах роботи функцій.

Функція **COUNT** рахує кількість елементів послідовності, що порівнюванні з об'єктом за тестом і повертає цю кількість.

(COUNT об'єкт послідовність ключі_та_їх_значення)

У кожної функції може бути кілька ключів та відповідних їм значень. Для роботи функція COUNT може застосовувати наступні ключі: :TEST, :TEST-NOT, :KEY, :START, :END.

Тип тестування вказується ключами :TEST, :TEST-NOT. Значення цих ключів - тести. Тест обирається за типом елементів послідовності. По замовчанню тест 'EQL'.

Ключ :TEST має значенням тест, за яким порівнюються елементи послідовності з об'єктом. Якщо ключ :TEST-NOT, то елементи послідовності порівнюються з об'єктом за змістом протилежним вказаному тесту.

Ключ :START має значенням індекс послідовності, що вказує на початок вибору елементів. Ключ :END має значенням індекс, до якого треба вибирати елементи. Елемент з вказаним індексом не обирається.

Ключ :KEY має значенням ім'я функції, яку застосовують до кожного елементу перед тестуванням, а потім виконують тестування результату роботи функції. Призначення функції відібрати елементи для тестування.

Приклад 1: Задамо послідовність списком чисел. Функція рахує і повертає кількість чисел менших 5.

```
CL-USER> (COUNT 5 '(6 2 10 14) :TEST '<)
```

```
3
```

Елементи послідовності порівнюються з об'єктом за тестом '<' (менше).

Виконуються порівняння:

5<6 Лічильник 1

5<2 Лічильник 1

5<10 Лічильник 2

5<14 Лічильник = 3

Приклад2: Задамо послідовність списком символів. У прикладі за замовчанням значення тесту – 'EQL. В цьому випадку ключ :TEST не вказується.

```
CL-USER> (COUNT 'с '(C с а K) )
```

2

Приклад3: Елементами рядка є знаки. У прикладі знаки рядка порівнюються з об'єктом #\с за тестом протилежним тесту CHAR>.

```
CL-USER> (COUNT #\C "скас" :TEST-NOT 'CHAR>)
```

3

```
#\с <= #\с Лічильник 1
```

```
#\с <= #\k Лічильник 2
```

```
#\с <= #\a Лічильник 2
```

```
#\с <= #\с Лічильник 3
```

Приклад 4: У прикладі заданий список символів. За ключем :START розглядаються елементи послідовності починаючи з індексу 1. За ключем :END - розглядаються елементи послідовності до індексу 5. Елемент з індексом 5 не тестується.

```
CL-USER> (COUNT 'd '(D с D u r d) :START 1 :END 5)
```

1

Приклад 5: У прикладі послідовність задається списком, елементи якого теж списки.

```
CL-USER> (SETQ L '( (N 98) (A 96) (C 90) ) )
```

```
((N 98) (A 97) (C 90) ) )
```

```
CL-USER> (COUNT 97 L :TEST '< :KEY 'CADR)
```

1

Функція CADR застосовується до кожного елементу послідовності - списку, після чого виконується тестування результату:

(CADR '(N 98)) --> 98	97 < 98 Лічильник 1
(CADR '(A 96)) --> 96	97 < 96 Лічильник 1
(CADR '(C 90)) --> 90	97 < 90) Лічильник 1

Приклад 6: Функція COUNT рахує кількість елементів списку, які менше середнього арифметичного елементів числового списку.

У прикладі для обчислення середнього арифметичного елементів застосовується функція користувача AVERAGE.

```
CL-USER> (SETQ L '(4 7 8))
L
CL-USER> (DEFUN AVERAGE (L) (/ (REDUCE '+ L) (LENGTH
L)))
AVERAGE
CL-USER> (COUNT (AVERAGE L) L :TEST '<)
2
```

COUNT-IF рахує кількість елементів послідовності, що задовольняють тесту і повертає цю кількість. Тест задається першим аргументом без ключового слова. Функція має ключі :START, :END, :KEY, :FROM-END.

(COUNT-IF тест послідовність ключі_та_їх_значення)

Приклад 1: Функція рахує кількість парних елементів.

```
CL-USER> (COUNT-IF 'EVENP '(4 7 8 2 9) )
3
```

Приклад 2: Функція рахує кількість чисел за індексами 1..4:

```
CL-USER> (COUNT-IF 'NUMBERP '(4 7 a 2 #h 9) :START 1
:END 5)
2
```

Приклад 3: В прикладі послідовність задається списком, елементи якого теж списки. У кожного внутрішнього списку перевіряється чи являється перший елемент рядком.

```
CL-USER> (COUNT-IF 'STRINGP '((4) ("7" 8 2) ("1"))) :KEY
'CAR)
```

```
2
```

```
(CAR '(4)) --> 4 (STRINGP 4) Лічильник 0
```

```
(CAR '("7" 8 2)) --> "7" (STRINGP "7") Лічильник 1
```

```
(CAR '("1")) --> "1" (STRINGP "1") Лічильник 2
```

Приклад 4: У прикладі застосовується ключ :FROM-END. Він вказує, що елементи розглядаються від кінця послідовності.

```
CL-USER> (COUNT-IF 'EVENP '(4 7 8 2 10) :END 2 :FROM-
END T)
```

```
1
```

Індекси елементів рахуються зліва направо, тому ключ :END вказує, що розглядати треба тільки 2 числа 4 і 7. А за ключем :FROM-END числа розглядаються зворотному порядку 7, 4.

Функція **FIND** знаходить перший знайдений елемент послідовності, який порівнюваний з об'єктом за тестом і повертає цей елемент.

(FIND об'єкт послідовність ключі_та_їх_значення)

Для роботи функція FIND може застосовувати наступні ключі: :TEST, :TEST-NOT, :KEY, :START, :END, FROM-END.

Приклад 1: Задамо послідовність списком чисел. Функція знаходить і повертає перший знайдений елемент більший 5.

```
CL-USER> (FIND 5 '(4 2 10 14) :TEST '<)
```

```
10
```

Виконуються порівняння:

```
5<4 NIL
```

```
5<2 NIL
```

```
5<10 T
```

Приклад2: Застосуємо у прикладі крім тесту '< ключі :START і :END.

```
CL-USER> (FIND 5 '(4 2 10 1 15 19) :TEST '< :START 3 :END 4)
15
```

Приклад3: Елементами рядка є знаки. У прикладі знаки рядка порівнюються з об'єктом #\с за тестом протилежним тесту CHAR>.

```
CL-USER> (FIND #\с "akas" :TEST-NOT 'CHAR>)
#\k
#\с <= #\a NIL
#\с <= #\k T
```

Приклад 4: У прикладі заданий список символів. За ключем :START розглядаються елементи послідовності починаючи з індексу 1. За ключем :END - розглядаються елементи послідовності до індексу 5. Ключ :FROM-END вимагає перегляду елементів списку від його кінця.

```
CL-USER> (FIND 'd '(D c D u d) :start 1 :END 5 :FROM-END
T)
```

D

Повертається підкреслений елемент d за індексом 4.

Приклад 5: У прикладі послідовність задається списком, елементи якого теж списки.

```
CL-USER> (SETQ L'( (N 98) (A 96) (A 90) ) )
((N 98) (A 96) (A 90) )
CL-USER> (FIND 'A L :KEY 'CAR)
(A 96)
```

Функція CAR застосовується до кожного елементу списку, після чого виконується тестування результату за EQL:

```
(CAR '(N 98)) --> N          (EQL A N)--> NIL
(CAR '(A 96)) --> A          (EQL A A)--> T
```

Повертається весь елемент списку (A 96)

Приклад 6: Функція FIND знаходить перший елемент, порівняний за тестом '<' з середнім арифметичним.

У прикладі для обчислення середнього арифметичного елементів застосовується функція користувача AVERAGE.

```
CL-USER> (SETQ L '(4 7 8))
```

```
L
```

```
CL-USER> (DEFUN AVERAGE (L) (/ (REDUCE '+ L) (LENGTH  
L)))
```

```
AVERAGE
```

```
CL-USER> (FIND (AVERAGE L) L :TEST '<)
```

```
7
```

Функція **FIND-IF** знаходить перший елемент послідовності, що задовольняє тесту і повертає цей елемент. Тест задається першим аргументом функції без ключового слова. Функція має ключі :START, :END, :KEY, :FROM-END.

(FIND-IF об'єкт послідовності ключі_та_їх_значення)

Приклад 1: Функція знаходить перший парний елемент списку.

```
CL-USER> (FIND-IF 'EVENP '(7 8 2 9) )
```

```
8
```

Приклад 2: Функція знаходить у списку перший знайдений символ за індексами 1..4:

```
CL-USER> (FIND-IF 'SYMBOLP '(4 7 a n #\h 9) :START 1 :END  
5)
```

```
A
```

Приклад 3: В прикладі послідовність задається рядком, в якому є цифри. Треба знайти у рядку першу цифру.

Щоб виявити цифру тестом 'NUMBERP, треба перетворювати кожен знак рядка у число.

Функція SR перетворює знак у число. Якщо число попадає в діапазон 0..9, то число повертається в функцію FIND-IF, інакше повертається NIL.

```

CL-USER> (DEFUN SR (N)
  (IF (AND ( >= ( - (CHAR-CODE N) 48)0) (<= ( - (CHAR-CODE
N) 48) 9))
  (RETURN-FROM SR (- (CHAR-CODE N) 48))
  (RETURN-FROM SR NIL)))
SR
CL-USER> (FIND-IF 'NUMBERP "a3 4 h" :KEY 'SR)
3
(SR #\a) --> NIL
(SR #\3)--> 3

```

Приклад 4: Знайти перше непарне число списку від його кінця.

```

CL-USER> (FIND-IF 'ODDP '(4 7 8 2 5 10) :END 2 :FROM-END
T)
7

```

Індекси елементів рахуються зліва направо, тому ключ :END вказує, що розглядати треба тільки 3 числа 4, 7, 8. А за ключем :FROM-END числа розглядаються зворотному порядку 8, 7, 4.

Функція **POSITION** знаходить і повертає індекс першого знайденого елементу послідовності, який порівнюваний з об'єктом за тестом.

(POSITION об'єкт послідовність ключі_та_їх_значення)

Для роботи функція POSITION може застосовувати наступні ключі: :TEST, :TEST-NOT, :KEY, :START, :END, FROM-END.

Приклад 1: Задамо послідовність списком чисел. Функція знаходить і повертає індекс першого знайденого елементу більшого 5.

```

CL-USER> (POSITION 5 '(4 2 10 14) :TEST '<)
2

```

Приклад2: Застосуємо у прикладі крім тесту ключі :START і :END.

```
CL-USER> (POSITION 5 '(4 2 10 14 15) :TEST '< :START 3
:END 4)
2
```

Приклад3: Елементами рядка є знаки. У прикладі знаки рядка порівнюються з об'єктом #\с за тестом протилежним тесту CHAR>.

```
CL-USER> (POSITION #\с "скас" :TEST-NOT 'CHAR>)
0
```

Приклад 4: Елементами рядка є знаки. Порівняння не виконується, бо знаки рядка #\с і знак #\С різні.

```
CL-USER> (POSITION #\С "скас" :TEST-NOT 'CHAR>)
NIL
```

POSITION-IF знаходить і повертає індекс першого елемента послідовності, що задовольняє тесту. Функція має ключі :START, :END, :KEY, :FROM-END.

(POSITION-IF об'єкт послідовність ключі_та_їх_значення)

Приклад 1: Функція знаходить і повертає індекс першого парного елемента списку.

```
CL-USER> (POSITION-IF 'EVENP '(4 7 8 2 9) )
0
```

Приклад 2: У задано атоми і структури. Знайти індекс першого атому.

```
CL-USER> (POSITION-IF 'ATOM "(2 3) 4 "df")
0
```

Перший атом у рядку – знак #\(. Кожний пропуск рядка теж знак.

1.1.3 Функції конструктори

(REMOVE об'єкт послідовність ключі_та_їх_значення)

Приклад 1: У прикладі вказано, що тестування ведеться за тестом протилежним 'EQL. З списку вилучаються всі числа, які не дорівнюють 3.

```
CL-USER> (REMOVE 3 '(3 12 3 4) :TEST-NOT 'EQL)
(3 3)
```

Приклад 2: Об'єкт 3 порівнюється за тестом з CAR - елементом кожного підписку. Якщо тест задовольняється, то такий підписок вилучається.

```
CL-USER> (REMOVE 3 '((5 7) (3 4) (12 3) (3 7)) :KEY 'CAR)
((5 7) (12 3))
```

Приклад 3: В рядку немає знаку #2.

```
CL-USER> (REMOVE #2 "1 4 3 7 5")
(1 4 3 7 5)
```

Приклад 4: З рядку вилучаються 2 знака 4. Після 1 і 3 залишаються 2 пропуски.

```
CL-USER> (REMOVE #4 "1 4 3 4 5 4" :COUNT 2)
"1 3 5 4"
```

Приклад 5: Вилучити у рядку всі пропуски. Після #\ -пропуск.

```
CL-USER> (REMOVE #\ "1 4 3 4 5 4")
"143454"
```

Приклад 6: У рядку вилучаються 2 цифри 4 від кінця рядку

```
CL-USER> (REMOVE #4 "143454" COUNT 2 :FROM-END T)
"1435"
```

Функція **REMOVE-IF** вилучає всі елементи послідовності за тестом. Тест обирається за типом елементів списку. Функція застосовує ключі: :COUNT, :KEY, :START, :END, :FROM-END. Тест обирається за типом елементів послідовності. Ключ :COUNT вказує скільки елементів послідовності можна вилучити.

(REMOVE-IF тест послідовність ключі_та_їх_значення)

Приклад 1: Вилучити у списку всі знаки.

```
CL-USER> (REMOVE-IF 'CHARACTERP '(3 #1 3 4 #1))
(3 3 4)
```

Приклад2: Вилучити із списку 1 знак.

```
CL-USER> (REMOVE-IF 'CHARACTERP '(3 #\1 3 4 #\1)
:COUNT 1)
(3 3 4 #\1)
```

Приклад 3: Вилучити із рядку один знак від кінця рядку.

```
CL-USER> (REMOVE-IF 'CHARACTERP "3 #\1 3 4 #\1 5"
:COUNT 1 :FROM-END T)
(3 #\1 3 4 5)
```

1.1.4 Функції модифікатори послідовностей

Функція **SORT** сортує за вказаним тестом елементи послідовності. Тип тесту повинен відповідати типу елементів послідовності. Функція повертає відсортовану послідовність.

Для тесту замовчання немає. Можна застосовувати ключ : KEY з ім'ям функції, яку попередньо застосовують до елементів послідовності.

(SORT послідовність тест)

Приклад 1: Відсортуємо елементи числового списку.

```
CL-USER> (SETQ S '(1 4 2 5))
(1 4 2 5)
CL-USER> (SORT S '<)
(1 2 4 5)
CL-USER> S
(1 2 4 5)
```

Приклад 2: Нехай є список, елементи якого теж списки. Відсортуємо внутрішні списки за першим елементом.

```
CL-USER> (SETQ S '((1 4) (12 2) (5)))
((1 4) (12 2) (5))
CL-USER> (SORT S '> :KEY 'CAR)
((12 2) (5) (1 4))
```

Приклад 3: Відсортуємо знаки рядку.

```
CL-USER> (SETQ S "flag")
```

```
"flag"
```

```
CL-USER> (SORT S 'Char>)
```

```
"lfga"
```

Функція **NREVERSE** одержує результат аналогічно функції конструктору **REVERSE** – повертає елементи послідовності у зворотному порядку. Але функція **NREVERSE** не будує новий список, а змінює посилки.

(NREVERSE послідовність)

Приклад 1: Змінити порядок елементів у списку на зворотний.

```
CL-USER> (SETQ L '(15 3 4))
```

```
(15 3 4)
```

```
CL-USER> (NREVERSE L)
```

```
(4 3 15)
```

```
CL-USER> L
```

```
(4 3 15)
```

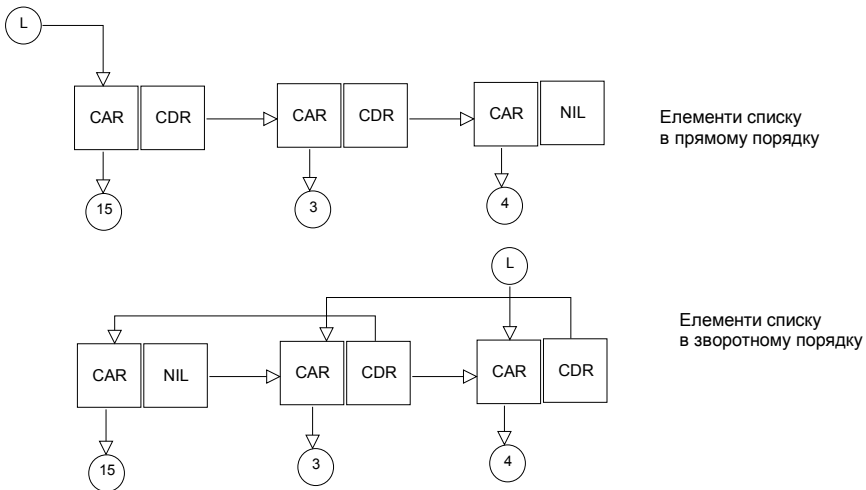


Рисунок 1.2 – Робота функції NREVERSE

Приклад 2: Змінити порядок знаків рядку на зворотний.

```
CL-USER>(SETQ S "a Red")
```

```
"a Red"
```

```
CL-USER>(NREVERSE S)
```

```
"deR a"
```

```
CL-USER>S
```

```
"deR a"
```

Функція **NSUBSTITUTE** замінює всі елементи послідовності, що порівнюються за тестом з вказаним, на новий елемент. Заміна для списків відбувається тільки у зовнішньому списку. Функція повертає одержаний список.

Функція має ключі: :TEST, :TEST-NOT, :KEY, :START, :END, FROM-END, COUNT. Замовчення тест EQL.

**(NSUBSTITUTE новий_елемент старий_елемент
послідовність_ключі_та_їх_значення)**

Приклад 1: Функція замінює всі числа 5 на 9 за тестом EQL.

```
CL-USER> (SETQ L '(15 5 3 5 4))
```

```
(15 5 3 5 4)
```

```
CL-USER> (NSUBSTITUTE 9 5 L)
```

```
(15 9 3 9 4)
```

```
CL-USER> L
```

```
(15 9 3 9 4)
```

Приклад 2: Заміна елементів відбувається тільки у зовнішньому списку.

```
CL-USER> (SETQ L '(15 5 (3 5) 4))
```

```
(15 5 (3 5) 4)
```

```
CL-USER> (NSUBSTITUTE 9 5 L)
```

```
(15 9 (3 5) 4)
```

Приклад 3: Заміна елементів за тестом '>.

```
CL-USER> (SETQ L '(15 5 3 5 4))
```

```
(15 5 3 5 4)
```

```
CL-USER> (NSUBSTITUTE 9 5 L :TEST '>)
```

```
(15 5 9 5 9)
```

Приклад 4: Якщо в прикладі застосовується ключ :key, то функція повинна застосовуватися до кожного елементу списку. Причому замінюється весь елемент списку.

```
CL-USER> (NSUBSTITUTE 9 5 '((5 3) (4 5)) :KEY 'CAR)
```

```
(9 (4 5))
```

Приклад 3: Заміна знаків рядка.

```
CL-USER> (SETQ R "SKKKD")
```

```
"SKKKD"
```

```
CL-USER> (NSUBSTITUTE #\A #\K R)
```

```
"SAAAD"
```

```
CL-USER> R
```

```
"SAAAD"
```

Функція **NSUBST** замінює всі елементи послідовності, що порівнюються за тестом з вказаним, на новий елемент. Заміна відбувається на всіх рівнях (у зовнішньому списку і внутрішніх списках. Ключі та тести як у функції **NSUBSTITUTE**. Синтаксис той же самий.

Приклад 1.

```
CL-USER> (SETQ L '((4 5 6) 4 (5 4)) )
```

```
CL-USER> (NSUBST 9 4 L)
```

```
((9 5 6) 9 (5 9))
```

```
CL-USER>L
```

```
((9 5 6) 9 (5 9))
```

Функції **DELETE** та **DELETE-IF** одержують результати аналогічно функціям **REMOVE** та **REMOVE-IF** - вони вилучають вказані елементи послідовності. Але ці функції модифікують посилки на конси списків. Функції **DELETE** та **DELETE-IF** застосовують аналогічні цим функціям ключі та їх значення.

(DELETE об'єкт послідовність ключі_та_їх_значення)

(DELETE-IF тест послідовність ключі_та_їх_значення)

Приклад 1. У прикладі із списку вилучаються всі 4.

CL-USER> (SETQ N '(2 4 7 4 6))

(2 4 7 4 6)

CL-USER> (delete 4 N)

(2 7 6)

На рисунку 1.3 подано модифікацію посилки на конси списку.

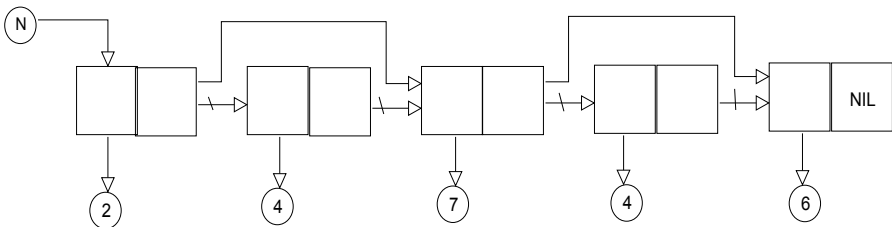


Рисунок 1.3 Модифікація посилки при вилучанні елементів списку

Завдяки модифікації посилки функції DELETE та DELETE-IF працюють швидше ніж конструктори. При роботі конструкторів час витрачається як при побудові нових послідовностей, так і при роботі програми «Смітник», яка звільнює пам'ять від непотрібних даних.

Приклад 2: Вилучити з рядку всі букви а.

CL-USER> (delete #\a "cada")

"cd"

ТЕМА2: Функції та макроси роботи із списками

2.1 Функції відбору

Загальна форма функції LAST подається:

(LAST список кількість елементів)

Функція повертає список із вказаної кількості останніх елементів.

Приклад1:

```
CL-USER> (SETQ L '(1 2 3 4))
```

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

```
CL-USER> (LAST L 2)
```

```
(3 4)
```

```
CL-USER> L
```

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

Якщо кількість елементів не вказано, то по замовчанню повертається останній конс списку. Останній конс списку тотожний списку із останнім елементом списку.

```
(4 . NIL) ≡ (4)
```

У конспекті лекцій частина 1 ми розглядали роботу функції LAST за замовчанням. Функція повертала останній конс списку.

Приклад2:

```
CL-USER> (LAST L)
```

```
(4)
```

Приклад 3: Якщо кількість елементів у виклику функції LAST більше або дорівнює кількості елементів списку, то повертається сам список.

```
CL-USER> (LAST '(2 4 6 8) 5)
```

```
(2 4 6 8)
```

Функція **LDIFF** – виділяє хвостом цього ж списку його початок і повертає початок списку до хвоста, що розташований за тією ж

адресою. На хвіст списку можна дивитися як на маску, якою виконується виділення. Важливо, що маску треба задавати формою саме цього ж списку.

(LDIFF список хвіст_списку)

Приклад 1: Функція NTHCDR повертає список (78 12). Сам список '(34 56 78 12)' не змінюється.

```
CL-USER> (SETQ N '(34 56 78 12))
```

```
N
```

```
CL-USER> (LDIF N (NTHCDR 2 N))
```

```
(34 56)
```

Адреса списку (34 56 78 12) і списку (34 56) та ж сама. Якщо маску задати (78 12)), то виділення списку не відбувається і повертається весь список:

```
CL-USER> (LDIF N '(78 12))
```

```
(34 56 78 12)
```

2.2 Функції конструктори

Функція **MAKE-LIST** створює список постійної довжини і заповнює його вказаним елементом.

**(MAKE-LIST довжина_списку :initial element
значення_елементу)**

Приклад 1: Треба заповнити список 0.

```
CL-USER> (SETQ M (MAKE-LIST 6 :INITIAL-ELEMENT 0))
```

```
(0 0 0 0 0 0)
```

Приклад 2: Записати за 2 індексом 5.

```
CL-USER> (SETF (NTH 2 M) 5)
```

```
(0 0 5 0 0 0)
```

Приклад 3: Записати за 2 індексом 5.

```
CL-USER> (SETF (NTH 6 M) 5)
```

```
(0 0 5 0 0 0)
```

ERROR: Індекс дуже великий.

Функція **APPEND** зливає будь-яку кількість списків у один список. Перед зливанням списків для кожного списку будується копія. Копії зливаються у порядку запису списків у виклику функції.

Завдяки копіям списків результат злиття списків не залежить від вмісту списків, якщо вони зміняться.

(APPEND список1 список2 ...списокN)

Приклад 1:

```
CL-USER> (SETQ L1 '(1 2) L2 '(3 4) M '(5 6))
(5 6)
```

```
CL-USER> (SETQ N (APPEND L1 M L2 M))
(1 2 5 6 3 4 5 6)
```

Приклад 2: Функцію можна застосовувати для створення циклічних списків.

```
CL-USER> (SETQ L1 '(1 3))
(1 3)
```

```
CL-USER> (SETQ M (APPEND L1 L1 L1))
(1 3 1 3 1 3)
```

Функція **MEMBER** повертає хвіст списку перший елемент якого порівнюється з об'єктом за тестом. Функція застосовує ключі: TEST, TEST-NOT, KEY. Індекс рахується з 0. Тест обирається за типом елементів списку. Замовчення для тесту 'EQL.

(MEMBER об'єкт список ключі_та_їх_значення)

Приклад 1: У прикладі вказано, що тестування ведеться за тестом протилежним 'EQL, тому хвіст починається з елемента 12.

```
CL-USER> (MEMBER 3 '(3 12 3 4) :TEST-NOT 'EQL)
(12 3 4)
```

Приклад 2: Об'єкт 3 порівнюється за тестом з Car-елементом списку.

```
CL-USER> (MEMBER 3 '((5 7) (3 4) (12 3) (2 7)) :KEY 'CAR)
```

```
((3 4) (12 3) ( 2 7))
```

Приклад 3: В списку немає елементу 2.

```
(MEMBER 2 '(1 3 5))
```

```
NIL
```

Приклад 4: У прикладі застосовується тест '<.

```
CL-USER> (MEMBER 3 '((5 7) (3 4) (12 3) (2 7)) :KEY 'CAR
:TEST '<)
```

```
((5 7)(3 4) (12 3) ( 2 7))
```

Функція **MEMBER-IF** повертає хвіст списку перший елемент якого задовольняє тесту. Тест обирається за типом елементів списку.

(MEMBER-IF об'єкт список ключі_та_їх_значення

```
CL-USER> (MEMBER-IF 'CHARACTERP '(3 #\1 3 4) )
```

```
(#\1 3 4)
```

КОПІЮВАННЯ СПИСКІВ

Іноді треба зробити копію списку, якщо список буде змінюватися, а далі треба його застосовувати ще. Копію списку не можна зробити, привласнюючи посилку на нього іншому символу.

Приклад 1: Посилка на список символу М привласнюється символу N.

```
CL-USER> (SETQ M '(10 32 16) N M)
```

```
(10 32 16)
```

```
CL-USER>N
```

```
(10 32 16)
```

При заміні першого елемента списку функцією модифікатором RPLACA посилка символу N буде показувати на змінений список.

```
CL-USER> (RPLACA M 7)
```

```
(7 32 16)
```

```
CL-USER> N
```

```
(7 32 16)
```

При заміні елементу списку обидві посилки вказують на змінений список. Дивися рисунок 2.1.

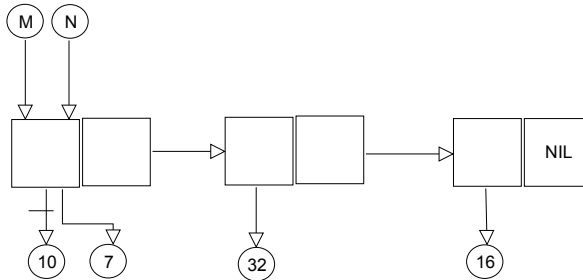


Рисунок 2.1 Копіювання посилки

Але при застосуванні модифікатору списку POP, який вилучає перший елемент списку, посилка на перший елемент списку M буде показувати на другий елемент списку, а посилка символу N вказує як і раніше на перший елемент списку. Дивися рисунок 2.2.

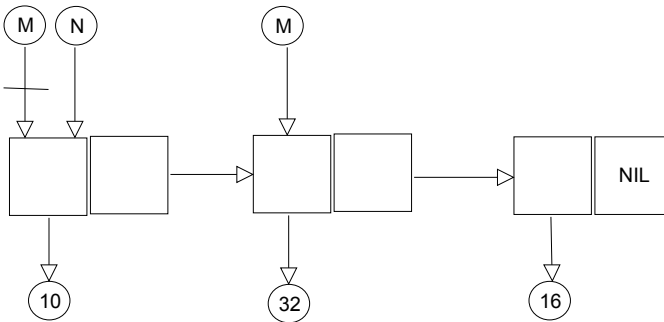


Рисунок 2.2 Символи M і N мають різні посилки

CL-USER> (POP M)

(32 16)

CL-USER> N

(10 32 16)

Щоб уникнути помилок краще робити копію списків функцією **COPY-LIST**. Функція створює копію списку і повертає її.

(COPY-LIST список)

Якщо привласнити посилку на копію символу, помилки не буде.

Приклад1.

```
CL-USER> (SETQ M '(10 32 16) N (COPY_LIST M)
(10 32 16)
```

```
CL-USER> (RPLACA M 7)
(7 32 16)
```

```
CL-USER> N
(10 32 16)
```

Приклад 2: Функцію COPY_LIST можна застосувати для перевірки чи відсортований список.

```
CL-USER> (SETQ L '(2 4 6 8))
(2 4 6 8)
```

```
CL-USER> (EQUAL L (SORT (COPY-LIST L) '>))
NIL
```

2.3 Функції модифікатори

Функції не будують новий список, а змінюють існуючий. Розглянемо функцію модифікатор, результат якої такий же як у функції APPEND, але виконує вона свої дії іншим чином.

Функція **NCONC** зливає кілька списків у один. Функція повертає злитий список.

(NCONC список 1 список 2 ... список N)

Приклад 1: Нехай задано три списки.

```
CL-USER> (SETQ A '(1 2) B '(25 14) C '(35 2))
(35 2)
```

Нагадаємо, що список складається з точкових пар (консів), де ліву частину консу прийнято називати CAR вічком, а праву частину CDR вічком за базовими функціями, які з ними працюють.

На рисунку 2.3. показані списки, що зливаються.

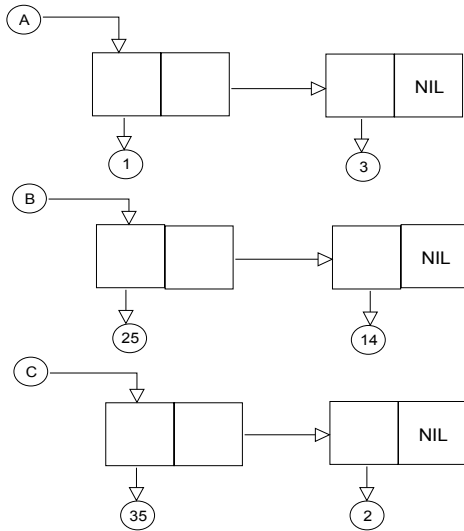


Рисунок – 2.3 Списки - аргументи функції NCONC

Зіллємо списки функцією NCONC.

CL-USER> (NCONC A B C)

(1 2 25 14 35 2)

Дивися рисунок 2.4.

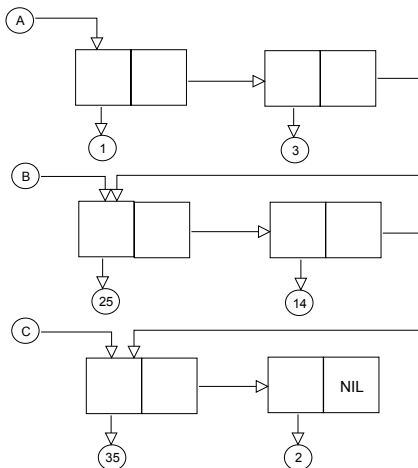


Рисунок – 2.4 Результат роботи функції NCONC

Після злиття списків модифікуються CDR вічки останніх консів всіх списків крім останнього списку. Переглянути сказане можна за наступними діями.

```
CL-USER> A
(1 2 25 14 35 2)
```

```
CL-USER> B
(25 14 35 2)
```

```
CL-USER> C
(35 2)
```

Якщо привласнити символу В або С покажчики на інші списки, то доступу через ці символи до частини злитого списку не буде.

```
CL-USER> (SETQ C '( 8 9 1))
(8 9 1)
```

```
CL-USER> C
(8 9 1)
```

```
CL-USER> A
(1 2 25 14 35 2)
```

Функція **RPLACA** заміщує у списку посилку з CAR елементу першого консу посилкою на вказаний елемент.

(RPLACA список елемент)

Приклад 1: Створимо список А. Замінімо посилку на число 5 посилкою на число 1

```
CL-USER> (SETQ A '(5 3 7))
(5 3 7)
```

```
CL-USER> (RPLACA A 1)
(1 3 7)
```

```
CL-USER> A
(1 3 7)
```

Результат роботи подано на рисунку 2.5.

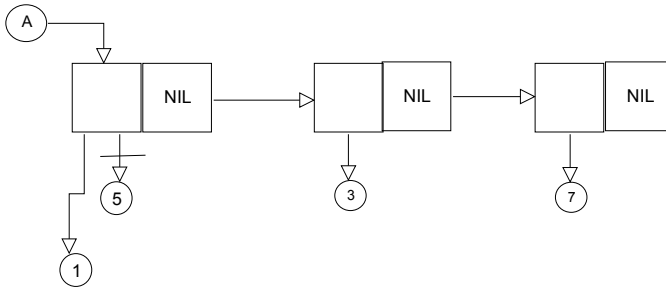


Рисунок 2.5 Робота функції RPLACA до прикладу 1

Приклад 2: У списку L замінимо посилку на число 4 посилкою на список N.

```
CL-USER> (SETQ L '(4 5 6) N '(7 8))
```

```
(7 8)
```

```
CL-USER> (RPLACA L N)
```

```
((7 8) 5 6)
```

```
CL-USER> N
```

```
(7 8)
```

Функція **RPLACD** заміщує у списку посилку з CDR елементу першого консу посилкою на вказаний елемент.

(RPLACD список елемент)

Приклад 1: Замінимо у списку A посилку на другий елемент цього списку посилкою на список C.

```
CL-USER> (SETQ A '(1 2 3) C '(6 7 8))
```

```
(6 7 8)
```

```
CL-USER> (RPLACD A C)
```

```
(1 6 7 8)
```

```
CL-USER> A
```

```
(1 6 7 8)
```

```
CL-USER> C
```

```
(7 8)
```

На рисунку 2.6 показано, як замінюється посилка.

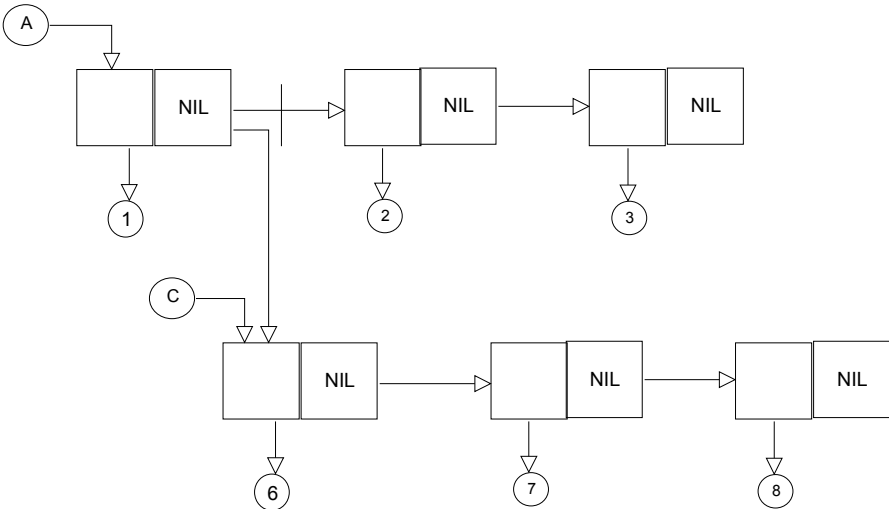
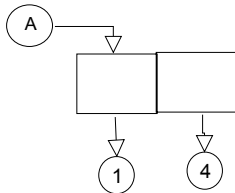


Рисунок 2.6 Робота функції RPLACD до прикладу 1

Приклад 2: У списку A з прикладу 1 замінимо CDR посилку з першого консу списку A на перший конс списку C посилкою на число 4. Дивися попередній рисунок 2.6. В результаті одержимо конс (рисунок 2.7).

CL-USER> (RPLACD A 4)

(1 . 4)



2.7-Результат роботи функції RPLACD до прикладу 2

Приклад 3: Задамо при виклику функції RPLACD форму (NTHCDR 1 A) замість списку A.

CL-USER> (SETQ A '(1 6 7 8))

```
(1 6 7 8)
```

```
CL-USER> (RPLACD (NTHCDR 1 A) 3)
```

```
(6 . 3)
```

Форма повертає число 6, тому функція RPLACD повертає конс (6 . 3). Сам список А змінюється – другим елементом стає конс.

```
CL-USER> A
```

```
(1 6 . 3)
```

Символ А містить посилку на список з двох елементів. Перший елемент списку – число 1, а другий елемент конс (6 . 3).

Приклад 4: Модифікуємо список L, додавши його останнім елементом число 9.

```
CL-USER> (SETQ L '(1 2 3 4))
```

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

```
CL-USER> (RPLACD (NTHCDR 3 L) '(9))
```

```
(4 9)
```

```
CL-USER> L
```

```
(1 2 3 4 9)
```

Функція **NBUTLAST** вилучає від кінця списку вказану кількість елементів і повертає залишок списку. Замовчання 1 елемент. Якщо кількість елементів при виклику функції більше або дорівнює довжині списку, то функція повертає NIL, а сам список не змінюється.

(NBUTLAST список кількість_елементів)

Приклад 1: Кількість елементів не вказана.

```
CL-USER> (SETQ A '(1 2 3 4 5))
```

```
(1 2 3 4 5)
```

```
CL-USER> (NBUTLAST A)
```

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

Приклад 2: Кількість елементів, що вилучаються 3.

```
CL-USER> (NBUTLAST A 3)
```

(1)

Приклад 3: Кількість елементів більше або дорівнює довжини списку.

```
CL-USER> (NBUTLAST A 1)
```

```
NIL
```

```
CL-USER>A
```

(1)

Нагадаємо, що константа NIL означає неправду або порожній список. У прикладі NIL означає неправду.

Функція NBUTLAST не може вилучати всі елементи списку.

ТЕМА3:Вектори

3.1 Загальні відомості про вектори

В математиці вектор - має кількісну величину та напрямлення. Для створення програм, що застосовують методи векторного аналізу, у програмуванні введено тип даного вектор.

У Common Lisp вектор є підтипом послідовності. В той же час вектор це підтип масиву. Тому вектор – множина об'єктів, до яких можна звертатися за індексами. Вектори загального призначення мають об'єкти будь-якого типу.

Наприклад:

```
CL-USER> (SETQ V #(R 45 T 3.14))
```

Існують спеціалізовані вектора, які мають всі об'єкти тільки одного типу. До спеціалізованих векторів відноситься рядки, бітові вектори. Об'єктами рядку можуть бути тільки є знаки. Об'єктами бітового вектору можуть бути тільки 1 і 0.

Спеціалізовані вектора мають свої функції, що орієнтовані на обробку даних певного типу.

Вектори можуть бути *фіксованої довжини*. Створити вектор фіксованої довжини (константу) можна за функціями:

Приклад 1: Створення констант - векторів.

```
CL-USER> (VECTOR 2 4 5)
```

```
#(2 4 5)
```

або

```
CL-USER> (MAKE-ARRAY 3 :INITIAL-ELEMENT 0)
```

```
#(0 0 0)
```

Змінювати вміст таких векторів небезпечно. Краще застосовувати для створення вектора фіксованої довжини функцію MAKE-ARRAY з ключем :FILL-POINTER. Це дозволить змінювати вміст вектору, але не довжину.

Приклад 2: Створення порожнього вектору фіксованої довжини для попереднього заповнення.

```
CL-USER> (SETQ L (MAKE-ARRAY 3 :FILL-POINTER 0))
```

```
#()
```

Приклад 3: Заповнення вектору за функцією VECTOR-PUSH. Повертається індекс на елемент

```
CL-USER> (VECTOR-PUSH 1 L)
```

```
0
```

```
CL-USER>L
```

```
#(1)
```

Приклад 4: Побудувати вектор довжини, яку можна змінити.

```
CL-USER> (SETQ L (MAKE-ARRAY 3 :FILL-POINTER 0  
:ADJUSTABLE T))
```

```
T))
```

```
#()
```

Заповнення вектору виконується за функцією VECTOR-PUSH-EXTEND, яка працює як функція VECTOR-PUSH.

```
CL-USER> (VECTOR-PUSH-EXTEND 15 L)
```

```
0
```

```
CL-USER>L
```

```
#(15)
```

```
CL-USER> (VECTOR-PUSH-EXTEND 127 L)
```

```
0
```

```
CL-USER>L
```

```
#(15 127)
```

3.2 Спеціалізовані вектори - рядки. Функції роботи з рядками

Раніше ми розглядали функції, які можна застосувати до будь-яких послідовностей, в тому числі і до рядків. В цьому параграфі ми розглянемо функції, які можна застосовувати до рядків і знаків.

Функції цього розділу доповнюють та розширюють матеріал, який подано у методичних вказівках до лабораторних роботах.

Рядок – спеціалізований вектор фіксованої довжини. Доповнювати рядок неможна.

При роботі із знаками та символами треба розрізняти об'єкти.

1. Символ 'a ==> 'A
2. Символ '\a ==> 'a
3. Знак #\A код 65
4. Знак #\a код 97

3.2.1 Функції відбору знаків та рядків

Функція **CHAR** дозволяє одержати знак рядка за індексом. Функція повертає цей знак. Кожному знаку по замовчанню відповідає код ASCII.

(CHAR рядок індекс)

Індекс рахується з 0, як у одномірного масиву. Тому замість функції CHAR можна застосовувати функцію AREF.

```
CL-USER> (AREF "asdf" 1)
```

```
#\s
```

```
CL-USER> (CHAR "asdf" 1)
```

```
#\s
```

Якщо індекс більше дозволеного, то повертається NIL.

```
CL-USER> (CHAR "asdf" 4)
```

```
NIL
```

Приклад 1: Функція CHAR не змінює рядок.

```
CL-USER> (SETQ STR "LISP")
```

```
"LISP"
```

```
CL-USER> (CHAR STR 0)
```

```
#\L
```

```
CL-USER>STR
```

```
"LISP"
```

Приклад 2: Перевірити чи починається і закінчується слово на ту ж саму букву.

```
CL-USER> (DEFUN SLOVO ()
```

```
    (SETQ S (READ))
```

```
    (IF (EQL (CHAR S 0) (CHAR S (- (LENGTH S) 1)))
```

```
        (PRINC "YES") (PRINC "NO")))
```

```
SLOVO
```

```
CL-USER> (SLOVO)
```

```
"arka"
```

```
YES
```

```
"YES"
```

Функція **SEARCH** шукає і повертає позицію початку підрядку в рядку, який знайдено першим. Функція може шукати не весь підрядок, а у вказаних межах. В той же час пошук може проводитися не у всьому рядку, а за вказаними межами. За замовчанням тест EQL

(SEARCH підрядок рядок ключі та їх значення)

Функція застосовує ключі :START1, :END1 для підрядку; :START2, :END2 ДЛЯ РЯДКУ; :TEST, TEST-NOT, FROM-END.

Приклад 1: Пошук підрядку починається в рядку з індексу 1.

CL-USER> (SEARCH "AS" "AAAS AS D" :START2 1)

2

Приклад 2: У рядку ведеться пошук знаку S.

CL-USER> (SEARCH "AS" "AAAS AS D" :START1 1)

3

Приклад 3: Перший знайдений символ, який не дорівнює знаку A, це пропуск за індексом 1.

CL-USER> (SEARCH "A" "A S D" :TEST-NOT 'EQL)

1

Приклад 4: Пошук знаку A ведеться від кінця рядку.

CL-USER> (SEARCH "A" "A S D A" :FROM-END T)

6

Приклад 5: Функція дозволяє шукати об'єкт у списку.

CL-USER> (SEARCH '(1) '(2 4 1 3))

2

Приклад 6: У списку шукається число 3. Аналогічно для списків працюють інші ключі.

CL-USER> (SEARCH '(1 3) '(3 4 1 3) :START1 1)

0

Функція **SUBSEQ** вибирає і повертає підрядок рядку за вказаними індексами. Вибір виконується від індексу початку підрядку до індексу його кінця. Сам індекс кінця не розглядається. Якщо індекс кінця підрядку відсутній або більший за індекс рядку, то вибирається підрядок до кінця рядка. Якщо індекс початку підрядку виходить за межі рядку, то повертається порожній рядок.

(SUBSEQ рядок позиція_початку позиція_кінця)

Приклад 1: Підрядок не включає знак за індексом ключа.

CL-USER> (SUBSEQ "extended" 2 5)

"ten"

Приклад 2: Відсутній індекс кінця підрядку.

```
CL-USER> (SUBSEQ "extended" 2)
"tended"
```

Приклад 3: Індекс початку підрядку більше останнього індексу рядка.

```
CL-USER> (SUBSEQ "extended" 8)
""
```

3.2.2 Функції конструктори рядків

Функція **CONCATENATE** може об'єднати будь-яку кількість рядків. Функція робить копії рядків крім першого, об'єднуючи їх в один рядок. Хоч рядки не можна розширити, бо вони мають фіксовану довжину, функція **CONCATENATE** розширює свій перший рядок. Тому рядки не можна модифікувати.

(CONCATENATE тип рядок1 рядок2 ... рядокN)

Приклад1: При конкатинації рядів функція робить їх копії.

```
(SETQ S1 "AS" S2 "GH")
"GH"
```

```
CL-USER> (SETQ S (CONCATENATE 'STRING S1 S2))
"ASGH"
```

```
CL-USER> (SETQ S1 "23" S2 "56")
"56"
```

```
CL-USER> S
"ASGH"
```

Приклад 2: У прикладі функція об'єднує списки.

```
CL-USER> (CONCATENATE 'LIST '(1 2) '(3 4))
(1 2 3 4)
```

Функції **STRING-LEFT-TRIM** та **STRING-RIGHT-TRIM** відсікають вказаний підрядок зліва або права відповідно.

(STRING-LEFT-TRIM підрядок рядок)

(STRING-RIGHT-TRIM підрядок рядок)

Приклад 1:

```
CL-USER> (STRING-LEFT-TRIM " " " COMPUTER ")
"COMPUTER "
```

Приклад 2:

```
CL-USER> (STRING-RIGHT-TRIM " " " COMPUTER ")
" COMPUTER"
```

3.2.3 Функції перетворення рядків та знаків

Функції **CHAR-CODE** і **CODE-CHAR** перетворюють знак в код і код в знак відповідно. Функції застосовують ASCII код.

Приклад 1: Перетворення знаку #\ (в код.

```
CL-USER> (CHAR-CODE #\ ( )
40
```

Приклад 2: Перетворення коду 32 в знак пропуск.

```
CL-USER> (CODE-CHAR 32 )
#\)
```

Функції **STRING-UPCASE** та **STRIG-DOWNCASE** перетворюють знаки рядків у верхній та нижній регістри відповідно. Функції застосовують ключі :START, :END.

(STRING-UPCASE рядок)**(STRIG-DOWNCASE рядок)**

```
CL-USER> (STRING-UPCASE "Apple")
"APPLE"
```

```
CL-USER> (string-downcase "ApPle")
"apple"
```

Функція **STRING-CAPITALIZE** повертає кожне слово рядку з великої букви.

(STRING-CAPITALIZE рядок)

Приклад 1:

```
CL-USER> (STRING-CAPITALIZE "i am teacher")
"I AM TEACHER"
```

3.2.4 Функції порівняння рядків та знаків

Для порівняння знаків застосовують функції:

CHAR=, CHAR<, CHAR>, CHAR<=, CHAR>=, CHAR/=

Приклад 1: Відсортувати знаки рядку.

```
CL-USER> (SORT "PROLOG 2 LISP 1" 'CHAR<=)
" 12GILLOOPPRS"
```

Функція **EQUAL** порівнює рядки з урахуванням регістру. Виконується лексиграфічне порівняння за кодами знаків.

Приклад 1: Знаки L та l рядків записані у різних регістрах.

```
CL-USER> (EQUAL "Lisp" "lisp")
NIL
```

Приклад 2: Знаки рядків записані у однакових регістрах, але рядки різні.

```
CL-USER> (EQUAL "Lisp" "Lisp2")
NIL
```

Приклад 3: Знаки рядків записані у однакових регістрах, рядки однакові.

```
CL-USER> (EQUAL "Lisp" "Lisp")
T
```

Функція **STRING-EQUAL** порівнює рядки без урахування регістру.

Приклад 1: Знаки L та l рядків записані у різних регістрах, але функція повертає T.

CL-USER> (EQUAL "Lisp" "lisp")

T

Приклад 2: Рядки різні.

CL-USER> (EQUAL "Lisp" "Lisp2")

NIL

Приклад 3: Рядки однакові.

CL-USER> (EQUAL "Lisp" "Lisp")

ТЕМА4: Символи. Застосування символів для обробки текстів

4.1 Чому слова подаються символами?

Мова Лісп орієнтована на розробку інтелектуальних систем. Один із основних компонентів інтелектуальної системи є база знань. Найкраще людина описує зовнішній світ текстом. Тому і текстова база знань найкраще описує будь-яку предметну область.

Знання – єдина цілісна структура, яка подається поняттями. Кожне поняття має ім'я (слово), а вміст поняття подається властивостями у формі тверджень.

Наприклад, слово «яблуко» має наступні властивості:

(клас яблуко фрукт)

(смак яблуко солодке)

(колір яблуко жовте)

(форма яблуко округле)

Твердження в свою чергу складаються із слів. Таким чином зміст кожного слова виражається через інші слова. Тобто, база знань являє собою мережу.

Для подавання слів і зв'язку між словами в мові Лісп застосовуються символи та списки символів, а не рядки.

Покажемо перевагу застосування символів та списків символів для подавання слів та тверджень.

ПЕРЕВАГИ

1. Символ автоматично створюється в середовищі при його вводі, якщо цього символу ще не було введено. Повторна поява символу не створює його, а звертається до цього символу. Кожен символ подається в середовищі тільки один раз. Тому подавання слова символом забезпечує його унікальне подавання в базі знань. В той же час рядки подаються в середовищі стільки раз, скільки з'являються.

2. Подавання слів символами забезпечує швидкість вибору слова. Звертання до символу виконується миттєво, бо у середовищі для кожного пакету є список символів.

3. Для людини природно було б подавати слова рядками. Але порівняння символів виконується швидше ніж порівняння рядків за знаками.

4. Кожен символ має свій список властивостей, які подаються парами «властивість – значення». Властивість, а часто і її значення, також подається символом. Таким чином символи пов'язані у мережу аналогічну зв'язкам у базі знань. Але поповнення середовища знаннями часто відбувається реченнями у формі рядків.

Для перетворення рядка у символ існує функція **INTERN**.

(INTERN РЯДОК)

CL-USER> (INTERN “ЯБЛУКО”)

ЯБЛУКО

:INTERNAL

4.1 Символ. Властивості символів

При створенні середовища мови Лісп перед розробниками стояли також такі задачі: мінімізація оперативної пам'яті під середовище, швидкість роботи в середовищі.

Для мінімізації оперативної пам'яті треба було вести облік змінних і вилучати вже непотрібні змінні та данні; узагальнити представлення змінної та функції в єдиній структурі – символ. Для швидкості роботи забезпечити швидкий пошук даних та змінних через посилення та створення у середовищі пакетів за призначенням.

Тому середовище Ліспу являє собою мережу із символів та даних, які розподілені по пакетах та пов'язані між собою посилками. Якщо в середовищі появляється символ, який не посилається на певне дане, функцію або інший символ, то програма «Смітник» вилучає такий символ із середовища. Аналогічні дії виконуються при появі даного на яке немає посилки із символу.

Символ - об'єкт, що розпізнається функцією предикатом SYMBOLP, і містить поля з посилками. Кожне поле має своє призначення і свої функції, які працюють з ним (дивися конспект лекцій частина1, розділ 5.2.1). Зараз нас буде цікавить посилка на список властивостей символу.

Властивості кожного символу подаються списком, на який направлена посилка із поля символу. Дивися рисунок 4.1.

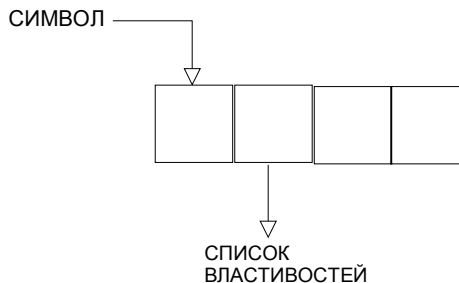


Рисунок 4.1 – Структура символу

Для одержання значень властивостей символу та їх поповнення застосовується функція **GET**.

(GET символ назва_властивості)

Якщо символ не має властивостей, то повертається NIL (порожній список властивостей). Цей факт застосовують, щоб занести у список властивостей символу властивість та її значення.

Приклад 1: Створити властивість «клас фрукт» для символу «яблуко». Для запису властивості застосовується функція SETF – запис за формою. Для роботи із кирилицею працюйте в середовищі CLISP.exe.

```
CL-USER> (SETF (GET 'ЯБЛУКО 'КЛАС) 'ФРУКТ)
ФРУКТ
```

Одержати весь список властивостей символу можна за функцією **SYMBOL-PLIST**.

(SYMBOL-PLIST символ)

Приклад 2:

```
CL-USER> (SYMBOL-PLIST 'ЯБЛУКО)
(КЛАС ФРУКТ)
```

Приклад 3: Додамо до властивостей символу «яблуко» властивість «смак яблуко солодке»

```
CL-USER> (SETF (GET 'ЯБЛУКО 'СМАК) 'СОЛОДКЕ)
СОЛОДКЕ
```

Переглянемо одержаний список властивостей символу «яблуко». Список властивостей заповнюється як стек.

```
CL-USER> (SYMBOL-PLIST 'ЯБЛУКО)
(СМАК СОЛОДКЕ КЛАС ФРУКТ)
```

Приклад 4: Для заміни значення властивості застосовуються ті ж функції: заміна властивості символу не залежить від розташування властивості у списку.

```
CL-USER> (SETF (GET 'ЯБЛУКО 'СМАК) 'КИСЛЕ)
КИСЛЕ
```

```
CL-USER> (SYMBOL-PLIST 'ЯБЛУКО)
(СМАК КИСЛЕ КЛАС ФРУКТ)
```

Для вилучання властивості символу застосовують функцію **REMPROP**. Якщо властивість знайдено, то функція повертає Т, інакше NIL.

(REMPROP символ властивість)

Приклад 5: Вилучимо властивість символу.

```
CL-USER> (REMPROP 'ЯБЛУКО 'КЛАС)
Т
```

```
CL-USER> (SYMBOL-PLIST 'ЯБЛУКО)
(СМАК КИСЛЕ)
```

Приклад 6: Нехай треба скласти речення за шаблоном «діяч; дія; об'єкт дії». Створимо для цього у середовищі словник, у якому будуть вказані біля слів їх семантика у реченні.

```
CL-USER> (SETF (GET 'СТУДЕНТ 'СЕМАНТИКА) 'ДІЯЧ)
ДІЯЧ
```

```
CL-USER> (SYMBOL-PLIST 'СТУДЕНТ)
(СЕМАНТИКА ДІЯЧ)
```

```
CL-USER> (SETF (GET 'СЛУХАЄ 'СЕМАНТИКА) 'ДІЯ)
ДІЯ
```

```
CL-USER> (SYMBOL-PLIST 'СЛУХАЄ)
(СЕМАНТИКА ДІЯ)
```

```
CL-USER> (SETF (GET 'ЛЕКЦІЮ 'СЕМАНТИКА)
'ОБ'ЄКТ_ДІЇ)
ОБ'ЄКТ_ДІЇ
```

```
CL-USER> (SYMBOL-PLIST 'ЛЕКЦІЮ)
(СЕМАНТИКА ОБ'ЄКТ_ДІЇ)
```

Слова словника і шаблон створимо списками. Функцію, що складає речення напишіть самі.

```
CL-USER> (SETQ СЛОВНИК ('ЛЕКЦІЮ, СЛУХАЄ, СТУДЕНТ))
('ЛЕКЦІЮ, СЛУХАЄ, СТУДЕНТ)
```

```
CL-USER> (SETQ ШАБЛОН ('(ДІЯЧ, ДІЯ, ОБ'ЄКТ_ДІЇ))
(ДІЯЧ, ДІЯ, ОБ'ЄКТ_ДІЇ)
```

ТЕМА 5: Числа

5.1 Класифікація чисел

Типи чисел – ієрархічна структура. На рисунку 5.1 зображена структура типів чисел. Функції вищих у структурі типів даних можна використовувати до типів даних розташованих нижче у структурі.

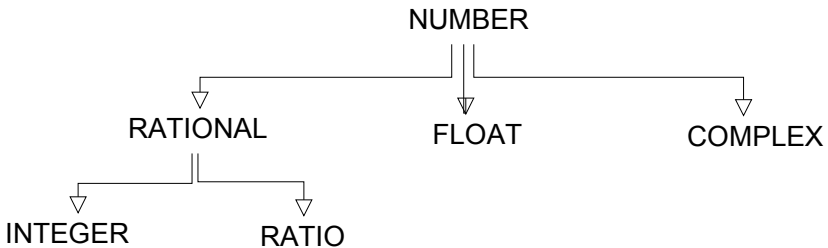


Рисунок 5.1 Типи чисел

Тип числа можна визначити за функцією **TYPEP**. Функція перевіряє, чи відноситься число до вказаного типу.

(TYPEP число тип)

CL-USER> (TYPEP -3 'NUMBER)

T

CL-USER> (TYPEP -3 'RATIONAL)

T

CL-USER> (TYPEP -3 'INTEGER)

T

CL-USER> (TYPEP 1/2 'RATIO)

T

CL-USER> (TYPEP 1.5 'RATIONAL)

NIL

CL-USER> (TYPEP 1.5 'FLOAT)

T

CL-USER> (TYPEP 1.5e+1 'FLOAT)

T

CL-USER> 1.5e+1

15

CL-USER> (TYPEP #C(5 3) 'COMPLEX)

T

Функція **TYPE-OF** визначає тип числа і повертає діапазон.

CL-USER> (TYPE-OF 10)

(INTEGER 0 16777215)

Тип FLOAT має багато під типів, що відрізняються точністю:

S - SHORT-FLOAT скорочене число;

F - SINGLE-FLOAT одинарна точність 7 (замовчення);

D - DOUBLE-FLOAT подвійна точність 14;

L - LONG-FLOAT довге число 19.

Робота з числами може виконуватися у різних системах числення з основами від 2 до 36. По замовчанню застосовується 10-система числення.

Для запису у десятковій системі двійкових чисел застосовують синтаксис #b1010 (10); шістнадцятирічних чисел #xA1 (161); восьмирічних чисел #\o12 (10).

5.2 Арифметичні функції

Функції виконання арифметичних операцій над числами “+”, “-”, “*”, “/” можуть мати будь-яку кількість аргументів.

Операція “+” :

У виразі числа автоматично приводяться до ширшого типу.

CL-USER> (+ 2 4 7.5 1/2)

14.0

CL-USER> (+ 10)

10

Операція “+” без операндів повертає ідентифікатор операції. – 0.
 Додавання ідентифікатору операції не змінює результату.

CL-USER> (+)

0

Операція “” :*

CL-USER> (* -2 1.5 -3)

9.0

CL-USER> (* 1/3)

1/3

Операція “*” без операндів повертає ідентифікатор операції. – 1.

CL-USER> (*)

1

Операція “-” :

Від першого операнду відняти суму інших операндів.

CL-USER> (- 5 3 7 -2)

-3

CL-USER> (- 5)

-5

Операція “/” :

Перший операнд ділиться на добуток інших операндів.

CL-USER> (/ 5 7 8 2)

5/112

CL-USER> (/ 5 10)

 $\frac{1}{2}$

CL-USER> (/ 10 2)

5

Функція знаходження найбільшого спільного дільника **GCD** може застосовуватися до будь-якої кількості цілих чисел. Функція повертає абсолютну величину результату.

```
CL-USER> (GCD 12 -4 -6)
```

2

```
CL-USER> (GCD 12 -7 -6)
```

1

Функція знаходження найменшого спільного кратного **LCM** застосовується до будь-якої кількості цілих чисел.

```
CL-USER> (LCM -12 -4 -6)
```

36

Функція **MAX** знаходить найбільше число з будь-якої кількості чисел. Числа можуть подаватися у різних формах.

```
CL-USER> (MAX 2 1/4 1.5)
```

2

Функція **MIN** знаходить найменше число з будь-якої кількості чисел. Числа можуть подаватися у різних формах.

```
CL-USER> (MIN 2 1/4 1.5)
```

1/4

5.3 Функції транзитивного порівняння

Відношення між елементами називають транзитивним, якщо з того, що перший елемент знаходиться в певному відношенні з другим елементом, а другий елемент знаходиться в цьому ж відношенні з третім, то і перший елемент знаходиться в тому ж відношенні з третім. До транзитивних відношень відноситься відношення “=”.

Функція “=” виконує транзитивне порівняння будь-якої кількості дійсних чисел. Функція повертає TRUE, якщо всі числа порівнюються транзитивно.

Приклад 1.

CL-USER> (= 5 5 5 5 5 6)

NIL

CL-USER> (= 5 5 5 5 5 5)

T

Функція “<” виконує транзитивне порівняння будь-якої кількості дійсних чисел.

Приклад 2.

CL-USER> (< 2 4 6 7 8)

T

CL-USER> (< 2 4 6 1 5 7)

NIL

Функція “/=” виконує транзитивне порівняння будь-якої кількості дійсних чисел.

Приклад 3.

CL-USER> (/= 2 3 4)

T

CL-USER> (/= 4/2 2.0 2)

NIL

5.3 Макроси модифікації чисел за місцем

Нехай посилка символу вказує на число. Макрос **INCF** збільшує це число на вказане число і записує результат за посилкою символу. По замовчанню додається 1.

(INCF символ число)

CL-USER> (SETQ N 6)

6

CL-USER> (INCF N 4)

10

CL-USER> N

10

CL-USER> (INCF N)

11

Макрос **DECF** працює аналогічно макросу INCF, але число зменшується

CL-USER> (SETQ N 16)

16

CL-USER> (DECF N 4)

12

CL-USER> N

12

CL-USER> (DECF N)

11

5.4 Ірраціональні та трансцендентні функції

Алгебраїчні функції складаються з раціональних та ірраціональних функцій, Ірраціональними називають функції, що містять аргумент під знаком радикалу. Трансцендентними називають функції, які не відносять до алгебраїчних.

Всі обчислення по замовчанню мають 7 цифр після точки.

Приклад1: Ірраціональна функція **SQRT** знаходить корінь квадратний.

CL-USER> (SQRT 2)

1.4142135

Приклад 2: Показникова функція e^x .

CL-USER> (EXP 1)

2.7182817

Приклад 3: Знаходиться x^y .

CL-USER> (expt 2 3)

8

Приклад 4: Знаходиться логарифм десятковий $\log_y X$. По замовчанню $Y = e$.

CL-USER> (log 2 4)

1/2

Приклад 5: Знаходиться значення числа π , PI – константа. Кількість знаків після точки 19.

CL-USER> PI

3.1415926535897932385L0

CL-USER> (TYPE-OF PI)

LONG-FLOAT

Приклад 6: Знаходиться $\sin(\pi/2)$.

CL-USER> (SIN (/ PI 2))

1.0L0

Приклад 7: Знаходиться $\cos(\pi)$.

CL-USER> (COS pi)

-1.0L0

Приклад 8: Знаходиться $\tan(\pi)$.

CL-USER> (TAN (/ Pi 4))

1.0L0

Приклад 9: Одержати випадкове дійсне число.

CL-USER> (RANDOM 1.0)

0.1709497

CL-USER> (RANDOM 6)

5

5.5 Функції округлення та усікання

Часто при обчисленні непотрібна дробова частина числа. В цьому разі застосовують функцію **ROUND** для округлення дробового числа до цілого.

Функція ROUND округлює дробове число до цілого і повертає округлене число і відкинуту або добавлену частину числа.

```
CL-USER> (ROUND PI)
3;
0.14159265358979323851L0

CL-USER> (ROUND 4.51)
5;
-0.489999977
```

Часто після виконання дії ділення необхідно зробити зворотну дію - за дільником, часткою і залишком знайти ділене. Для цілих чисел результат ділення - це звичайний дріб або ціле число. Для чисел типу FLOAT результатом буде десятковий дріб.

Приклад 1: У прикладі числа цілі і результат перетворюється у звичайний дріб. Тому зворотна дія виконується точно.

```
CL-USER> (/ 3 4)
3/4

CL-USER> (* 4 3/4)
3
```

Приклад 2: У прикладі результат кінцевий десятковий дріб. Тому зворотна дія дає кінцевий десятковий дріб.

```
CL-USER> (/ 3.0 4.0)
0.75

CL-USER> (* 0.75 4.0)
3.0
```

Приклад 3: У прикладі результат нескінченний десятковий дріб. Тому для зворотної дії застосовують точну верхню грань або точну нижню грань. Точність обчислення залежить від типу FLOAT. У прикладі точність по замовчанню.

```
CL-USER> (/ 1.0 3.0)
0.33333334
```

Назвемо точною верхньою гранню – супремумом (sup) найменше число із всіх чисел, які обмежують певну множину чисел зверху. Назвемо точною нижньою гранню – інфінумом (inf) найбільше число із всіх чисел, які обмежують певну множину чисел знизу.

Функція **CEILING** усікає частку від ділення двох дійсних чисел до цілого за точною верхньою гранню і повертає точну верхню грань і залишок.

Функція **FLOOR** усікає частку від ділення двох дійсних чисел до цілого за точною нижньою гранню і повертає точну нижню грань і залишок.

Приклад 1: Знаходиться $SUP = 1$; залишок = -2. Аргументами функції є цілі числа. Але функція працює з дійсними числами і тому із їх підмножиною цілими числами. Дивися рисунок 5.1.

```
CL-USER> (CEILING 1 3)
```

```
1;
```

```
-2
```

```
CL-USER> (+ (* 3 1) -2)
```

```
1
```

Приклад 2: Знаходиться $INF=0$; залишок = 1

```
CL-USER> (FLOOR 1 3)
```

```
0;
```

```
1
```

```
CL-USER> (+ (* 3 1) -2)
```

На рисунку 5.1 показано супремум та інфінум для частого від ділення 1 на 3.

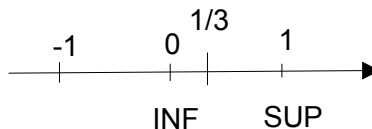


Рисунок 5.1 – Нижня та верхня границі для частки $1/3$

Приклад 3: Знаходиться $SUP = 0$; залишок = -1.

CL-USER> (CEILING -1 3)

0;

-1

CL-USER> (+ (* 3 0) -1)

-1

Приклад 4: Знаходиться $INF = -1$; залишок = 1

CL-USER> (FLOOR -1 3)

-1;

2

CL-USER> (+ (* 3 -1) 2)

-1

На рисунку 5.2 показано супремум та інфімум для частого від ділення 1 на 3.



5.2 - Нижня та верхня границі для частки $-1/3$

Функція **TRUNCATE** усікає частку від ділення двох дійсних чисел до 0. Для додатних чисел працює як FLOOR, а для від'ємних чисел працює як CEILING.

CL-USER> (TRUNCATE 1 3)

0;

1

CL-USER> (TRUNCATE -1 3)

0 ;

-1

Функція **MOD** повертає залишок від ділення двох чисел. Функція працює сумісно з FLOOR.

CL-USER> (MOD 1 3)

1

ТЕМА 6: Змінні

6.1 Поняття змінної

У мові COMMON LISP символ – це об'єкт, що розпізнається і складається із 4-х полів. Кожне поле містить певну посилку. Якщо символ ім'я змінної, то перше поле містить посилку на місце де розташоване дане. Друге поле містить посилку на список властивостей символу. Якщо символ ім'я функції, то третє поле містить посилку на місце коду функції. Четверте поле містить посилку на зовнішнє подавання символу - Print-ім'я.

Нас будуть цікавити символи – імена змінних. На рисунку 5.1 показано, що перше поле символу A містить посилку на місце числа 17.

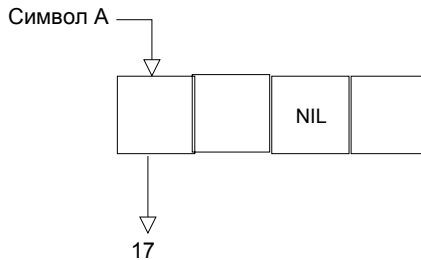


Рисунок 5.1 – Змінна A

Посилка, що розташована у змінній A, прив'язує до змінної число 17. Прив'язка виконується за функцією SET.

```
CL-USER> (SET 'A 17)
```

```
17
```

```
CL-USER>A
```

```
17
```

Посилку змінної A можна замінити посилкою на інший об'єкт а. Значення змінних - це посилки на різні об'єкти. Такий підхід дозволяє застосовувати одну змінну до будь-яких типів даних – реалізовувати динамічну типізацію.

У мові COMMON LISP існують 2 типи змінних лексичні і динамічні. З грецької слово «лексика» означає словниковий запас

мови. Механізм роботи лексичних змінних залежить від імен. По замовчанню всі змінні лексичні. Область видимості лексичної змінної - це область програми, що обмежена одною формою, в тексті якої з'являється і застосовується ця змінна.

Наприклад: Областю видимості лексичної змінної L є все означення функції S, включаючи форму LET, яка є частиною тексту означення функції. Областю видимості змінної A є LET форма.

```
CL-USER> (DEFUN S (L)(LET ((A 3)) (PRINT (+ A (CAR L))) )
(PRINT L))
```

```
S
```

```
CL-USER> (A '(2 4 6))
```

```
5
```

```
(2 4 6)
```

```
(2 4 6)
```

Якщо спробувати надрукувати змінну A замість списку L, то це помилка. Змінна A після форми LET не видна.

Динамічні змінні – це глобальні змінні. Динамічні змінні об'являються. Динамічна змінна видна у будь-якому місці програми. За ANSI CL можливість обійтися без об'яви динамічної змінної залежить від конкретної реалізації мови COMMON LISP. В наступному прикладі ми будемо застосовувати глобальну змінну без об'яви.

Нехай в середовищі динамічній (глобальній) змінній Z. «привласнено» число 5. В тексті означення функції SUM застосовуються лексичні (локальні) змінні X і Y і динамічна змінна Z. Означення функції SUM – це область видимості для X, Y. Глобальну змінну видно в будь-якому місці програми, тому при виконанні функції Sum застосовується вона.

```
CL-USER> (SETQ Z 5)
```

```
5
```

```
CL-USER> (DEFUN SUM (X Y) (+ X Y Z))
```

```
SUM
```

```
CL-USER> (SUM 6 3)
```

6.2 Механізми дії лексичних змінних

До лексичних змінних відносять формальні параметри функції. Областю видимості формальних параметрів є тіло функції. Будемо називати посилку змінної до об'єкту прив'язкою.

Приклад 1: Розглянемо приклад означення функції, яка одержує суму двох чисел. При кожному виклику функції лексична змінна *X* має різні прив'язки до чисел, а динамічна змінна *Y* має ту ж саму прив'язку до числа 5.

```
CL-USER>(SETQ Y 5)
5
CL-USER> (DEFUN SUM (X) (+ X Y))
SUM
CL-USER> (SUM 6)
11
CL-USER> (SUM 12)
17
```

Механізм дії лексичних змінних залежить від імен. Нехай перед застосуванням функції SUM глобальна змінна *Y* має ту ж посилку на число 5, а серед формальних параметрів є змінна теж з ім'ям *Y*.

Правило: при наявності у програмі двох однакових імен внутрішньої змінної і зовнішньої змінної працює внутрішня змінна. Це правило глобальне і працює на всіх рівнях вкладеності області видимості імен.

Приклад 2: У програмі є глобальне ім'я *Y* (зовнішнє) і локальне ім'я *Y* (внутрішнє). При виконанні функції застосовується внутрішнє ім'я.

```
CL-USER> (SETQ Y 5)
5
CL-USER> (DEFUN SUM (X Y) (+ X Y))
```

SUM

CL-USER> (SUM 6 3)

9

Приклад 3: При виконанні рекурсивної функції одна лексична змінна може мати одночасно кілька прив'язок до даних. Пояснимо це на прикладі.

Функція SUM одержує суму елементів списку низхідним методом рекурсії. У функції застосовуються 2 лексичні змінні обов'язковий параметр L і необов'язковий параметр S з початковим значенням 0.

CL-USER> (DEFUN SUM (L &OPTIONAL (S 0))

(COND ((NULL L) S)

(T (+ (CAR L)(SUM (CDR L))))))

SUM

Задамо трасування функції SUM, щоб спостерігати, як поступово для кожної лексичної змінної L і S створюються нові прив'язки.

CL-USER> (TRACE SUM)

:: TRACING FUNCTION SUM.

(SUM)

CL-USER> (SUM '(1 3 5))

Трасування наочно показує, що всі прив'язки до змінної L зберігаються, інакше на етапі рішення не можна було б знаходити CAR елементи при знаходженні суми S.

На етапі редукції змінюються прив'язки до лексичної змінної L:

1. TRACE: (SUM '(1 3 5))

2. TRACE: (SUM '(3 5))

3. TRACE: (SUM '(5))

4. TRACE: (SUM 'NIL)

На етапі одержання рішення змінюються прив'язки до лексичної змінної S:

4. TRACE: SUM ==> 0

3. TRACE: SUM ==> 5

2. TRACE: SUM ==> 8

1. TRACE: SUM ==> 9

9

Форма **LET** дозволяє створювати лексичні змінні, які будуть функціонувати у всіх формах тіла LET.

(LET ((змінна значення) ... (змінна значення) змінна...)

Форми тіла LET)

Перед виконанням форм тіла LET кожна створена лексична змінна ініціалізується своїм значенням або по замовчання значенням NIL. Замість значення може задаватися форма, результатом якої ініціалізується створена змінна. Форма Let повертає результат останньої виконаної форми тіла.

Приклад 4: Розглянемо приклад, коли у програмі є 2 лексичні змінні з одним ім'ям. Внутрішня змінна форми LET - L ховає зовнішню змінну формальний параметр - L на час роботи форми LET.

```
CL-USER> (DEFUN SP (L)
```

```
(LET ((N 2) L)
```

```
(PRINT (SETF L (CONS N L))) )
```

```
L)
```

```
SP
```

```
CL-USER>(SP '(3 ))
```

```
(2)
```

```
(3)
```

До виконання форми LET формальний параметр L має прив'язку до списку (3). У формі LET створюються лексична змінна L з посиланням на NIL та лексична змінна N, що ініціюється 2.

Форма LET повертає список з одним елементом (2), а лексична змінна L форми LET містить посилку на цей список. Після роботи

LET форми лексична змінна L вилучається. Формальний параметр L містить посилку на список (3).

Якщо форму LET буде розміщено в циклі або в рекурсії, то на кожній ітерації будуть створюватися нові лексичні змінні.

Форма **LET*** працює як форма LET за винятком того, що форми початкових значень змінних можуть містити попередні змінні цієї форми.

(LET* ((змінна значення) ... (змінна значення) змінна...)

Форми тіла LET*)

Приклад 5: У прикладі значення змінної Y обчислюється через значення змінної X.

```
CL-USER> (LET* ((X 5) (Y (+ 3 X))) (LIST X Y))
(5 8)
```

Лексичні змінні автоматично створюються у багатьох макросах, які вивчалися раніше: DOTIMES, DOLIST, DO, DO*. Вони існують тільки усередині цих форм.

6.3 Замикання

Якщо у LET формі LET - тіло містить неіменовану функцію, яка застосовує лексичні змінні ініційовані в цій формі LET, то форма поверне посилку на цю функцію. Важливо, що у функції будуть зафіксовані прив'язки форми LET. Тому функцію називають замиканням. Кажуть, що неіменована функція замкнена навколо прив'язки, створеною LET формою.

Приклад 1: Неіменована функція у формі LET доповнює рядок введенням рядком. Створена у формі LET лексична змінна S застосовується у тілі неіменованої функції. Форма LET повертає посилку на неіменовану функцію, у якій фіксується прив'язка змінної S до місця розташування рядка.

```
CL-USER> (LET ((S "")) #'(LAMBDA ()
  (SETF S (CONCATENATE 'STRING S (READ)))))
#<FUNCTION:LAMBDA NIL (SETF S (CONCATENATE
'String S (READ)))>
```

Для виклику функції треба посилку на функцію привласнити динамічній змінній об'явленій через DEFPARAMETER. Задамо цій змінній ім'я *SL*, а початковим значенням змінної вище розглянуту форму LET.

```
CL-USER> (DEFPARAMETER *SL* (LET ((S ""))
  #'(LAMBDA () (SETF S (CONCATENATE 'STRING S
    (READ))))))
*SL*
```

Тепер можна викликати замкнену функцію через змінну *SL*, застосовуючи функціонал FUNCALL.

```
CL-USER> (FUNCALL *SL*)
"AB"
"AB"
```

Щоб зафіксувати місце рядка, треба привласнити рядок глобальній змінній.

```
CL-USER>(SETQ R (FUNCALL *SL*))
"CD"
"ABCD"
CL-USER> R
"ABCD"
```

При кожному виклику робота продовжується з тим самим об'єктом. Виконується розширення того самого рядку.

Замикання дозволяє зафіксувати у кількох функціях прив'язку до конкретного об'єкту. Для цього треба створити список з кількох неіменованих функцій.

Спеціальна форма **FUNCTION**. Результат виклику спеціальної форми завжди має тип - функція. Спеціальною формою FUNCTION можна створювати замикання.

(FUNCTION змінна)

Приклад 1: Спеціальна форма **FUNCTION** інтерпретує свій аргумент як функцію. Вона вказує, що це функція з ім'ям і формальними параметрами, в якому пакеті існує та які виконує дії.

```
CL-USER> (DEFUN S (X Y) (+ X Y))
```

```
S
```

```
CL-USER> (FUNCTION S)
```

```
#<FUNCTION S (X Y) (DECLARE (SYSTEM::IN-DEFUN S))  
(BLOCK S (+ X Y))>
```

Приклад 2: У прикладі аргументом спеціальної форми є системна функція SET.

```
CL-USER> (FUNCTION SET)
```

```
#<SYSTEM-FUNCTION SYSTEM::SET-SYMBOL-VALUE>
```

Приклад 3: Спеціальна форма не може застосовуватися до спеціальної форми, макросу, змінної з даним.

```
CL-USER> (FUNCTION SETQ)
```

```
FUNCTION: undefined function SETQ
```

Приклад 4: Спеціальна форма FUNCTION, застосована до неіменованої функції створює функціональне замикання (блокування).

Створимо замикання неіменованої функції. У означенні функції SUM прив'язка лексичної змінної X фіксується у функції, яку вона повертає. Привласнимо змінній посилку на неіменовану функцію з фіксованою прив'язкою змінної X до 1. Застосуємо функціонал FUNCALL і одержимо результат.

```
CL-USER> (DEFUN SUM (X) (FUNCTION (LAMBDA (Y) (+ X  
Y Z))))
```

```
SUM
```

```
CL-USER> (SETQ SUM1 (SUM 1))
```

```
#<FUNCTION :LAMBDA (Y) (+ X Y Z)>
```

```
CL-USER> (FUNCALL SUM1 5 9)
```

```
15
```

Замикання дозволяє створювати нові функції, у яких фіксується один параметр. Створена функція подібна функції 1+.

6.4 Динамічні змінні

Динамічна змінна діє глобально на всю програму. Для об'яви динамічної змінної є погодження - назва змінної береться у зірки.

Існує 2 способи об'яви динамічної змінної через **DEFVAR** або **DEFPARAMETER**.

(DEFPARAMETER змінна початкове_значення [рядок_докумен.])

Форма DEFPARAMETER завжди привласнює змінній посилку на початкове значення. DEFPARAMETER повертає назву змінної. Форму DEFPARAMETER застосовують у випадку зміни коду програми, якщо значення змінної треба теж змінити.

Приклад 1.

```
CL-USER> (SETQ *C* 3)
```

```
3
```

```
CL-USER> (DEFPARAMETER *C* 10)
```

```
*C*
```

```
CL-USER>*C*
```

```
10
```

Форма **DEFVAR** привласнює змінній посилку на початкове значення, якщо в програмі вона ще невизначена. Форму DEFVAR застосовують у випадку, коли змінні не буде змінювати своє значення, навіть при зміні коду.

(DEFVAR змінна [початкове_значення рядок_документації])

Приклад 2: При об'яві глобальної змінної *COUNT* через DEFVAR початкове значення ігнорується, якщо значення у змінної вже є.

```
CL-USER> (SETQ *COUNT* 6)
```

```
6
```

```
CL-USER> (DEFVAR *COUNT* 0)
```

COUNT

CL-USER>*COUNT*

6

Приклад 3: При об'яві динамічної змінної початкове значення можна не вказувати.

CL-USER> (DEFVAR *SUM*)

SUM

CL-USER> (SETQ *SUM* 15)

15

Стандартне функції часто застосовують динамічні змінні. Наприклад: змінна *READ-BASE* впливає на ввід чисел за вказаною основою, а змінна *PRINT-BASE* впливає вивід чисел за вказаною основою.

Приклад 4: Задамо інтерпретатору вводити числа за двійковою основою. По замовчанню динамічна змінна *PRINT-BASE* має основу 10. Тому функція повертає основу 2. Надалі числа будуть вводиться у двійковій системі.

CL-USER> (SETQ *READ-BASE* 2)

2

CL-USER> (SETQ N (READ))

101

5

При вводі цифр крім 1 і 0 помилка.

CL-USER> 5

Помилка. Число сприймається як символ |5|.

Повернемо ввід у десятковій системі.

(SETQ *READ-BASE* 1010)

10

CL-USER> (SETQ N (READ))

5

5

Динамічну змінну можливо пов'язувати з лексичною змінною в будь-якому місці програми.

Приклад 5: У формі LET динамічна змінна *COUNT* пов'язується з лексичною змінною C. В цьому випадку глобальна прив'язка змінної *COUNT* до 0 замінюється локальною прив'язкою змінної C до 0.

```
CL-USER>(DEFPARAMETER *COUNT* 0)
```

```
CL-USER> (LET ((C *COUNT*)) (INCF C 5) )
```

5

```
CL-USER> *COUNT*
```

0

Глобальні прив'язки можна перезначати. Наприклад, стандартний вивід направлений на екран можна назначити на файл. Для цього треба відкрити файл і пов'язати стандартний потік виводу із файловою поточною змінною.

Приклад 6: У прикладі число 123 записується у файл A.TXT через стандартний вивід, бо функція PRINT записує число без вказівки потоку. Ім'я змінної *N* є спеціальним, бо об'явлено формою DEFPARAMETER. Прив'язка до такої змінної завжди буде динамічною.

```
CL-USER> (DEFUN V ()
```

```
  (DEFPARAMETER *N* (OPEN "A.TXT" :DIRECTION
:OUTPUT :IF-DOES-NOT-EXIST :CREATE))
```

```
  (LET ((*STANDARD-OUTPUT* *N*)) (PRINC 123) (CLOSE
*N*)))
```

V

```
CL-USER> (V)
```

T

У прикладі остання прив'язка до динамічної змінної замінює попередню. Після закінчення роботи функції стандартний потік автоматично відновлюється.

Усі прив'язки до динамічної змінної розміщуються у стеку і після виходу із поточної форми вилучаються із стеку. Внаслідок чого стає видимою попередня прив'язка до глобальної змінної.

Пошук поточної прив'язки глобальної змінної виконується динамічно. Це і причиною назви – *динамічна змінна*. Пов'язування глобальної змінної може змінити поведінку коду, що лежить нижче. В той же час це дозволяє коду, що лежить нижче привласнювати нове значення прив'язки, що лежить вище у стеку.

6.5 Константи

Константи іменовані. Вони завжди глобальні. Константи об'являються формою **DEFCONSTANT**.

(DEFCONSTANT **ім'я** **форма_ініціалізації**
рядок_документації)

Константу неможна застосовувати як формальний параметр або заново пов'язувати із значенням. Часто програмісти, щоб відрізнити константу від змінної починають її ім'я знаком **+**. Прикладом констант є **T** і **NIL**. Константу можна перевизначити, але в цьому випадку код програми треба заново обчислювати.

```
CL-USER> (DEFCONSTANT A 19)
```

```
A
```

```
CL-USER> (DEFCONSTANT A 10)
```

WARNING: (DEFCONSTANT A 10) redefines the constant A. Its old value was 19.

ТЕМА 7: Керування роботою програми

7.1 Макроси керування роботою програми

Часто у програмі треба оброблювати велику кількість ситуацій. Для виявлення певної ситуації аналізується велика кількість параметрів і може виконуватися велика кількість дій.

Застосування у таких програмі двох гілкової спеціальної форми **IF** вимагає вкладеності цих форм, а виконання великої кількості дій вимагає застосування спеціальної форми **PROGN**.

Форма **IF** робить програму не компактною і її важко читати. Щоб полегшити читання і розуміння програми можна застосовувати одно гілкові макроси.

Макрос **WHEN** перевіряє вказану умову. Якщо умова істина, то виконуються всі форми в тілі макросу. Макрос повертає результат останньої виконаної дії.

(WHEN умова форма1...формаN)

Приклад 1: Написати програму, яка одержує дане і аналізує його тип. Якщо введено число, то треба збільшити число на 1. Якщо введений знак - цифра, то треба перевести знак у число і збільшити його на 1. Якщо введений список і перший елемент число, то треба перший елемент збільшити на 1. Інакше треба видати повідомлення про неможливість виконати дії.

У програмі застосовується функція **RETURN-FROM** функція завершує виконання функції і повертає результат роботи макросу **WHEN**.

(RETURN-FROM ім'я_функції результат)

Завдяки функції **RETURN-FROM** програма не робить зайві перевірки умов.

```
CL-USER> (DEFUN NUM (N)
  (WHEN (NUMBERP N) (RETURN-FROM NUM (+ N 1)))
  (WHEN (AND (CHARACTERP N)(>= (CHAR-CODE N) 0)(<=
  (CHAR-CODE N) 9))
  (RETURN-FROM NUM (+ (- (CHAR-CODE N) 48) 1)))
```

```
(WHEN (AND (LISTP N)(NUMBERP (CAR N)))
(RETURN-FROM NUM (+ 1 (CAR N))))
(PRINT "НЕ ЧИСЛО"))
```

```
CL-USER> (NUM 3)
```

```
4
```

```
CL-USER> (NUM #\5)
```

```
6
```

```
CL-USER> (NUM '(7 3))
```

```
8
```

```
CL-USER> (NUM "3")
```

```
НЕ ЧИСЛО
```

Макрос **UNLESS** перевіряє вказану умову. Якщо умова невірна, то виконуються всі форми в тілі макросу. Макрос повертає результат останньої виконаної дії.

(UNLESS умова форма1...формаN)

Приклад 2: Функція вводить знак. Умовою макросу **UNLESS** є знак не цифровий. Тому макрос оброблює тільки знаки цифр. Функція перетворює знак у код і додає до цього числа 1.

```
CL-USER> (DEFUN UN ()
```

```
(SETQ R (READ-CHAR))
```

```
(UNLESS (AND (CHAR> R #\9) (CHAR< R #\0))
```

```
(+ 1 (- (CHAR-CODE R) 48))))
```

```
UN
```

```
CL-USER> (UN)
```

```
4
```

```
5
```

Іноді потрібно одержати код функції користувача. В цьому випадку можна застосувати функцію **SYMBOL_FUNCTION**. Функція працює, якщо середовище збережено або користувач не вийшов із середовища.

```
CL-USER> (SYMBOL_FUNCTION UN)
#<FUNCTION UN NIL (DECLARE (SYSTEM::IN-DEFUN UN))
  (BLOCK UN (SETQ R (READ-CHAR))
    (UNLESS (AND (CHAR> R #9) (CHAR< R #0)) (+ 1 (-
(CCHAR-CODE R) 48))))>
```

Макрос **CASE** застосовується для виконання групи дій за ключем із певної групи ключів.

```
(CASE ключ
(список ключів1 форма11 форма12 ...форма1N)
(список ключів2 форма21 форма22 ...форма2N)
...
(T форми))
```

Ключ задається формою. За замовчанням ключ NIL. В цьому випадку дії не виконуються. Після обчислення значення ключа він шукається у кожному списку ключів. Якщо ключ знайдено, то виконуються вказані дії. Якщо ключ не знайдений в жодному списку, то виконуються дії по T.

Приклад 1: Макрос CASE можна застосовувати для створення програми «ТЕСТ». З програмою працюють 2 групи користувачів викладачі та студенти. Викладачі корегують або додають тести. Студенти тестуються та дивляться свої результати. Кожен студент або викладач має свій пароль - ключ.

```
CL-USER> (DEFUN TEST ()
(PRINC "PAROL? ") (SETQ P (READ))
(CASE P ((VUKL1 VUKL2) (KOREG) (DODAT))
((ST1 ST2 ST3) (TESTUV) (REZ))
(T (PRINC "ПАРОЛЬ ВІДСУТНІЙ "))))
TEST

CL-USER> (DEFUN KOREG () (PRINC "КОРЕГУВАТИ
ТЕСТ"))
KOREG
```

```
CL-USER> (DEFUN DODAT () (PRINC “ДОДАТИ ТЕСТ ”))
DODAT
```

```
CL-USER> (DEFUN TESTUV () (PRINC “ТЕСТУВАННЯ ”))
TESTUV
```

```
CL-USER> (DEFUN REZ () (PRINC “РЕЗУЛЬТАТИ ”))
REZ
```

```
CL-USER> (TEST)
PAROL? ST3
ТЕСТУВАННЯ РЕЗУЛЬТАТИ
```

```
CL-USER> (TEST)
PAROL? ST4
ПАРОЛЬ ВІДСУТНІЙ
```

7.2 Блочна структура функції

Функція може мати всередині кілька блоків для реалізації певної логіки. Кожен блок має мітку (символ); умову, щоб минути цей блок; форми. Блоки виконуються послідовно згідно логіці вказаної в умовах. Якщо умова вірна, то блок минається і виконується наступний блок. Тому функція з блочною структурою виконується швидше. Така структура функції робить логіку функції більш простою і зрозумілою.

(BLOCK мітка умова_виходу_з_блоку форма1 форма2 ...)

Умова реалізується спеціальною формою IF. Для виходу із блоку застосовується функція RETURN-FROM. Якщо результат не вказати, то функція поверне NIL.

(RETURN-FROM мітка результат)

Приклад 1: Нехай треба знайти суму елементів числового списку, якщо його перший елемент більше нуля. Якщо перший елемент списку менше нуля, то треба помножити кожен елемент списку на -1. Якщо перший елемент дорівнює нулю, то залишити список без змін. У кінці кожного блоку функція закінчує свою роботу, бо варіанти не сумісні.

```

CL-USER> (DEFUN REZ (L1) (LET ((L2 NIL))
(BLOCK \+ (IF (<= (CAR L1) 0) (RETURN-FROM \+))
(SETQ L2 (REDUCE \+ L1))
(RETURN-FROM \REZ L2))
(BLOCK \- (IF (ZEROP (CAR L1)) (RETURN-FROM \-))
(DOLIST (a L1 L2)
(PUSH (* -1 a) L2))
(RETURN-FROM \REZ (REVERSE L2)))
(SETQ L2 L1))))
CL-USER> (REZ '(1 3 5))
9
CL-USER> (REZ '(-1 3 5))
(1 -3 -5)
CL-USER> (REZ '(0 3 5))
(0 3 5)
7

```

8 Функції вводу виводу

8.1 Універсальні функції вводу-виводу

Інтерпретатор застосовує універсальні функції вводу-виводу **READ** і **PRINT**. Меню **REPL** має назву за основними функціями інтерпретатору **READ-EVAL-PRINT-LOOP** (Читати вираз, обчислити вираз, вивести результат обчислення, повторити дії в циклі).

Функція **READ** читає будь-який вираз з типізованих елементів. Функція **READ** реалізується через функцію **READ-CHAR**, яка читає знак, тому бінарні дані вона не читає.

(**READ** потік Eof-error-p Eof-value)

EOF-ERROR-P – по замовчанню має значення **T**. По **T** проводиться контроль вводу. Якщо увід переривається за **CTRL-Z**, то повертається помилка **END-OF-FILE**. Якщо значення **FALSE**, то повертається значення **EOF-VALUE**.

Приклад 1: Перший NIL – стандартний потік з клавіатури; другий NIL - EOF-ERROR-P, "END" – значення EOF-VALUE (по замовчанню T).

```
CL-USER> (READ NIL NIL "END")
```

```
^Z
```

```
"END"
```

На ввід чисел впливає вибір основи системи чисел.

Приклад 2: За замовчанням основа системи чисел 10. Основа зберігається у глобальній змінній *READ-BASE*.

```
CL-USER> *READ-BASE*
```

```
10
```

Приклад 3: Змінимо основу системи чисел на двійкову. Введемо число -11.

```
CL-USER> (SETQ *READ-BASE* 2)
```

```
2
```

```
CL-USER> (SETQ N (READ))
```

```
-1001
```

```
-11
```

Приклад 4: Змінна *READ-BASE* на дійсні числа не впливає.

```
CL-USER> (SETQ *READ-BASE* 2)
```

```
2
```

```
CL-USER> (SETQ N (READ))
```

```
3.14
```

```
3.14
```

Приклад 5: Якщо ввести цифру не 0 або 1, то число розглядається як символ.

```
CL-USER> (SETQ N (READ))
```

```
102
```

```
|102|
```

Приклад 6: Увід числа за основою 16.

```
CL-USER> (SETQ *READ-BASE* 16)
```

```
16
```

```
CL-USER> (READ)
```

```
A
```

```
16
```

Призначення іншої основи чисел в інтерпретаторі буде впливати на всі функції.

Приклад 7: Нехай при основі 10 створено означення функції S. Після чого у інтерпретаторі основа буде змінена на 2. Тоді при виклику функції всі числа списку крім 1 і 0 будуть сприйматися як символи.

```
CL-USER> (DEFUN S (L) (+ (CAR L) 5))
```

```
S
```

```
CL-USER> (SETQ *READ-BASE* 2)
```

```
2
```

```
CL-USER> (S '(4 6 8))
```

```
|4| IS NOT A NUMBER
```

Тому краще змінити прив'язку до динамічної змінної *READ-BASE* на 2 у формі LET. Остання прив'язка до динамічної змінної замінює попередню.

```
CL-USER> (+ 5 4)
```

```
9
```

```
CL-USER> (DEFUN BIN () (LET ((*READ-BASE* 2)) (SETQ N  
(READ))))
```

```
BIN
```

```
CL-USER> (BIN)
```

```
111
```

```
7
```

```
CL-USER> (+ 1 4)
```

```
5
```

Після закінчення роботи функції автоматично відновлюється основа 10.

Функція **PRIN1** має 2 аргументи: форму, значення якої виводиться; потік. Якщо виконується вивід на екран, то потік стандартний і тому не вказується.

Функція виводить будь-який вираз з типізованих елементів. Функція **PRIN1** реалізується через функцію **WRITE-CHAR**, яка виводить знак, тому бінарні дані вона не виводить.

Функція **PRIN1** виводить вираз на місце курсору, а після виводу переводить курсор на наступну позицію. Розділові знаки виводяться.

Приклад 7.

```
CL-USER> (DEFUN OUT () (PRINT "23")(PRINT "15")(PRINT
"98"))
OUT
CL-USER> (OUT)
"23""15""98"
```

Функція **PRIN1** має дві модифікації **PRINC** і **PRINT**. Вони відрізняються від функції **PRIN1** лише способом виводу.

Функція **PRINC** виводить вираз на місце курсору, а після виводу переводить курсор на наступну позицію. Вивід рядку залежить від значення змінної ***PRINT-ESCAPE***. Якщо значення змінної **NIL**, то рядки виводить в лапках. При значенні **T** розділові знаки не виводяться (замовчання **T**).

Приклад 8.

```
CL-USER> (DEFUN OUT () (PRINT "23")(PRINT "15")(PRINT
"98"))
OUT
CL-USER> (OUT)
231598
CL-USER> (SETQ *PRINT-ESCAPE* NIL)
NIL
```

```
CL-USER> (DEFUN OUT () (PRINT "23")(PRINT "15")(PRINT
"98"))
```

```
OUT
```

```
CL-USER> (OUT)
```

```
“23”“15”“98”
```

Функція **PRINT** перед виводом пропускає рядок. Виводить вираз з початку рядку, а після виводу переводить курсор на початок наступного рядку. Виводить розділові знаки.

Приклад 9.

```
CL-USER> (DEFUN OUT () (PRINT "23")(PRINT "15")(PRINT
"98"))
```

```
OUT
```

```
CL-USER> (OUT)
```

```
"23"
```

```
"15"
```

```
"98"
```

```
"98"
```

На вивід чисел впливає вибір основи системи чисел

Змінна ***PRINT-BASE*** діє на функції **PRINT**, **PRINC**, **PRIN1**.

Приклад 10.

```
CL-USER> (SETQ *PRINT-BASE* 2)
```

```
10
```

```
CL-USER> (+ 8 2)
```

```
1010
```

8.2 Функції вводу

Функція **READ-LINE** вводить будь-який тип даного, перетворює його у рядок і повертає його.

Приклад 1: Вводиться число.

```
CL-USER> (SETQ N (READ-LINE))
```

```
123456789123
```

```
"123456789123"
```

Приклад 2: Вводиться список.

```
CL-USER> (SETQ N (READ-LINE))
```

```
( 1 3 5 7)
```

```
“(1 3 5 7)”
```

Приклад 3: Одержаний із функції **READ-LINE** рядок цифр перетворюється функцією **READ-FROM-STRING** в число.

```
CL-USER> (+ 5 (READ-FROM-STRING (READ-LINE)))
```

```
20
```

```
25
```

Приклад 4: Вводиться знак.

```
CL-USER> (SETQ N (READ-LINE))
```

```
#\r
```

```
“#\r”
```

Приклад 5: Вводиться рядок. Функція повертає рядок, в якому введені лапки вважаються символами рядку. Вони помічаються знаком «\».

```
CL-USER> (SETQ N (READ-LINE))
```

```
“fgh”
```

```
“\“fgh\””
```

Функція **READ-CHAR** вводить і повертає один знак. Функція не може ввести **#CAR**, бо для неї це 3 знаки.

Приклад 6.

```
CL-USER> (SETQ Z (READ-CHAR))
```

```
h
```

```
#\h
```

```
CL-USER>(SETQ Z (READ-CHAR))
```

```
9
```

```
#\9
```

Функція **READ-BYTE** читає один байт з бінарних даних файлу. Якщо у файлі S.bin перший байт містить код 49, то форма LET повертає знак #\1.

Приклад 7.

```
CL-USER> (LET ((IN (OPEN "D:\\TEMP\\S.BIN" :DIRECTION :INPUT
:ELEMENT-TYPE '(UNSIGNED-BYTE 8))))
(PRINT (CODE-CHAR (READ-BYTE IN))) (CLOSE IN))
#\1
T
```

8.3 Функції виводу

Функцію **WRITE** використовують для виводу даних будь-якого типу крім бінарного. Робота функції залежить від значення динамічної змінної ***PRINT-ESCAPE***. Якщо значення змінної NIL, розділові знаки не виводяться. При значенні T розділові знаки виводяться (замовчання T).

Приклад 1: Функція виводить число.

```
CL-USER> (WRITE 142)
142
```

Приклад 2: Функція виводить рядок без розділових знаків.

```
CL-USER> (SETQ *PRINT-ESCAPE* NIL)
NIL
```

```
CL-USER> (SETQ N (WRITE "142"))
142
```

```
CL-USER> (+ N 2)
+: "123" IS NOT A NUMBER
```

Приклад 3: Функція виводить знак Q без розділових знаків #\.

```
CL-USER> (SETQ *PRINT-ESCAPE* NIL)
NIL
CL-USER> (SETQ N (WRITE #\Q))
Q
```

Приклад 4: Функція виводить список.

```
CL-USER> (SETQ N (WRITE '(1 2 3)))
```

```
(1 2 3)
```

Приклад 5: Функція працює, як аналог стандартної функції SYMBOL-FUNCTION. Вона друкує і повертає зміст означення функції.

```
CL-USER> (DEFUN MULT (X Y) (* X Y))
```

```
MULT
```

```
CL-USER> (WRITE #\MULT)
```

```
#<FUNCTION MULT (X Y) (DECLARE (SYSTEM::IN-DEFUN  
MULT)) (BLOCK MULT (* X Y))>
```

```
#<FUNCTION MULT (X Y) (DECLARE (SYSTEM::IN-DEFUN  
MULT)) (BLOCK MULT (* X Y))>
```

Порівняйте результати функції WRITE, якщо функцію задати символом. Треба розрізняти символ – ім'я функції і знак - ім'я функції. Що застосовувати залежить від функції, у тілі якої використовується ім'я функції.

```
CL-USER> (WRITE 'MULT)
```

```
MULT
```

```
MULT
```

Функції **WRITE-STRING** і **WRITE-LINE** виводять тільки рядки. Обидві функції не виводять розділові знаки. Вивід функцій не залежить від змінної *PRINT-ESCAPE*. Функції відрізняються тільки способом виводу.

Приклад 1: Функція WRITE-STRING після виводу переводить курсор в наступну позицію.

```
CL-USER> (LET ((A "AS") (B "BN"))
```

```
(WRITE-STRING A)(WRITE-STRING B))
```

```
ASBN
```

Приклад 2: Функція WRITE-LINE після виводу переводить курсор в початок наступного рядку.

```
CL-USER> (LET ((A "AS") (B "BN")))
          (WRITE-LINE A)(WRITE-LINE B))
```

```
AS
```

```
BN
```

Функція **WRITE-CHAR** виводить знак без розділових знаків #\ і повертає знак.

```
CL-USER> (WRITE-CHAR #\1)
```

```
1
```

```
#\1
```

Функція **WRITE-BYTE** виводить вміст байту без розділових знаків у файл. Після чого читає з бінарного файлу цей байт, перетворює його вміст у знак і виводить на екран.

```
CL-USER> (DEFUN BIN () (LET ((OUT (OPEN "D:\\TEMP\\S.BIN"
: DIRECTION :OUTPUT :IF-EXISTS :SUPERSEDE :ELEMENT-TYPE
'(UNSIGNED-BYTE 8))))
```

```
  (WRITE-BYTE 65 OUT) (CLOSE OUT))
```

```
  (LET ((IN (OPEN "D:\\TEMP\\S.BIN" :DIRECTION :INPUT
:ELEMENT-TYPE '(UNSIGNED-BYTE 8))))
```

```
    (PRINT (CODE-CHAR (READ-BYTE IN))) (CLOSE IN)))
```

```
BIN
```

```
CL-USER> (BIN)
```

```
#\A
```

9 Файли та потокова організація роботи з ними

9.1 Поняття потоку

Увід-вивід даних здійснюється через функції вводу-виводу і потокову організацію. З'ясуємо, що таке потокова організація.

У різних операційних системах застосовуються наступні одиниці роботи. Дивися таблицю 7.1.

Таблиця 7.1

OS/2	WINDOWS NT	UNIX
Задача	Процес	Код
Під задача	Потік	Породжений код

Процеси (програмні коди) ізольовані один від одного. Потоки більш дрібні одиниці роботи ніж процеси.

Потоки – це *динамічні об'єкти певного класу*, що створюються під час виконання коду процесу. Потоки бувають системні і користувача. Системні потоки створює ядро операційної системи. Системні потоки – це потоки команд. Потоки користувача створює код програми користувача і це потоки даних. Процес може породжувати кілька потоків, які існують одночасно. Непотрібні потоки вилучаються.

Операційна система планує час ядер процесору між потоками і їй все одно до якого процесу належить потік. Важливий тільки пріоритет потоку.

Нас цікавлять тільки потоки вводу-виводу даних користувача.

Достоїнства застосування потоків: Робота з потоками значно полегшує роботу з конкретним пристроєм. Але потік вводу-виводу не може звертатися до пристрою безпосередньо. Для цього він використовує сервіс ОС – API (програмний інтерфейс).

Потоки дозволяють застосовувати у програмі достатньо простий єдиний синтаксис. Об'єкт - потік одержує з програми параметри, деталізує звертання відповідно необхідному пристрою і передає їх сервісу ОС. До таких параметрів відносяться напрям потоку, тип даних тощо. Після чого сервіс ОС реалізує взаємодію з

пристроєм. Через API створюються також черги до кожного пристрою.

Застосування потоків забезпечує надійність роботи програми. Потоки дають можливість одночасно читати дані з одного файлу кільком програмам. Крім того є можливість читати із однієї програми за кількома потоками із одного файлу.

Недоліки застосування потоків: Потоки зменшують швидкість роботи програми. Об'єкт – потік розрахований на всі варіанти пристроїв та файлів, що збільшує об'єм програми і зменшує швидкість роботи програми.

Але при роботі з пристроями застосовується буферизація в ОП. Це значно прискорює роботу з даними. Вивід даних на пристрій виконується тільки по заповненню буферу або по закриттю файлу. Об'єм буферу залежить від пристрою (драйвера), а він завжди кратний 128 кбайт.

9.2 Робота з файлами через потоки

9.2.1 Увід-вивід через стандартні потоки

Читання та запис з використанням стандартних пристроїв не вимагає створення потоків. Для цього існують стандартні потоки.

По замовчанню для вводу з клавіатури застосовується стандартний потік **STANDARD-INPUT**. Замовченням для виводу на екран є стандартний потік **STANDARD-OUTPUT**. Для клавіатури та екрану можна застосовувати той самий потік **TERMINAL-IO**. Стандартні потоки можна застосувати для перепризначення стандартних потоків на файли тощо.

Увід з клавіатури та вивід на екран в мові LISP виконується універсальними функціями READ і PRINT. Через стандартні потоки функції працюють із будь-якими типами даних крім бінарних.

Розглянемо приклади роботи із стандартними потоками.

Приклад 1: Дані виклики тотожні. Аналогічно працює стандартний вивід. Потік NIL вказує на стандартний пристрій.

```
CL-USER> (SETQ R (READ))
```

23

23

CL-USER> (SETQ R (READ *STANDARD-INPUT*))

23

23

CL-USER> (SETQ R (READ NIL))

23

23

CL-USER> (PRIN1 'lisp)

LISP

LISP

CL-USER> (PRIN1 'lisp *STANDARD-OUTPUT*)

LISP

LISP

Приклад 2: Можна застосовувати той самий стандартний потік для вводу та виводу, але в означенні функції. Інтерпретатор працює за циклом REPL і тому зациклює увід.

```
CL-USER> (DEFUN VV () (PRINT (READ *TERMINAL-IO*)
*TERMINAL-IO*))
```

VV

CL-USER> (VV)

23

23

9.2.2 Функції обробки файлів

Потоки класифікують:

- по типу інформації, яку передають (символьна, бінарна);
- по направленню інформації (вхідні, вихідні, двунаправленні);
- по призначенню (файлові, діалогові);
- по стану (відкриті закриті).

У цьому розділі ми будемо працювати з усіма переліченими типами потоків крім діалогових.

При читанні із файлу або запису в нього вказують файловий потік через який будуть передаватися дані. Потік – об’єкт створюється (відкривається) функцією **OPEN**. У програмі

(OPEN “шлях до файлу та ім’я” напрям_руху_даних тип_елементу_дії_за_ситуаціями)

Функція **OPEN** створює потік за вказаними параметрами і повертає посилку на потік. Тому посилку треба відразу привласнити змінній. На пристрою за шляхом відкривається файл.

Всі дані записуються у файл або читаються з файлу через буфер пристрою. Вивід даних у файл або читання даних з файлу здійснюється після заповнення буферу. Це значно прискорює роботу з даними. Після передачі даних у буфер може статися ситуація, що буфер не заповнено. Тому залишок виводиться функцією **CLOSE**.

Функція **CLOSE** виводить залишок буферу в файл, закриває файл, вилучає потік.

Приклад 1: Функція **Open** відкриває файл і створює потік, посилка на який розміщується у змінній **IN**. За потоком читається рядок із файлу і виводиться на екран. Після чого функція закриває файл, при цьому вилучається потік.

```
CL-USER> (LET ((IN (OPEN "D:\TEMP\STUD.TXT"))))
(PRINT (READ-LINE IN)) (CLOSE IN))
“LISP”
T
```

Для вказівки напрямку руху даних у функції **OPEN** існує ключ **:DIRECTION**:

- значення ключа **:INPUT** вказує на потік для читання;
- значення ключа **:OUTPUT** вказує на потік для запису;
- значення ключа **:probe** вказує, що дії виконуватимуться над всім файлом, а не над його даними;
- значення ключа **:IO** вказує на потік для запису та читання.

Для читання та запису можна також відкрити 2 потоки – один для читання, другий для запису.

У прикладі 1 напрям не вказується він береться по замовчанням :INPUT.

Макрос **WITH-OPEN-FILE** дозволяє не застосовувати функції OPEN та CLOSE. Макрос ініціює їх за своїми параметрами автоматично. Макрос WITH-OPEN-FILE автоматично закриває файл навіть при збою. Тіло макросу містить спеціальну форму UNWIND-PROTECT, яка приводить в порядок стек після збою. Закриття файлу необхідно також для виводу залишку буферу у файл.

(WITH-OPEN-FILE (потік “шлях до файлу та ім’я” напрям_руху_даних тип_елементу_дії_за_ситуаціями) форми)

Приклад 2: Замість функцій OPEN та CLOSE застосовується макрос WITH-OPEN-FILE.

```
CL-USER> (WITH-OPEN-FILE (IN "D:\\TEMP\\STUD.TXT")
(PRINT (READ-LINE IN)) )
“LISP”
“LISP”
```

При помилці може виникнути ситуація, коли файл не закритий. Якщо в програмі застосовується функція OPEN, при повторному запуску програми виникає збійна ситуація. Закрити файл можна прямо в інтерпретаторі:

```
CL-USER> (CLOSE IN)
T
```

Функція **LISTEN** перевіряє чи є у потоці дані. Якщо дані є вона повертає T. Якщо даних у потоці немає, повертає NIL.

(LISTEN потік)

Приклад 3: Нехай у текстовому файлі записано цифри 1 2 3 4. Форма (NOT (LISTEN IN)) повертає T, якщо дані у потоці закінчилися.

```
CL-USER> (WITH-OPEN-FILE (IN "D:\\TEMP\\S1.TXT" :DIRECTION
:INPUT)
```

```
(LOOP (IF (NOT (LISTEN IN))(RETURN T))
(PRIN1 (READ IN)) ) )
1234T
```

Приклад 4: Функція TXT за ключем :DIRECTION і його значенням:OUTPUT виконує запис у файл. По замовчанню тип елементу CHARACTER, тому тип елементу не вказується. Ключ IF-EXISTS :SUPERSEDE дозволяє не переривати роботу програми, якщо вказаний файл вже існує, а замінити файл.

```
CL-USER> (DEFUN TXT () (LET ((OUT (OPEN "D:\\TEMP\\S.TXT"
:DIRECTION :OUTPUT :IF-EXISTS :SUPERSEDE )))
```

```
(DOTIMES (N 3 T)
(WRITE-CHAR (READ-CHAR) OUT) ) (CLOSE OUT)))
TXT
```

```
CL-USER> (TXT)
```

```
abc
```

```
T
```

Вміст файлу S.txt: abc. Вводяться відразу 3 символи, а WRITE-CHAR записує по одному знаку.

```
CL-USER> (TXT)
```

```
a
```

```
b
```

```
T
```

Вводяться 3 символи: a BK b.

Приклад 5: Читається файл створений у прикладі 4. Потік створюється у формі LET. Тому змінна IN існує тільки у цій формі.

```
CL-USER> (LET ((IN (OPEN "D:\\TEMP\\S.TXT" :DIRECTION
:INPUT)))
```

```
(PRIN1 (READ IN))(CLOSE IN))
```

```
ABC
```

```
T
```

ОСОБЛИВІ СИТУАЦІЇ

При запису у файл або читанні файлу можуть виникати особливі ситуації.

При спробі відкрити файл для читання, виявляється, що файл відсутній. Якщо задати у функції OPEN ключ IF-DOES-NOT EXIST, то можливо створити файл з вказаним ім'ям і відкрити його знову за значенням ключа :CREATE; повернути NIL при неможливому створенні потоку; закінчити роботу програми за помилкою :ERROR (замовчення).

При спробі відкрити файл для запису, виявляється, що файл вже існує. Якщо задати у функції OPEN ключ IF-EXISTS, то можливо доповнити файл – значеннями :APPEND для до запису файлу; :OVERWRITE для перезапису певних даних у файлі; SUPERSEDE для зміни файлу новим; повернути NIL при неможливому створенні потоку.

Приклад 6: Функція R повинна прочитати дані з файлу S.TXT. Якщо файл відсутній, то замість посилки на потік повернеться NIL. При наявності файлу вміст файлу числа у символьному виді: 1 2 3. Функція READ читає 3 числа і перетворює їх у числа. Після чого знаходить суму чисел.

У функції потік задається глобальною змінною *OUT*. Після призначення глобальній змінній посилки на потік, значення змінної *OUT* існує у всій програмі і не може змінитися. Зверніть увагу, що NIL – не значення ключа, а вказує на відсутність ключа.

```
CL-USER> (DEFVAR *OUT*)
```

```
CL-USER> (DEFUN R () (SETQ *OUT* (OPEN "D:\\TEMP\\S.TXT"
:DIRECTION :INPUT :IF-DOES-NOT-EXIST NIL))
```

```
  (IF (NULL *OUT*) (RETURN-FROM S NIL))
```

```
  (SETQ A1 (READ *OUT*)) (SETQ A2 (READ *OUT*))
```

```
  (SETQ A3 (READ *OUT*)) (CLOSE *OUT*) (+ A1 A2 A3))
```

```
R
```

```
CL-USER> (R)
```

Приклад 7: Функція R створює файл S.TXT, якщо файл відсутній. Функція вводить дані з клавіатури в циклі. Закінчити увід можна за ^Z. Кінець вводу контролюється у циклі. Якщо не кінець вводу, то введене дане записується у файл. По кінцю даних (змінна A містить T) файл закривається і програма закінчує свою роботу.

При наявності файлу функція дописує дані в кінець файлу. Увід даних з клавіатури виконується так само.

У виклику функції READ перший NIL вказує на стандартний ввід; другий NIL вказує, що по CTRL-Z ввід закінчується без помилки; T повертається по закінченню вводу.

Ключ :DIRECTION має значення :IO, що дозволяє :APPEND застосовувати :INPUT для знаходження кінця файлу, а :CREATE застосовувати :OUTPUT для запису у файл.

```
CL-USER> (DEFUN R () (LET (A (N (OPEN "D:\\TEMP\\S.TXT"
:DIRECTION :IO :IF-EXISTS :APPEND :IF-DOES-NOT-EXIST
:CREATE))))
```

```
(LOOP
```

```
(SETQ A (READ NIL NIL T))
```

```
(IF (NOT (EQL A T)) (PRIN1 A N) (PROGN (CLOSE N)
(RETURN-FROM R T))))))
```

```
R
```

```
CL-USER> (R)
```

```
COM
```

```
MON
```

```
^Z
```

```
T
```

```
CL-USER> (R)
```

```
LI
```

```
SP
```

```
^Z
```

```
T
```

Для установки показника на байт файлу застосовують функцію FILE-POSITION. Якщо замість позиції вказати вільну змінну, то можна одержати вміст поточного байту.

(FILE-POSITION потік позиція_байту)

Приклад 8: Нехай треба перезаписати 2 байт файлу. Створимо файл S.txt блоком. Вміст файлу: abcd.

```
CL-USER> (DEFUN S () (WITH-OPEN-FILE (N
"D:\\TEMP\\S.TXT" :DIRECTION :IO :IF-EXISTS :OVERWRITE)
(FILE-POSITION N 2) (PRIN1 9 N)(READ N)))
```

S

CL-USER> (S)

D

Вміст файлу після виконання функції S: ab9d.

Елементи даних подаються певним типом. Тип елемента залежить від пристрою і від того, яким типом записані дані. Для роботи з бінарними даними вказують ключ та його код :ELEMENT-TYPE '(UNSIGNED BYTE 8).

Приклад 9: Розглянемо приклад, в якому створюється бінарний файл. У циклі Змінна A послідовно приймає значення 0...3. Код 0 – 48. Тому у файл запишуться 48, 49, 50, 51. Такий бінарний файл можна прочитати текстовим редактором.

```
CL-USER> (WITH-OPEN-FILE (OUT "D:\\TEMP\\S1"
:DIRECTION :OUTPUT :ELEMENT-TYPE '(UNSIGNED-BYTE 8))
(DOTIMES (A 4 T)
(WRITE-BYTE (+ 48 A) OUT) ) )
```

9.2.3 Робота над файлами

При необхідності функція користувача може виконувати дії не над даними файлу, а над самим файлом.

Приклад 1: Функція PROBE-FILE перевіряє чи існує файл D:\\TEMP\\S1.TXT. Якщо файл не існує повертає NIL. Для роботи функції потік непотрібен.

```
CL-USER> (DEFUN ISN () (PROBE-FILE "D:\\TEMP\\S1.TXT"))
#P"D:\\temp\\S1.TXT"))
```

Приклад 2: Функція працює із інтерпретатора.

```
(PROBE-FILE "D:\\TEMP\\S1.TXT")
#P"D:\\temp\\S1.TXT"
```

Приклад 3: У формі WITH-OPEN-FILE автоматично перевіряється за значенням :PROBE, чи існує файл. Якщо файл не існує, то повертається NIL. Інакше файл D:\\TEMP\\S1.TXT вилучається і повертається шлях до файлу.

```
CL-USER> (WITH-OPEN-FILE (IN "D:\\TEMP\\S4.TXT"
:DIRECTION :PROBE) (DELETE-FILE "D:\\TEMP\\S4.TXT"))
#P"D:\\TEMP\\S4.TXT"
```

Нижче подані функції працюють аналогічно показаному у прикладах 1-3.

```
CL-USER> (DELETE-FILE "D:\\TEMP\\S1.TXT")
#P"D:\\temp\\S1.TXT"))
```

```
CL-USER> (RENAME-FILE "D:\\TEMP\\S.BIN"
"D:\\TEMP\\A.BIN")
#P"D:\\TEMP\\A.BIN" ;
#P"D:\\TEMP\\S.BIN" ;
#P"D:\\TEMP\\A.BIN"
```

```
CL-USER> (FILE-LENGTH "D:\\TEMP\\S1")
4
```

```
CL-USER> (WITH-OPEN-FILE (IN "D:\\TEMP\\S1.TXT"
:DIRECTION :INPUT)(FILE-LENGTH IN))
4
```

10 Функціонали

Функціонал – це функція, яка має один або кілька функціональних аргументів, або може повертати функцію.

Стандартні функціонали розподіляються на типи: застосовні функціонали; відображуючі функціонали; предикати планування.

Функціонали одержують функціональний аргумент, який застосовують до переданих йому аргументів.

10.1 Застосовні функціонали

Функціонал **APPLY** може мати скільки завгодно аргументів, якщо вони необхідні для функціонального аргументу. Але останнім обов'язково повинен бути список. Застосування списку дозволяє оброблювати будь-яку кількість аргументів. Тип аргументів залежить від типу функціонального аргументу.

Функціонал **APPLY** застосовує функціональний аргумент до кожного елементу списку, переданого йому. Функція може бути стандартною або функцією користувача.

(APPLY функціональний_аргумент елемент1 елемент2 ...список)

Приклад 1: Функція “+” може мати будь-яку кількість числових аргументів. Але останній аргумент повинен бути списком, навіть у випадку порожнього списку.

```
CL-USER> (APPLY '+ 1 2 3 NIL)
```

```
6
```

```
CL-USER> (APPLY '+ 1 2 3 '(4 5))
```

```
15
```

Приклад 2: Функціонал застосовує функцію **CONS** до кожного елементу списку. Тому повертається точкова пара, а не список.

```
CL-USER> (APPLY 'CONS 1 '(2))
```

```
(1 . 2)
```

Приклад 3: Функціонал застосовує функцію CONS до перших двох елементів, якщо третій аргумент порожній список. Інакше помилка. Функція CONS має тільки 2 аргументи.

```
CL-USER> (APPLY 'CONS 1 '(2 3) '())
(1 2 3)
```

Приклад 4: Функціонал застосовує функцію APPEND до перших двох аргументів, а потім послідовно до елементів останнього списку.

```
CL-USER> (APPLY 'APPEND '(1) '(2 3)'((4) (5)))
(1 2 3 4 5)
```

Функціонал **FUNCALL** застосовують у випадку, коли відома кількість аргументів. Аргументи мають відповідати типу і кількості аргументів функції, що застосовується.

Приклад 1: Напишемо універсальну функцію, яка виконує 4 арифметичні дії за допомогою функціоналу FUNCALL.

```
CL-USER> (DEFUN A (F N1 N2) (FUNCALL F N1 N2))
A
CL-USER> (A '* 3 4)
12
CL-USER> (A '/' 3 4)
3/4
CL-USER> (A '+' 3 4)
7
CL-USER> (A '-' 3 4)
-1
```

Функціонал FUNCALL не може застосовувати макрос або λ -вираз

10.2 Відображуючи функціонали

Функціонали цього типу відображують кілька списків у один за певною функцією.

Функціонал **MAPCAR** застосовує послідовно функцію до CAR-елементів списків, потім до CADR - елементів списків і так далі, поки не закінчиться хоч один список. З результатів складається кінцевий список.

(MAPCAR функціональний_аргумент список1 список2...)

Приклад 1: Найкоротший список має 2 елементи, тому результат має теж 2 елементи

```
CL-USER> (MAPCAR '+ '(1 2 3) '(4 5 6) '(7 8 9 1))
(12 15 18)
```

Виконуються дії:

```
(CAR '(1 2 3) ->1 (CAR '(4 5 6) ->4 (CAR '(7 8 9 1) ->7
(+ 1 4 7)-> 12
```

```
(CADR '(1 2 3) ->2 (CADR '(4 5 6) ->5 (CADR '(7 8 9 1) ->8
(+ 2 5 8) -> 15
```

```
(CADDR '(1 2 3) ->3 (CADDR '(4 5 6) ->6 (CADDR '(7 8 9 1) ->9
(+ 3 6 9)-> 18
```

Приклад 2: Порожній список не містить елементів, тому результат NIL.

```
CL-USER> (MAPCAR '+ '(1 2 3) '(4 5 6) '())
NIL
```

Приклад 3:

```
CL-USER> (MAPCAR 'CONS '(місто стоїть річка) '(іменник
дієслово іменник))
((місто . іменник) (стоїть . дієслово) (річка . іменник))
```

Функціонал **MAPLIST** застосовує послідовно функцію до всіх списків, потім до CDR - списків і так далі, поки не закінчиться хоч один список. З результатів складається кінцевий список.

(MAPLIST функціональний_аргумент список1 список2...)

Приклад 4: Функція REVERSE працює тільки з одним списком.

```
CL-USER> (MAPLIST 'REVERSE '(1 2 3))
((3 2 1) (3 2) (3))
```

Виконуються дії:

```
(REVERSE '1 2 3) -> (3 2 1)
(REVERSE (CDR '1 2 3)) -> (3 2)
(REVERSE (CDDR '1 2 3)) -> (3)
```

З результатів складається один список ((3 2 1) (3 2) (3)).

Приклад 5: Функція працює з будь-якою кількістю списків.

```
CL-USER> (MAPLIST 'APPEND '(1 2 3)'(6 7))
((1 2 3 6 7) (2 3 7))
```

Виконуються дії:

```
(APPEND ( 1 2 3)'(6 7)) -> (1 2 3 6 7)
(APPEND (CDR ( 1 2 3)) (CDR '(6 7))) -> ( 2 3 6 7)
```

З результатів складається один список ((1 2 3 6 7) (2 3 7))

Функціонал **MAPCAN** застосовує послідовно функцію до CAR-елементів списків, потім до CADDR - елементів списків і так далі, поки не закінчиться хоч один список. Результати зливаються у кінцевий список функцією APPEND.

Приклад 6: Порівняйте результати виконання функціоналів MAPCAN і MAPCAR при застосуванні функції List до тих самих списків.

```
CL-USER> (MAPCAN 'LIST '(1 2 3)'(4 5))
(1 4 2 5)
```

Виконуються дії:

```
(LIST 1 4)->(1 4)
(LIST 1 4)->(2 5)
```

В результати списки зливаються в один (1 4 2 5)

```
CL-USER> (MAPCAR 'LIST '(1 2 3)'(4 5))
((1 4) (2 5))
```

Функціонал **MAPCON** застосовує послідовно функцію до всіх списків, потім до CDR - списків і так далі, поки не закінчиться хоч один список. Результат зливаються у кінцевий список функцією APPEND.

Приклад 7: Порівняйте результати виконання функціоналів MAPCON і MAPLIST при застосуванні функції REVERSE до тих самих списків.

```
CL-USER> (MAPCON 'REVERSE '(1 2 3))
```

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

```
CL-USER> (MAPLIST 'REVERSE '(1 2 3))
```

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

10.3 Предикати планування

Предикати планування – це група функціоналів, що застосовують предикати для групової перевірки типів елементів списків.

Функціонал **EVERY** перевіряє кожен елемент всіх списків за тестом - предикатом. Якщо всі елементи відповідають тесту, то повертає Т, інакше NIL. Вибір тесту залежить від кількості списків та типів елементів.

```
CL-USER> (EVERY 'PLUSP '(1 3 6))
```

```
T
```

```
CL-USER> (EVERY 'ODDP '(1 3 6))
```

```
NIL
```

```
CL-USER> (EVERY '=' '(2 2) '(2 2))
```

```
T
```