

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Запорізький національний технічний університет

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни

«Функціональне та логічне програмування»

«Функціональне програмування»

ч.1

для студентів спеціальності

121 Інженерія програмного забезпечення»

освітня програма

«Програмне забезпечення систем»

усіх форм навчання

Конспект лекцій з дисципліни «Функціональне та логічне програмування» «Функціональне програмування» ч.1 для студентів спеціальності 121 «Інженерія програмного забезпечення» освітня програма «Програмне забезпечення систем» усіх форм навчання /Укл.:І.В.Левада.- Запоріжжя: ЗНТУ, 2016.-90с.

Укладач: старший викладач І.В. Левада,

Рецензент: доц., к.т.н. А.В. Пархоменко

Відповідальний за випуск: старший викладач І.В. Левада

Затверджено на засіданні
НМК КНТ- факультету
від 23 жовтня 2016р.

Затверджено на засіданні
кафедри ПЗ протокол №4
від 4 жовтня 2016 р.

ЗМІСТ

ПЕРЕДМОВА	5
ТЕМА 1: ПАРАДИГМА ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ	6
1.1 ПОНЯТТЯ ПАРАДИГМИ ПРОГРАМУВАННЯ.....	6
1.2 ПАРАДИГМА ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ.....	6
ТЕМА 2: ПРИНЦИПИ ПОБУДОВИ ТА ФУНКЦІОНУВАННЯ МОВИ LISP	8
2.1 Виникнення мови LISP.....	8
2.2 Середовище мови LISP.....	9
2.3 Декларативність мови LISP.....	10
2.4 LISP – мова функціонального програмування.....	10
2.5 LISP - мова обробки списків.....	13
2.6 Уніфікований підхід до даних і програм.....	14
2.7 Відкритість мови.....	16
2.8 Динамічна типізація змінних.....	17
2.9 LISP - мова обробки символічної інформації.....	18
2.10 LISP - РЕКУРСИВНА мова.....	18
2.11 Принцип модульності.....	19
2.12 Сучасний стан розробок діалектів мови LISP та їх вплив на ПРОЕКТ COMMON LISP.....	19
ТЕМА 3: СТАНДАРТ COMMON LISP.....	21
ТЕМА 4: СИСТЕМА ПРОГРАМУВАННЯ МОВИ PRACTICAL COMMON LISP	23
4.1 Текстовий редактор EMACS як основа системи ПРОГРАМУВАННЯ.....	24
4.1.1 Загальні властивості Emacs.....	24
4.1.2 Фрейми.....	24
4.1.3 Буфери та їх режими роботи.....	25
4.1.4 Рядок режиму буфера.....	26
4.1.5 Область луни.....	28
4.1.6 Меню текстового редактору.....	28
4.1.7 Команди текстового редактору.....	29
4.1.8 Ключі та команди EMACS.....	31
4.1.9 Міні буфер.....	32
4.1.10 Команди роботи з текстом.....	33
ТЕМА 5: ОСНОВНІ ПОНЯТТЯ COMMON LISP	37
5.1 АБСТРАКТНІ ТИПИ ДАНИХ.....	37
5.2 ПОНЯТТЯ СЕРЕДОВИЩА COMMON LISP І ЙОГО ВМІСТ.....	39
5.3 Символи.....	42

5.4 АТОМИ І СПИСКИ	45
5.5 ІНТЕРПРЕТАТОР COMMON LISP	49
ТЕМА 6: ФУНКЦІЇ, СПЕЦІАЛЬНІ ФОРМИ ТА МАКРОСИ	52
6.1 БАЗОВІ ФУНКЦІЇ	52
6.2 ПОНЯТТЯ СПЕЦІАЛЬНОЇ ФОРМИ	56
6.3 МАКРОСИ	57
6.4 ФУНКЦІЇ КОМПАРАТОРУ	58
6.5 ФУНКЦІЇ ПРЕДИКАТИ	63
6.6 ЛОГІЧНІ ФУНКЦІЇ	66
ТЕМА 7: УЗАГАЛЬНЕННЯ ПОНЯТТЯ «ЗМІННА»	67
7.1 ЗАСТОСУВАННЯ ФОРМИ У ЯКОСТІ «МІСЦЯ»	67
7.2 МАКРОСИ РОБОТИ З «МІСЦЕМ»	68
ТЕМА 8: КОНСТРУКЦІЇ, ЩО КЕРУЮТЬ ВИКОНАННЯМ	
ПРОГРАМИ	71
8.1 СПЕЦІАЛЬНА ФОРМА QUOTE	71
8.2 СПЕЦІАЛЬНА ФОРМА PROGN	71
8.3 СПЕЦІАЛЬНА ФОРМА IF	71
8.4 СПЕЦІАЛЬНА ФОРМА COND	72
ТЕМА 9: ФОРМАЛЬНІ ТА ФАКТИЧНІ ПАРАМЕТРИ	74
9.1 ОБОВ'ЯЗКОВІ ПАРАМЕТРИ	74
9.2 НЕОБОВ'ЯЗКОВІ КЛЮЧОВІ ПАРАМЕТРИ	74
9.3 ІМЕНОВАНІ КЛЮЧОВІ ПАРАМЕТРИ	75
9.4 ОСТАТОЧНІ КЛЮЧОВІ ПАРАМЕТРИ	77
ТЕМА 10: РЕКУРСІЯ	78
10.1 ВИСХІДНИЙ МЕТОД РЕКУРСІЇ	78
10.2 НИЗХІДНИЙ МЕТОД РЕКУРСІЇ	80
10.3 ПАРАЛЕЛЬНА РЕКУРСІЯ	81
ТЕМА 11: ЦИКЛИ	84
11.1 МАКРОСИ DO	84
11.2 МАКРОС LOOP	87
ЛІТЕРАТУРА	90
ОСНОВНА	90
ДОДАТКОВА	90

ПЕРЕДМОВА

Стандарт мови програмування Common Lisp – це опис, а не програмна реалізація. Для стандарту Common Lisp не існує однієї програмної реалізації, яку вважають еталоном, як у інших сучасних мовах програмування.

Існує багато реалізацій стандарту Common Lisp, у яких ядро будується за стандартним описом (Allegro Common Lisp, Ppcre Common Lisp, Steel Bank Common Lisp тощо). Тому у дисципліні «Функціональне програмування» вивчається стандарт Common Lisp, а реалізацією стандарту обрано мову Practical Common Lisp яка вільно розповсюджується і сьогодні є мовою промислового призначення.

Мова Common Lisp є діалектом Lisp – сімейства. Для тих, хто знає можливості хоча б однієї мови з Lisp – сімейства, відомі її переваги перед іншими мовами. Lisp вбирає в себе все найкраще, що має програмування на сьогодні. Ніякі вдосконалення мов C++ і Pascal не дозволять реалізувати всі методи програмування, властиві лісноподібним мовам.

Діалекти Lisp – сімейства базуються на функціональній парадигмі програмування. Діалекти відрізняються один від одного настільки, що кожен діалект вважають окремою мовою. Проте, не кожен діалект сімейства відноситься до «чистих» мов, що суворо придержуються принципів функціонального програмування. Сучасне практичне програмування часто вимагає реалізації методів традиційного програмування.

Мову Practical Common Lisp застосовують для розв'язку інженерних задач, задач штучного інтелекту. Мовою Practical Common Lisp реалізують модулі для WEB - застосувань; текстові редактори; створюють експертні системи, програми для розуміння природних мов, перекладачі; розробляють спеціалізовані мови вищого рівня; програми спілкування з реляційними базами даних, тощо.

У мережі Internet існує багато бібліотек з відкритим для користувачів кодом. Вміст бібліотек дозволяє значно розширювати базові можливості реалізацій Common Lisp. Мобільність мови дозволяє легко розширювати її. Метою вивчення Common Lisp є одержання знань і практичних навиків роботи з сучасним діалектом мови Lisp, показ мобільності мови Practical Common Lisp.

ТЕМА 1: ПАРАДИГМА ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ

1.1 Поняття парадигми програмування

Під парадигмою програмування розуміють спільний для групи мов програмування спосіб мислення над програмою в однакових термінах і спільний спосіб програмування. В основі кожної такої групи мов лежить одна обчислювальна модель.

Парадигма програмування характеризується: засобами опису та подавання даних, методами і ефективністю цих методів до розв'язку певних класів задач.

Для алгоритмічних мов обчислювальною моделлю є абстрактна машина Тьюринга-Поста, що описує алгоритми.

Обчислювальна модель об'єктно-орієнтованого програмування заснована на поняттях об'єкту та класу і має тільки одну операцію – посилку об'єкту повідомлення. Повідомлення є також об'єктом і може мати параметри, що також є об'єктами.

Обчислювальну модель логічного програмування засновано на описі задачі сукупністю тверджень записаних формальною логічною мовою та одержання розв'язку задачі логічним виводом у деякій формальній системі.

1.2 Парадигма функціонального програмування

Обчислювальна модель функціонального програмування заснована на лямбда-численні Черча. Модель функціонального програмування розглядає кожну програму як вираз, а виконання програми як обчислення значення цього виразу.

У мовах програмування, що засновані на лямбда - численні Черча основною програмною конструкцією є функція. Форма запису визначення функції, виклику функції и обчислення значення функції взято з Лямбда – числення. Лямбда – числення є розділом математичної логіки.

Основою для запису визначення функції і виклику функції є Лямбда – вираз. Лямбда – вираз – це тіло функції с формальними параметрами, що записане без імені:

(LAMBDA (формальні параметри) тіло функції)

Форма виклику функції (Лямбда – виклик) визначається лямбда – механізмом. Форма виклику вказується в префіксній нотації:

(LAMBDA - вираз аргумент1 ... аргументN).

Підставляючи Лямбда – вираз у Лямбда – виклик одержимо загальний вираз Лямбда – виклику:

((LAMBDA (формальні параметри) тіло функції) аргумент1 ... аргументN)

Лямбда – вираз точно відображує параметризовані обчислення і дозволяє *вільно робити вкладені виклики функцій, об'єднувати Лямбда – вирази з іншими формами і Лямбда – виразами.*

Лямбда – виклик дозволяє функціям динамічно викликати одна одну, рекурсивно викликати себе безпосередньо або через інші функції. Виклики функцій можуть вкладатися, формуючи складну функцію. Лямбда – вираз складної функції дозволяє об'єднання Лямбда – викликів. Визначення функції може бути без імені.

Дана обчислювальна модель звичайно застосовується для задач, які важко сформулювати в термінах послідовних операцій.

Конкуренція на ринку збиту примусила розроблювачів мов функціонального програмування ще удосконалювати їх. Сьогодні мови функціонального програмування – мови промислового призначення.

Використовуючи обчислювальну модель функціонального програмування, можна реалізовувати звичайне операторне (імперативне) програмування. Сучасні діалекти мови LISP всі мають функції, що дозволяють реалізовувати імперативний підхід.

Обчислювальна модель функціонального програмування також дозволяє реалізовувати об'єктно-орієнтований метод програмування, ситуаційне програмування, продукційне, логічне програмування тощо.

ТЕМА 2: ПРИНЦИПИ ПОБУДОВИ ТА ФУНКЦІОНУВАННЯ МОВИ LISP

2.1 Виникнення мови Lisp

Основні ідеї, на яких базувалась мова програмування LISP, були розвинені Джоном Маккарті у 1956 році.

Треба було розробити мову для створення експертної системи. Експертна система – це вид інтелектуальних систем. Система повинна була мати базу знань, вміти самонавчатися (одержувати самостійно знання), саморозвиватися (формувати алгоритми під час виконання програмування) та розуміти обмежену природну мову. Тому мова повинна була значно відрізнятися від існуючих мов.

У 1962 р. Джон Маккарті застосував свої ідеї при розробці мови програмування теоретичний LISP. Мова LISP засновувалась на одній програмній конструкції - функції. В основу мови було покладено апарат математичної логіки лямбда – числення Черча. Це дозволило утворити мобільну просту та ясну для розуміння мову програмування.

Програма мовою LISP являла собою визначення однієї складеної функції, побудованої методом суперпозиції з викликів інших функцій або виклику себе. Користувач утворював своє визначення функції, використовуючи виклики стандартних функцій та виклики інших функцій користувача.

Розглянемо суть методу суперпозиції. Метод суперпозиції прийшов в програмування з математики.

Складна функція містить виклики інших функцій. Обчислення починається з обчислення внутрішніх функцій. Порядок обчислення внутрішніх функцій зліва направо. Результати обчислення внутрішніх функцій повертаються на місця їх виклику, як аргументи для зовнішнього виклику функції. Таким чином, будується композиція функцій – одна складна функція. Рівень вкладеності функцій може бути довільним. Результат виклику останньої функції, являв собою результат роботи всієї програми.

Всі визначення функцій користувача зберігались окремо одне від одного в середовищі LISP - виділеної та структурованій певним чином оперативній пам'яті. Це дозволяло динамічно викликати означення функцій з будь-якої функції користувача, а також

рекурсивно викликати себе. Основним типом даних мови було обрано список. Тому мову було названо LISP (List Procession - обробка списків). Для роботи із списками в мові Lisp були розроблені класи функції різного призначення. За роки свого існування мова LISP породила множину діалектів. Діалекти мови LISP засновуються на спільних принципах.

2.2 Середовище мови Lisp

На початку роботи системи програмування мови LISP у оперативній пам'яті виділяється пам'ять під середовище в яке завантажуються функції і дані.

Середовищем називають усі активні на даний момент константи, змінні, визначення функцій стандартних або користувача. Всі компоненти середовища пов'язані в єдину мережу посилками. Якщо на якийсь компонент відсутня посилка, то такий компонент вилучається з середовища програмою «Прибиральник сміття».

На середовищі працює інтерпретатор Common Lisp. На вхід інтерпретатору подаються вирази. Інтерпретатор працює в циклі REPL.

Read - стандартна функція вводу виразу.

Eval – стандартна функція обчислення виразу.

Print – стандартна функція виводу результату обчислення.

Loop – повторити в циклі Read – Eval – Print.

Від типу виразу залежить його обчислення. Дивися таблицю 2.1

Таблиця 2.1 - Типи виразу і відповідні результати обчислення

№	Вираз	Результат обчислення
1	Константа	Константа
2	Змінна	Значення змінної або повідомлення про помилку, якщо змінна не має посилки на значення
3	Виклик функції	Обчислення функції за її аргументами та повернення результату обчислення
4	Визначення функції	Компіляція функції в псевдокод.

2.3 Декларативність мови LISP

На декларативних мовах програмування не подають алгоритм розв'язку задачі, а декларують укрупнені дії - функції, які треба виконати, щоб розв'язати задачу. Кожна функція розв'язує певну підзадачу головної задачі. А головна функція декларує зв'язок між своїми функціями.

Для декларативних мов характерний «декларативний» підхід і до організації даних. Це означає, що поняття даного, ніяк не пов'язано з його подаванням в пам'яті комп'ютера, а визначається через операції, що виконуються над ним (абстрактні типи даних). В декларативних мовах найбільш розповсюджено наступні абстрактні типи даних: дерева, списки, вектори, множини. Наприклад, тип множина визначається через операції перетину, об'єднання, доповнення множин.

2.4 LISP – мова функціонального програмування

В мові LISP базовою конструкцією програми є функція. Виклик функції подається в формі списку. Назва функції завжди перший аргумент списку (префіксна форма запису). Функція обчислює свої вхідні аргументи зліва направо, далі обчислює результат і повертає його на місце свого виклику.

Приклад 1: Обчислюється сума 2-х чисел. Число 15 вводиться з клавіатури.

```
CL-USER> (+ 23 (Read))
```

```
15
```

```
38
```

Функція може не мати аргументів або мати довільне число аргументів. Функції з прикладів 1 і 2 можуть мати довільну кількість аргументів.

Приклад 2: Обчислюється добуток 4-х чисел.

```
CL-USER> (* 10 2 3 4)
```

```
240
```

Функція «+» може не мати аргументів. В цьому випадку вона повертає свій ідентифікатор операції 0.

```
CL-USER> (+)
0
```

По замовчанню аргументи функції обчислюються, якщо нема заперечення на їх обчислення. У випадку, коли аргументи функції змінні, обчислюються спочатку значення змінних. Якщо аргумент константа, то він не обчислюється.

Приклад 3: Обчислити добуток чисел, де одно з чисел визначається через змінну.

```
CL-USER> (SETQ R 5)
5
CL-USER> (* R 5)
25
```

Змінні подаються символами. Функція SETQ привласнює символу 'R посилку на число 5 (кажуть привласнює змінній R число 5). Перед множенням обчислюється значення змінної R. Результат множення 25.

Аргументами функції можуть також бути виклики інших функцій.

```
Приклад 4:
CL-USER> (* R (/ 10 5))
10
```

У прикладі 4 змінна R зберігає своє значення з прикладу 3 у середовищі. Другий аргумент – функція обчислюється. Результат роботи функції « / » повертається на місце її виклику. Після чого виконується операція множення. Результат роботи повертається, але не зберігається в середовищі.

```
Приклад 5:
CL-USER> (SETQ L (+ 4 R))
9
```

У прикладі 5 функція «+» додає 4 і 5, а результат 9 привласнюється змінній L. Результат привласнювання зберігається в середовищі. Застосовано принцип суперпозиції.

Можна заперечити обчислювання значень змінних або функцій. Апостроф перед змінною означає «функцію», що заперечує обчислення. Іншою формою запису функції є QUOTE. Знак «Q» наприкінці функції SETQ означає, що вона заперечує обчислення першого аргументу функції.

Приклад 6:

```
CL-USER> (SETQ M 6)
```

```
6
```

```
CL-USER> M
```

```
6
```

```
CL-USER> (SETQ M 'R)
```

```
R
```

Запис означає, що символу (QUOTE M) привласнюється символ 'R.

```
CL-USER> M
```

```
R
```

Запис M без заперечення обчислення означає, що необхідно виявити на яке дане посилається символ M. Тобто M розглядається як змінна.

Можна заперечити обчислення функції.

Приклад 7:

```
CL-USER> (SETQ N '(+ 3 5))
```

```
(+ 3 5)
```

```
CL-USER> N
```

```
(+ 3 5)
```

У прикладі 7 заперечене обчислення функції «+». Значенням змінної N є список, у якого 3 елемента: «+», «3», «5».

У загальному вигляді виклик функції подається:

(Ім'я_функції Фактичний_параметр1, ... ,Фактичний_параметрN),

де фактичний параметр може бути змінною, константою або викликом іншої функції.

У загальному вигляді означення функцій подається:

(DEFUN Ім'я_функції (Формальні параметри) Тіло_функції)

Приклад 8:

(DEFUN SUM (X Y) (+ (* X X) (* Y Y))

де DEFUN – ім'я функції, що компілює означення функції користувача у псевдокод. При виконанні функції псевдокод інтерпретується інтерпретатором. Формальні параметри задаються в формі списку; «тіло функції» містить дії - функції.

2.5 LISP - мова обробки списків

Базовим типом мови LISP є список. Список записується в круглих дужках. Елемент від елемента відділяється пропуском. В список можуть вкладатися інші списки. Елементами списків можуть бути дані різних типів. Цей факт важливий для створення БД інтелектуальних систем.

Приклад 1: Скласти список, що містить підсписки, в яких є аббревіатури дисциплін і години лекцій.

CL-USER> (SETQ D '((ОП 32 32) (ЛП 32) (ФП 34))

Дані з прикладу записані вкладеними списками. Список дозволяє реалізовувати складні дані різної структури та різної довжини.

Як ми бачили у попередніх прикладах, виклики функцій та визначення функцій користувача також записують у формі списку. Формальні параметри визначення функції вкладені списки.

Приклад 2: У прикладі створюється список, елементами якого є рядки.

CL-USER> (SETQ N '(("понеділок" "вівторок" "середа" "четвер"
"п'ятниця"))
("понеділок" "вівторок" "середа" "четвер" "п'ятниця"))

Складні списки можна створювати розміщуючи в списку замість списків змінні, що посилаються на свої списки.

Приклад 4: Нехай змінним M і N привласнено списки. Треба побудувати список із цих списків.

```
CL-USER> (SETQ M '(3 4 6) N '(7 9 1))
(7 9 1)
```

```
CL-USER> (SETQ P (LIST M N))
((3 4 5) (7 9 1))
```

Функція LIST утворює список із будь-якої кількості своїх аргументів. Попередньо обчислюються значення змінних M і N. В результаті буде одержаний список P з двох елементів – списків.

Застосування змінних дозволяє гнучко структурувати списки.

У мові LISP список є універсальною формою запису даних і програм.

2.6 Уніфікований підхід до даних і програм

Форми представлення програм і даних в мові LISP однакові – списки. Це дозволяє програмі користувача розглядати визначення функції як дане-список; створювати, оброблювати та перетворювати визначення інших функції або навіть своїх функцій. Крім того, можливість застосовувати функції інтерпретатору у програмі користувача, дозволяє програмі компілювати їх і виконувати.

Ці властивості мови LISP застосовується для реалізації систем, що самонавчаються і саморозвиваються.

Приклад 1: Означення функції можна створити як звичайний список символів. Після чого виконати програмно компіляцію функції та її виконання. У прикладі розглядається процес створення працюючої функції програмно. Функцією STV створюється і виконується функція SUM.

```
CL-USER> (DEFUN STV ()
  (SETQ D '(DEFUN SUM (X Y) (+ X Y) )) )
STV
```

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

CL-USER> (EVAL D)
(SUM)

CL-USER> (SUM 2 3)
5
```

Програма STV створює список D. Спочатку обчислюється аргумент функції EVAL. Потім функція EVAL ініціює (обчислює) виконання функції DEFUN, а та компілює функцію SUM.

Дані, які записані в формі списку, можна застосовувати як виклик функції, при умові, що перший елемент функції є іменем функції. Тоді інші елементи списку будуть застосовуватись, як фактичні параметри функції.

Приклад 2:

```
CL-USER> (SETQ D '(DEN 23.12.06) )
(DEN 23.12.06) )

CL-USER> (EVAL D)
«Так»
```

Функція EVAL викликає функцію «DEN» для перевірки, чи є вказане число вихідним днем.

```
(DEFUN DEN (число) (перевірка вказаного числа на ознаку
«вихідний»))
```

В середовищі однаково зберігаються, як списки даних, так і визначення функцій. Середовище зберігає всі списки разом незалежно від їх призначення.

Зберігання даних і програм у єдиній пам'яті відкриває великі можливості для програмування.

Розміщення даних і програм у єдиній пам'яті було запропоновано фон Нейманом для реалізації *універсальної обчислювальної машини ще в 40 роки.*

У ті ж роки були висловлювана аналогічна ідея про створення універсальної обчислювальної машини Лебедєвим Сергієм

Олексійовичем у Києві. Під його керівництвом була створена перша у СРСР лабораторія розробки обчислювальних машин.

2.7 Відкритість мови

Мова програмування, що має спеціальні засоби, які дозволяють модифікувати або розвивати мову, називається відкритою.

LISP відноситься до відкритих мов за наступними причинами:

- службові програми мови LISP (текстовий редактор, відлагоджувач) створено мовою LISP. Це дозволяє користувачу змінювати їх для себе;
- застосовуючи макрозасоби можна переносити функції з одного діалекту мови LISP, в інший діалект, розширюючи мову;
- за значеннями керуючих змінних у середовищі LISP можна змінювати режим роботи певних стандартних функцій;
- функції інтерпретатору доступні для роботи користувача, що дозволяє йому застосовувати функції нижнього рівня.
- LISP – мова відкрита для створення мов вищого рівня. Вона дозволяє утворювати функції для реалізації об'єктно-орієнтованого методу програмування, ситуаційного, продукційного, логічного, програмування, що керується даними.

Приклад1. Розглянемо можливість вводити числа функцією інтерпретатору READ у іншій системі числення. Для цього застосовується стандартна змінна *Read-Base* (інший режим роботи функції READ).

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

```
2
```

```
CL-USER> (+ 11 (READ))
```

```
110
```

```
9
```

При вводі виразу (+ 11 (READ)) не можна вводити ніякі цифри крім 0 і 1. Інші цифри будуть сприйматися як символи. Знаходиться

сума числа 3 (у двійковій формі 11) і числа 6, що вводиться (у двійковій формі 110). Результат 9 виводиться у десятковій формі.

2.8 Динамічна типізація змінних

В мові LISP немає об'яв типів для змінних. LISP використовує динамічну типізацію змінних. Одній змінній можна привласнювати дані різних типів. У мові LISP типи прив'язуються до даних.

Вказана властивість дозволяє будувати нові абстрактні типи даних для мов більш високого рівня. Застосовуючи динамічну типізацію можна писати універсальні функції.

Приклад 1: Написати функцію, що вводить дане і визначає його тип. Функція повертає тип даного і кількість елементів, а для чисел діапазон типу.

```
CL-USER> (DEFUN UNIVERSAL () (SETQ M (READ)) (TYPE-OF M))
```

```
UNIVERSAL
```

```
CL-USER> (UNIVERSAL)
```

```
12
```

```
(INTEGER 0 16777215)
```

```
CL-USER> (UNIVERSAL)
```

```
"asd"
```

```
(SIMPLE-BASE-STRING 3)
```

Змінна M не має типу, їй можна привласнювати посилку на дане будь-якого типу.

Для сталої роботи програми перед виконанням дії можна контролювати тип даного функціями предикатами. Якщо виконуються арифметичні функції, то нечислові аргументи викличуть переривання за помилкою.

Приклад 2. Введемо контроль даного, що вводиться перед виконанням функції.

```
CL-USER> (DEFUN S () (SETQ K (READ))
```

```
(IF (NUMBERP K) (+ 7 K) (PRINT "Не числовий аргумент"))) )
```

S

CL-USER> (S)

12

19

CL-USER> (S)

"Q"

"Не числовий аргумент"

"Не числовий аргумент"

2.9 LISP - мова обробки символічної інформації

Для обробки символічної інформації в мові LISP існують функції роботи із символами, рядкові функції та функції роботи із знаками.

Вказаними функціями можна *під час роботи програми* формувати ідентифікатори програми (імена змінних, функцій, констант), змінювати їх.

Мова Lisp дозволяє при роботі з природними мовами будь-яке слово, спец знак, цифру, що введені зовні, застосовувати як ім'я змінної або функції. Це дозволяє легко поповнювати бази знань, в яких поняття подаються через символи та їх властивості.

Застосовуючи функції атрибутів і властивостей символів, можна для кожного слова визначати його граматичні і семантичні характеристики. Структуру речення можна подавати списками.

Інтелектуальні системи працюють з поняттями бази знань, що подаються твердженнями природною мовою. Роботу програми з символічними даними називають *символічними обчисленнями*.

2.10 LISP - рекурсивна мова

Визначення функції користувача можна подавати двома способами: суворо функціональним та імперативним.

При суворо функціональному способі означення функції має тільки одну складну функцію.

При реалізації визначення функції суворо функціональним способом, операції що повторюються, реалізуються рекурсією. Рекурсія і суворо функціональний спосіб написання функцій

дозволяють писати компактні, красиві функції. Рекурсією зручно оброблювати рекурсивні дані: списки, рядки, масиви.

При імперативному способі LISP імітує послідовне виконання функцій. Такий спосіб програмування більш звичний, а цикли організуються стандартними макросами.

2.11 Принцип модульності

Відомо, що складну проблему легше та швидше вирішити, якщо розділити проблему на кілька простіших проблем. Принцип модульності, сформульований Г. Майерсом, забезпечує можливість утворювати програми будь-якої складності.

Принцип модульності реалізовано у мовах всього Lisp-сімейства за рахунок того, що кожна програма складається з окремих функцій. Функції пов'язані у єдину програму викликами. По суті програма на мові LISP – це ієрархічна структура з викликів функцій.

У чітку структуру програми легко додати нову функцію, ідея розроблювача прозора відображується у кожній її функції.. За рахунок модульності збільшується швидкість написання програми, Все це дозволяє легко модифікувати програму. При налагоджуванні програми можна кожен функцію налагоджувати окремо. Модульність мови дозволяє розв'язувати складні задачі з максимальною надійністю і ефективністю.

2.12 Сучасний стан розробок діалектів мови Lisp та їх вплив на проект COMMON LISP

Одна з найважливіших подій в житті мови Lisp відбулась у другій половині 1970 року: була розроблена мова програмування *Scheme* у Массачусетському технологічному інституті (MIE). Головним достоїнством мови *Scheme* була лексична простота (мале ядро) та замкненість, першокласне розширення та спрощений синтаксис. Деякі з цих достоїнств мали великий вплив на проект Common Lisp.

Пізніше, концепції програмування, орієнтовані на об'єкт, стали мати сильний вплив на мову Lisp. Це стало причиною появи об'єктно-орієнтованої окремої мови Common Lisp Object System (CLOS).

З 1980 компанії Symbolics та LMI стали розвивати мову Lisp для Lisp-машин; групи реалізації апаратних засобів EOM розвивали NIL, Franz Lisp, і PSL; Xerox розвивав InterLisp; а Spice проєкт розвивав MacLisp-подібний діалект Lisp, що називався SpiceLisp.

У квітні 1981, після зустрічі відносно розколотого LISP - товариства Symbolics, SPICE -проєкт, NIL проєкт, и S-1 LISP проєкт, вони об'єднуються разом, щоб визначити стандарт Common Lisp.

Стандарт Common Lisp був розроблений, як опис сімейства мов. Первинний вплив на Common Lisp мали мови: Lisp компанії Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, Scheme.

Наприкінці 80-років існувало багато лабораторій та компаній із своїми реалізаціями мови LISP. Щоб уникнути розбіжностей у мовах, була сформована технічна робоча група, щоб розробити проєкт для ANSI Common Lisp Standard.

Цілі включали: сувору стандартизацію для мобільності мови, орієнтованої на об'єктні системи програмування, системи станів, ітеративні засоби обслуговування, способи спілкування з великими наборами символів.

У 1986 був випущений ANSI Common Lisp стандарт. Починаючи з цього моменту розвивають тільки Common Lisp стандарт. Причому пропозиції подаються у ANSI, який ретельно розглядає їх. Сьогодні реалізації мови Common Lisp активно розвиваються, як більшість мов з відкритим кодом. Мову Practical Common Lisp було створено у 2006 році.

Серед найбільш відомих діалектів мови Lisp треба відмітити ELisp, AutoLisp, Scheme, PSL, Franz Lisp, MacLisp, InterLisp, Llist.

На ELisp реалізовано текстовий редактор Emacs дуже популярний серед користувачів. Редактор відкритий для модифікування і дозволяє компонувати з ним програми.

Мова AutoLisp розширена для програми AutoDesk AutoCAD, яка дуже часто застосовується для розв'язку інженерних задач.

Мова Scheme часто використовується у ВНЗ для знайомства з функціональною парадигмою програмування на початкових курсах. Мова легка при вивченні її специфікації; також використовують для досліджень у області мов програмування. Сьогодні мову Scheme часто застосовують для промислового програмування. Мова Common Lisp стала мовою промислового застосування.

ТЕМА 3: СТАНДАРТ COMMON LISP

Простота і зрозумілість стандарту Common Lisp, як для програмування, так і для реалізації мови, зробила його популярним.

Стандарт описує спільне ядро реалізацій. Існують реалізації за стандартом Common Lisp на машинах різних типів, що робить переносним програмне забезпечення. Однак переносність відноситься тільки до ядра мови.

Реалізації мови Common Lisp мають відкритий код, що дозволяє легко розширювати мову, утворювати прототипи для реалізації системи за спіральною моделлю. У Інтернеті розроблено багато бібліотек для Common Lisp різного призначення також відкритих для поповнення.

Засобами Common Lisp легко реалізувати підхід, що орієнтовано на об'єкти. Для мов характерна динамічна типізація; гнучка система обробки умов виконання дій.

Швидкість реалізацій Common Lisp може бути порівняна з швидкістю роботи програми на мові C, при наявності хорошого компілятора і застосуванні спеціальних додаткових об'єктів.

Будь-якому розроблювачу дозволяється розробити свою реалізацію мови Common Lisp відповідно стандарту.

Завдяки вимогам стандарту програми користувача можна виконувати на інших реалізаціях Common Lisp. Проте, кожна реалізація мови Common Lisp розширює можливості мови, тобто виходить за стандарт.

Якщо можливості майбутньої системи вимагають розширення мови, то підбирають таку реалізацію мови, що забезпечує необхідні можливості.

У Інтернеті існує багато бібліотек з кодами Common Lisp, що розширюють стандарт у певному напрямку. Наприклад, для утворення графічного інтерфейсу.

Серед реалізацій Common Lisp є, як комерційні, так і безкоштовні. До комерційних реалізацій відносяться: Allegro Common Lisp, Harlequin, Corman, Common Lisp – Controller, PPCRE_Common Lisp, LispWorks. До безкоштовних реалізацій відносяться: GNUCL, Clisp, CmuCL.

Компанії, розроблювачі мови Common Lisp, випускають його реалізації для різних операційних систем.

Компанія Franz Lisp випустила версію Allegro Common Lisp для GNU / Linux (безкоштовну), Windows, OSX.

Реалізація CMUCL розроблена в університеті Карнегі Мелан.

Реалізація SBCL є розвитком реалізації CMUCL і дозволяє працювати з відкритими вхідними текстами системи програмування. У реалізації є можливість компілювати програму користувача у машинний код. Реалізація SBCL – розроблено для Unix систем.

Ведуча фірма – розроблювач роботів IS Robotics пропонує на ринці відкриту програмну архітектуру для керування робото технічними системами на основі технології Corba – Mobility Robot SoftWare Development Environment Package.

Програма має набір готових компонентів для конструювання керуючих систем. Графічна оболонка середовища виконана на Java, а програмування виконано на мові LLisp – удосконаленому діалекті Common Lisp.

Мова L підтримує паралельне виконання і синхронізацію роботи множини процесів, що необхідно при утворенні роботів з кількома системами керування рухом і аналізом інформації від різних периферійних датчиків.

ТЕМА 4: СИСТЕМА ПРОГРАМУВАННЯ МОВИ PRACTICAL COMMON LISP

Для установки системи програмування Practical Common Lisp використовуйте архів LispBox-0.7-ccl-1.6-Windowsx86. або LispBox-0.7-Clisp-2.37.rar.

Добутий з архіву дистрибутив легко установити за допомогою файлу LispBox.bat.

Для застосування кирилиці у першому архіві треба знайти у каталозі LispBox-0.7 файл LispBox.bat і замінити:

```
set TO-EVAL=“(progn (load “\lispbox\”)(slime))” (1)
```

на рядок:

```
set TO-EVAL=“(progn (setq slime-net-coding-system ‘utf-8-unix)
(load “\lispbox\”)(slime))”
```

Система програмування Practical Common Lisp забезпечує користувача оболонкою, яка дозволяє використовувати будь-яку реалізацію мови Common Lisp, що створена за стандартом мови. Оболонка згладжує нестиковки між реалізаціями Common Lisp.

Базовими програмами системи програмування є:

- текстовий редактор Emacs;
- інтерпретатор Practical Common Lisp;
- оболонка Slime.

Текстовий редактор Emacs написаний об’єктна – орієнтованою мовою і зберігає всю інформацію про свою роботу та роботу користувача разом з процедурами, що обслуговують цю інформацію, у об’єктах.

Об’єкти Emacs називають буферами.

Інтерпретатор, середовище Common Lisp та інші програми системи програмування також подаються об’єктами. Об’єкти текстового редактору та Common Lisp взаємодіють через оболонку Slime.

Під час роботи Common Lisp зв'язок оболонки з Common Lisp може бути розірвано. Зв'язок можна поновити за командою «ALT-X Slime».

Оболонка містить меню для роботи з інтерпретатором, текстовим редактором та іншими програмами системи програмування.

4.1 Текстовий редактор EMACS як основа системи програмування

4.1.1 Загальні властивості Emacs

За викликом командного файлу Lispbox.bat завантажується текстовий редактор Emacs. Текстовий редактор написаний мовою ELisp за винятком деяких функцій, які написано мовою C (для швидкості роботи).

Редактор має властивість розширюватись, користувач може додавати нові команди. Редактор Emacs дозволяє себе корегувати, завантажуючи свої функції у буфер під час роботи [2].

Працює Emacs в багатозадачному режимі, реальному часі. Екран часто поновлюється (після набору одного або кількох символів).

Emacs відноситься до редакторів, що настроюються: можна призначати команди до інших клавіш. Меню редактору змінюється, залежно від ситуації.

Текстовий редактор Emacs має всі властивості сучасних розвинених редакторів.

4.1.2 Фрейми

Весь екран або вікно називають фреймом. Фрейм складається з основного вікна для подавання вмісту інформації буферу та додаткових вікон. Додаткові вікна вказують режими роботи з інформацією буферу або команди, які можна виконати з цим буфером. Кожне основне вікно розширюється додатковими вікнами (extended), тому фрейм називають X - вікном.

Редактор запускається з одним фреймом, але можна утворити ще фрейм за опцією меню *File/New frame* або командою *CTRL-x символ*. Залежно від вибору символу застосовуються різні способи створення фрейму.

У текстовому редакторі існує два способи роботи користувача:

- за опціями меню;
- за командами.

Фрейм – екран містить рядок – меню та один або декілька фреймів (X - вікна). Фрейм X - вікно складається з середньої робочої частини для тексту; двох нижніх рядків: рядку режиму, області луни. Рядок режиму розташовано над областю луни.

4.1.3 Буфери та їх режими роботи

Кожному фрейму відповідає буфер редактору або буфер Common Lisp. Один буфер може мати кілька вікон. Ім'я буферу може бути будь-якої довжини.

Ім'я буфера співпадає з ім'ям файлу, вміст якого розміщено у буфері. Якщо користувач звертається до файлу(команда *Open File*), то для нього автоматично утворюється буфер та вікно.

Для виконання команд Emacs автоматично створює необхідний буфер. Наприклад, для вмісту каталогу за командою *Open Directory* або за командою *Help*. Користувач може сам створити порожній буфер з будь-яким ім'ям.

Буфер з *курсором називають обраним*. Позиція курсору в тексті буфера відмічається кольоровим прямокутником(точкою). Щоб запам'ятати позицію курсору у тексті, що раніше було обраним, використовується порожній прямокутник (точка).

Більшість команд та увід відноситься до обраного буферу. *Команди миші відносяться до буферу на якому виконано щиглик*. У довідковій буфері не можна вводити. При вході у текстовий редактор Emacs утворює буфери та встановлює їх основні режими роботи (Дивися таблицю 4.1).

Таблиця 4.1 - Основні режими роботи буферів

Назва буферу	Об'єм	Основний режим	Призначення
slime-repl clisp-2.37	34	REPL	Робота з Common Lisp
scratch	0	Lisp Interaction	Допоміжний

Messages	859	Fundamental	Повідомлення користувачу
Compile-Log	0	Fundamental	Інформація про компіляцію
slime-events	2251	Fundamental	Інформація про події у оболонці
inferior-lisp	4527	Comint	Повідомлення про роботу Elisp та C

У певний момент кожен буфер може працювати у одному *основному режимі*: Fundamental, Text, Lisp, C, Texinfo та інші.

Режим „REPL”(опція меню) визначає можливості керування роботою Common Lisp. Режим “Lisp Interaction” – режим взаємодії користувача з ELISP. Режим “Fundamental” – режим тільки для читання. Режим “Comint” коментує події на нижньому рівні.

4.1.4 Рядок режиму буфера

Кожен буфер має рядок режиму. Щоб висвітлити рядок режиму у вікні на екрані, треба розгорнути вікно.

Рядок режиму містить назву буфера. Для швидкої зміни вікна буфера треба клацнути на назві буфера у рядку режиму. Потягнувши за рядок режиму можна змінити розмір вікна. Вміст рядку показано у таблиці 4.2.

Таблиця 4.2 - Символи рядку режиму

Символи					Назва буферу	Режими		№ поточ. рядку L	Позиція
1	2	3	4	5		Основний	Другорядний		
-	5	\	*	*	*slime-repl clisp-2.37*	REPL	Abbrev	5	ALL

Встановлюючи курсор на 1 символ рядку, можна одержати інформацію про можливі у даному контексті команди:

- зробити на екрані одне вікно командою (C-x 1), незалежно від того скільки вікон було створено на екрані;
- зробити друге вікно, що розташовано горизонтально можна командою: (C-x 2), вміст вікна – дублюється обраний буфер;
- зробити друге вікно, що розташовано вертикально: (C-x 3), вміст вікна обраний буфер.

Встановлюючи курсор на 2 символ рядку, можна одержати інформацію про набір кодування системи, а натиснувши праву клавішу миші можна одержати буфер з повним описом кодування.

Якщо 2 символ містить:

- «-», то кодування по замовчання;
- «1», то кодування ISO Latin 1;
- «5», то кодування Cyrillic 5.

Символ 3 вказує на метод вводу. Символ 3 заміщується на « \ », якщо ознака кінця рядку «BK» та «ПС».

Символи 4 і 5 заміщуються , якщо буфер вікна:

- редагувався «*»;
- не редагувався «--».

Встановлюючи курсор на символ 4, можна одержати режим роботи з буфером:

- «тільки читати», за правою кнопкою миші можна змінити вказаний режим. Знак % у 4 символі означає, що режим “тільки читати”;
- «не тільки читати», за правою кнопкою миші можна змінити вказаний режим. Знак * означає, що режим “не тільки читати”.

Встановлюючи курсор на 5 символ можна одержати інформацію про команду модифікування режиму. Наприклад, натиснувши праву кнопку миші можна замінити режим «читати-писати» на режим «тільки читати».

У дужках рядка показано назву основного і другорядних режимів роботи з буфером. Наприклад, при роботі з Common Lisp використовується основний режим REPL, тобто можна застосовувати команди опції REPL.

Другорядний режим це – режим, який включено для обраного буфера і тільки для обраного вікна. Обрати другорядний режим можна встановивши покажчик миші на основний режим у рядку режиму та натиснувши праву кнопку миші. Прикладом другорядних режимів можуть бути режими Overwrite та Abbrev. Перший дозволяє перезаписувати файл, а другий працювати з командами редагування тексту текстового редактору Word.

У рядку біля літери «L» вказано номер рядка у якому знаходиться курсор. Позиція All вказує, що у вікні показано всі рядки буфера. Символ TOP вказує, що курсор знаходиться у першому рядку. Символ BOT показує, що курсор знаходиться у останньому рядку, або «число %» показує скільки % тексту знаходиться зверху вікна.

4.1.5 Область луни

Область луни знаходиться під рядком режиму. Вона відлунює події, які є наслідком дій користувача; або надає підказки про помилки і наступні дії. Інформаційні повідомлення з області луни накопичуються у буфері *Messages*.

Область луни використовується також для Міні буфера – вікна для вказівки аргументів команд. Наприклад, користувач хоче утворити порожній буфер з будь-яким ім'ям за командою *CTRL-x b*, а ім'я задати у міні буфері. Вийти з буферу можна за командою C-g.

Для вводу у міні буфер символу (імені) можна користуватись командою авто доповнення: клавіша TAB. Ввести один або декілька знаків і натиснути клавішу TAB. У вікно HELP виводиться список символів. Обраний символ заноситься у міні буфер.

4.1.6 Меню текстового редактору

Оболонка Slime системи програмування Common Lisp має головне меню, команди якого призначено для роботи з Common LISP (підменю REPL), текстовим редактором Emacs (підменю FILE, EDIT, OPTIONS, BUFFER, TOOLS, HELP), оболонкою Slime (підменю Slime).

Текстовий редактор забезпечує:

- роботу з файлами, каталогами, буферами (за командами підменю FILE);

- редагування текстів (за опціями підменю EDIT);
- встановлення додаткових можливостей для роботи з текстовим редактором: вибір мов спілкування, вибір набору символів, шрифту, кодування; застосування структурного редактору, синтаксичного контролю, тощо (за опціями підменю OPTIONS);
- інструментальними засобами для пошуку файлів; налагоджування функцій редактору EMACS; порівняння, злиття файлів; спілкування у Інтернеті, тощо (за опціями підменю TOOLS);
- перегляд та активізування буферів (за опціями підменю BUFFER);
- довідку про роботу з текстовим редактором (за опціями підменю HELP).

4.1.7 Команди текстового редактору

Виконати дію можна набравши команду на клавіатурі. Команду треба набирати швидко, тому вона повинна мати мало символів. Але таблиця символів *ASCII* має невелику кількість символів, а кількість команд Emacs дуже велика.

Використання комбінацій клавіш клавіатури дозволяє розширити таблицю символів ASCII для набору команд. Текстовий редактор EMACS використовує розширену таблицю ASCII.

Розглянемо команду Emacs „CTRL-h k”. Вона дозволяє одержати довідку про обрану команду Emacs. Символи „CTRL-h” набирають послідовно. Але клавішу „CTRL” не відпускають поки не буде натиснута клавіша „h”, на це вказує риска „-”. Символ „k” набирають після „CTRL-h”.

Комбінації клавіш застосовують наступні символи:

- модифікатори (керуючі символи з таблиці ASCII). У нашому прикладі - „CTRL”;
- додаткові керуючі символи (символи, що друкуються з таблиці ASCII. Emacs використовує їх, як керуючі). У нашому прикладі - „h”;
- символи, що друкуються. У нашому прикладі - „k”.

Кожна клавіша клавіатури має свій номер - SCAN код. При вводі друкованого символу з клавіатури він відображується двома байтами у буфері клавіатури. Молодший байт містить SCAN код клавіші, яка відповідає символу. Старший байт містить 0, що означає клавіші Shift/Ctrl/Alt/Win/Capslock не натиснуті.

Для комбінації клавіш старший байт описує, яка натиснута керуюча клавіша. Опис старшого байту по бітам подано нижче:

7 6 5 4 3 2 1 0 Усі біти = 0, а саме

0 0 0 0 0 0 0 0 вводиться друкований символ

| | | ^---- Комбінація Shift + Caps Lock = 1

| | ^----- Натиснута клавіша Ctrl

| ^----- Натиснута клавіша Alt

^----- Натиснута клавіша Win.

Комбінація керуючого символу разом з друкованим дає розширений код, дивися Таблицю 4.3.

Таблиця 4.3 Розширені коди для комбінацій клавіш у 16-річному коді

SCAN код	Код вводу для клавіш
0041	"A"
0141	Shift-A
0241	Ctrl-A
0441	Alt-A
0841	Win-A
0341	Ctrl-Shift-A
0741	Ctrl-Alt-Shift-A

Ми розглянули перший спосіб розширення таблиці *ASCII*. Розширити таблицю символів *ASCII* можна також за рахунок М - біту (останнього біту старшого байту). Такий біт прийнято називати мета-біт. Кожен символ *ASCII* таблиці може мати свій аналог мета.

Для набору команди М-а використовують мета клавішу. При відсутності мета клавіші можна набрати клавішу ESC. Наприклад, замість клавіш М-а можна набрати ESC-а.

Кожне звертання користувача до операційної системи через натискання клавіші *ASCII*-термінала, клавіші миші – подія. Подією є також натискання функціональних клавіш F1,F2,...,F12 або стрілки. Події можна модифікувати клавішами модифікаторами.

У командах, що вводяться з клавіатури, застосовують наступні скорочення: C - CTRL; M - ALT або ESC.

Нижче подається декілька корисних команд. Група довідкових команд визначається префіксною командою C-h(help) або F1.

- 1) Вивести Emacs документацію: C-h t.
- 2) Вивести зміст Emacs документації: C-h i.
- 3) Вивести повну довідку за командою: C-h ?. У довідці вказано опції відбору. Вказавши у міні буфері опцію відбору команд можна одержати довідку про групу команд; одержати відомості про ключі; кодування системи; функцій змінних Common Lisp, тощо.
- 4) Знайти команду в меню: C-h w.
- 5) Знайти за ключем опис команди: C-h k. Вводиться команда та ім'я ключа в міні буфер. Команда дає повну інформацію про ключ. Наприклад, ключ C-f.

4.1.8 Ключі та команди EMACS

Послідовність набраних символів поєднаних рисою означає одну подію. Символ, який набрано окремо, означає другу подію.

Таким чином команда „C-h w” – означає дві події: звернутись до довідки і знайти, де у меню розташована команда, ім'я якої вказується у міні буфері. Послідовність подій називають ключем. Ключ може бути закінченим – послідовність подій достатня для опису команди. Увід закінченого ключа - запускає команду.

Ключ може бути префіксним. Префіксний ключ вказує на групу команд. Продовження префіксного ключа може зробити ключ закінченим. Обмеження на довжину ключа немає.

Висновок: Команда може складатися з будь-якого числа подій динамічно. Префіксні ключі визначають групу команд.

У EMACS немає списку префіксних ключів. Користувач може утворювати нові префіксні ключі, використовуючи стандартну прив'язку або вилучати старі ключі. Вилучення префіксних ключів впливає на команди групи.

Текстовий редактор має таблиці ключів. Кожна компонента таблиці встановлює зв'язок між ключем(командою користувача) і внутрішньою командою(функцією на мові EMACS LISP). Функція одержує від користувача необхідні дані та передає їх програмі для виконання дій. Програма, що виконує дії також написана на мові LISP.

4.1.9 Міні буфер

Міні буфер використовують, щоб ввести додаткову інформацію для будь-яких команд. Додаткова інформація буде використовуватись функціями Lisp, як фактичні параметри.

Якщо команда вимагає додаткової інформації, то автоматично в області луни з'являється міні буфер і він стає поточним буфером. Міні буфер завжди підказує, що повинен ввести користувач. Підказка закінчується символом „:” та може містити замовчення для відповіді. Наприклад, команда меню FILE/OPEN DIRECTORY підказує каталог по замовчанню.

Користувач вводить інформацію та закінчує увід „ENTER”(у документації „RET” від слова RETURN). По замовченню вводять тільки „ENTER”. Після вводу рядку „ENTER” не натискають, для цього застосовують команду C-j.

При вводі можна застосовувати „Завершення”. Завершення застосовують тоді, коли забули команду або аргумент команди. Завершення означає, що можна ввести перші символи команди і завершити їх пропуском(“ “) або „ENTER”. Редактор EMACS запропонує можливі продовження.

Якщо є один варіант, то натиснувши клавішу TAB, команда поповниться автоматично. Наприклад, при вводі команди M-x insert-b, вона автоматично поповниться M-x insert-buffer.

Якщо варіантів багато, оберіть мишкою один варіант та натисніть „ENTER”, команда поповниться та виконається.

При роботі з міні буфером можуть виникнути ситуації, коли міні буфер потрібен для інших цілей. Тоді, міні буфер зберігається, а після вводу користувача повертається.

Усі команди введені у міні буфер зберігаються в списку – історія. Список можна переглянути, рухаючись за стрілками: вгору, вниз. Команди можна застосовувати повторно.

Для скасування команди, яку ви викликаєте, виконайте команду C-g. Після виконання команди міні буфер стає не активним і виконується перехід на верхній рівень.

4.1.10 Команди роботи з текстом

Ми будемо розглядати мінімальний набір команд, що дає можливість працювати з текстовим редактором. Насправді, текстовий редактор EMACS має набагато більше можливостей значно відмінних від можливостей інших редакторів.

ВСТАВКА СИМВОЛІВ

При вводі рядку тексту, довжина якого більше довжини рядка на екрані, зручно включити додатковий режим AUTOFILL. В режимі AUTOFILL довгий рядок автоматично розбивається на два рядка. Додаткові режими включаються за правою кнопкою миші, якщо встановити покажчик миші на основний режим у рядку режиму.

Для перезапису символів треба включити додатковий режим Overwrite. Для вставки у текст керуючого не графічного символу використовують перед його вставкою команду C-q. Командою C-q можна вставити символ, набравши його восьмирічний код. Наприклад, послідовність C-q 60 <Enter>вставляє символ 0. Замість <Enter> може використовуватись будь-який не цифровий символ

ВИЛУЧАННЯ ТЕКСТУ

Вилучити текст можна за командою – C-k. Якщо курсор стоїть на кінці рядку, то команда об'єднує рядки без вилучання.

ВІДМІНА ЗМІН

Всі буфери за винятком тих, імена яких починаються з пропуску, зберігають зміни і їх можна повернути. Команди відміни

раніше зміненого тексту EDIT/UNDO або C-x u. При змінах точка повертається до попереднього положення.

Увід іншої команди під час відміни змін може повернути відмінені вами зміни. Наприклад, команда C-f.

Можливо відмінити зміни не у всьому буфері, а тільки у області. Для цього треба встановити курсор лівою кнопкою миші на початок обраної вами області та не відпускаючи кнопку обрати область. Після чого виконати команди відміни.

При дуже великій кількості змін, найстаріші зміни, що зберігаються редактором EMACS вилучаються. Останні зміни ніколи не вилучаються.

Механізмом вилучання у EMACS є механізм збирання сміття. Можна задати кількість байтів, яку відводять для зберігання змін у змінній Undo-Limit. По замовчанню значення змінної 20000 байт.

ФАЙЛИ

Текст буферу зберігається у файлі за опцією меню FILE/ SAVE або FILE/ SAVE AS або командою C-x C-s з вже обраним ім'ям.

Завантажити файл в буфер текстового редактора можна за опцією меню FILE/OPEN FILE або за командою C-x C-f.

Для цього у міні буфер(область луни) виводиться ім'я поточного каталогу та пропонується ввести ім'я файлу. У разі необхідності каталог можна змінити. Якщо вводиться ім'я файлу і у каталозі файл відсутній, то пропонується утворити новий файл та зберегти його за командою FILE/ SAVE AS.

Наявність файлу викликає завантаження його вмісту в буфер для редагування або друку.

ВІДСІК РЯДКІВ

У рядках вікна можна не відображувати всю інформацію. У буфері ця інформація залишається. Опція меню Options/Truncate Long Lines in this Buffer дозволяє переглянути рядок за маркером стрілка. На екран виводиться маркери стрілки. Подивитись інформацію можна за горизонтальною прокруткою. Відключити відсік рядків можна надаючи локальній змінній TRUNCATE-LINES значення NIL.

ЧИСЛОВІ АРГУМЕНТИ

Будь-якій команді EMACS можна передати числовий аргумент. Аргумент називають префіксним, бо він вводиться перед викликом команди. Набрати числовий аргумент можна за командою: ALT-`<число>` або командою C-`и` `<число>`.

Якщо число від'ємне, то дія команди у зворотному напрямку.

Різні команди використовують числовий аргумент по-різному. Команди можуть використовувати числовий аргумент для повторювання дії. Наприклад, команда ALT-10 C-k вилучить від курсору 10 рядків.

Деякі команди перевіряють тільки наявність аргументу, а не значення. Наприклад, команда ALT-q без числового аргументу заповнює рядок неповними рядками, а з числовим аргументом вирівнює рядок за правою границею.

Числовий аргумент можна використовувати для багатократної вставки символу. Команда C-`и` `<символ>` без числового аргументу вставляє в текст 4 символи. Команда C-`и` C-`и` `<символ>` вставляє 16 символів. При застосування числового аргументу у команді вставляється вказана кількість символів. Наприклад, команда „C-`и`20`р`” вставляє 20 символів „`р`”.

ПОВТОРНЕ ВИКОНАННЯ КОМАНДИ

Для повтору тільки що виконаної будь-якої команди застосовують команду C-x z. Команда, що була виконана, застосовує ті ж самі аргументи. Для повторювання команди C-x z треба ввести символ z стільки раз, скільки треба повторити команду.

Наприклад, нехай введено символ „`к`”. Команда C-x z z добавить ще 2 символу „`к`”.

ЗАПУСК КОМАНДИ ЗА ІМЕНЕМ

Для запуску команди за іменем треба: набрати M-x; використати міні буфер для вводу імені команди і запуску. При вводі команди можна використовувати завершення.

ПОШУК, ЩО НАРОЩУЄТЬСЯ

Редактор EMACS дозволяє виконувати звичайний пошук, який не нарощується, та пошук, який нарощується. Під текстом ми будемо розуміти: сполучення символів, слово, словосполучення, фрагмент тексту.

Пошук, який нарощується, починається за ще не набраному повністю тексту. Для пошуку вперед треба вказати команду

C-s<текст>. Унизу екрану з'являється вікно пошуку Search з номером пошуку та відображеним текстом. Всі входження тексту відмічаються кольором. Можлива ситуація, при якій знайдено не весь текст, а тільки його частина. Такий текст відмічається також.

Аналогічно шукається текст назад. Набрати команду C-g<текст>.

Якщо у тексті є велика буква, то пошук розрізняє малі та великі букви. При помилці у тексті, що шукається, уживають команду DEL.

Невдача пошуку завершується повідомленням „Failed”. Можна повторити пошук змінивши текст. Повторний пошук завжди починається з початку буферу. Для повторного пошуку знову вживається команда C-s<текст> або C-g<текст>.

Вихід з пошуку за клавішею „ENTER” при невдачі вказує на прийняття невдачі. Вихід з пошуку при невдачі за командами C-q C-q. Команда виходу з пошуку при знайденому тексті C-q.

ПОШУК БЕЗ НАРОЩУВАННЯ

Звичайний пошук виконується за командою C-s „ENTER”. Унизу екрану з'являється вікно пошуку Search для набору тексту. Текст закінчується „ENTER”.

Можна набрати відразу дві команди:

C-s „ENTER” C-s Текст „ENTER”.

Після знаходження першого входження тексту курсор встановлюється після нього. Положення курсору відмічається „точкою”. Одержати всі входження тексту після „точки” можна за командами C-s C-s.

ТЕМА 5: ОСНОВНІ ПОНЯТТЯ COMMON LISP

5.1 Абстрактні типи даних

Стандарт Common Lisp засновано на абстрактних типах даних. При такому підході тип даних визначається через операції, які можливо виконувати над ними, а не через представлення даних у оперативній пам'яті. Застосування абстрактних типів даних, робить систему програмування відкритою для поповнення типів.

Один об'єкт може відноситися до кількох типів. На рисунку 5.1 зображено ієрархію типів даних.

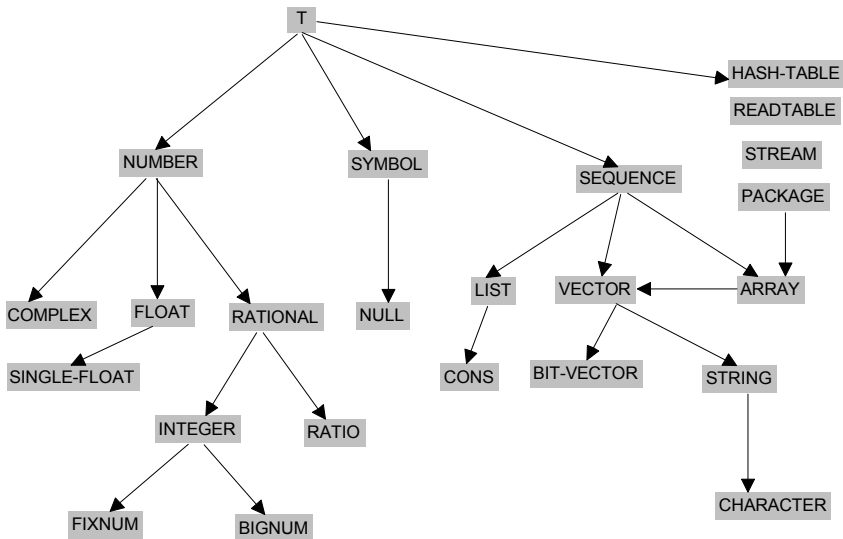


Рисунок 5.1-Ієрархія вбудованих типів у Common Lisp

З рисунку видно, що об'єкт певного типу є підтипом кількох типів, що розташовані у ієрархії вище. Наприклад, об'єкт типу STRING є підтипом типів VECTOR, ARRAY, SEQUENCE. Всі типи є підтипами типу T.

У Common Lisp типи даних не об'являються.

Для типів характерно, що функції роботи з даними цих типів можна застосовувати до даних підтипів. Наприклад, функції типу

SEQUENCE можна застосовувати для списків, рядків. Але це правило працює не для всіх типів. Так тип CHAR не можна оброблювати функціями для типу STRING, бо тип CHAR відноситься до атомів, а тип STRING до послідовностей.

Крім вказаних стандартних типів ми будемо застосовувати типи: асоціативні списки; списки властивостей, структури, атоми.

Розглянемо, у яких формах зображуються константи різних типів:

- а) $\#C(2\ 3)$ – тип комплексне число $2+i3$.
- б) 1.2 – тип single-float, форма запису з фіксованою точкою.
- в) $2e-2$ – single-float, форма запису з точкою, що пливе.
- г) $3/5$ – тип ratio, звичайний дріб.
- д) 279 – тип integer.
- е) 16777215 – fixnum, максимальне ціле.
- є) -16777216 – fixnum, мінімальне ціле.
- ж) 16777216111222333 – тип bignum, ціле більше ніж fixnum.
- з) $\#P$ – знак P типу character.
- и) "lisp" – тип рядок.
- і) $(2\ 3\ 4\ 5)$ - тип список.
- ї) $(A\ .\ N)$ - тип cons.
- к) $\#(2\ 4\ 3\ 1)$ - тип vector.
- л) $\#*101$ - тип bit-vector.
- м) $\#S(student :fam Petrenko : Микола)$ - структура

Основні операції на абстрактних об'єктах розподіляються на класи:

- функції конструктори (будують об'єкт);
- функції відбору (видбирають частину об'єкту, не змінюючи об'єкт);
- функції модифікатори (модифікують об'єкт);
- функції вводу-виводу об'єктів;
- функції компаратору;

– функції предикати.

Типи даних, що складаються з стандартних типів або раніше визначених користувачем типів, називають складними.

Перевірити, чи є об'єкт вказаного типу можна за функцією TYPEP:

```
CL-USER> (TYPEP 45 'integer)
```

T

Функція SUBTYPEP визначає чи є тип підтипом вказаного типу:

```
CL-USER> (SUBTYPEP 'Integer 'Number)
```

T

Функція TYPE-OF визначає до якого типу відноситься об'єкт і діапазон типу.

```
CL-USER> (TYPE-OF #\w)
```

STANDARD-CHAR

5.2 Поняття середовища COMMON LISP і його вміст

При завантаженні базової програми Common Lisp - CLisp.exe під середовище виділяється оперативна пам'ять.

Для середовища Common Lisp характерна пакетна організація. Середовище містить службові пакети. Знайти імена службових пакетів можна за функцією List-All-Packages.

```
CL-USER> (LIST-ALL-PACKAGES)
```

Повертається список покажчиків на стандартні пакети з іменами:

```
(GSTREAM, GRAY, I18N, SOCKET, FFI, SCREEN, CUSTOM,
EXT, CLOS, CS-COMMON-LISP-USER, CS-COMMON-LISP,
CHARSET, KEYWORD, SYSTEM, COMMON-LISP-USER,
COMMON-LISP, EXPORTING, POSIX, REGEXP).
```

Пакет COMMON-LISP-USER – поточний (активний) пакет при вході в програму Clisp.exe. Ім'я поточного пакету завжди привласнюється стандартній змінній цього пакету *PACKAGE*.

Пакет COMMON-LISP містить стандартні функції і змінні. Але користувачу заборонено інтернувати свої символи у цей пакет, тому був створений пакет COMON-LISP-USER, що успадковує від пакету COMMON-LISP функції та керуючі змінні і в якому користувач може працювати.

Пакет KEYWORD містить ключові слова, що впливають на режими роботи стандартних функцій. Ключові слова починаються з двокрапки «:». Пакет дозволяє обмежити ключові слова, щоб відкинути суперечливі ситуації в роботі функцій різних реалізацій стандарту.

Пакети CS-COMMON-LISP-USER, CS-COMMON-LISP містять відомості із стандарту відповідно пакетам COMMON-LISP-USER, COMMON-LISP.

Пакет CHARSET містить знаки. Якщо увійти в пакет за функцією (IN-PACKAGE "CHARSET"), то можна з'ясувати які є в ньому символи.

```
CHARSET> #\l
```

```
#\CYRILLIC_SMALL_LETTER_EL
```

Пакет SYSTEM призначено для взаємодії із операційною системою. Стандартні пакети обслуговуються інтерпретатором, але не всі успадковують функції Common Lisp. У кожного пакета своє призначення. В стандартні пакети теж можна інтернувати функції, якщо це необхідно.

Користувач може створювати свої пакети, в яких розташовує свої функції. Символи кожного пакету розподіляються на зовнішні (доступні іншим пакетам для їх виклику) і внутрішні, інтерновані в ядро свого пакету (таблицю з компонентами «рядок – символ»). Зовнішні символи створюють інтерфейс пакету. В той же час пакет користувача – складається з усіх активних на даний момент об'єктів Common Lisp (аналогічно середовищу мови Lisp).

Кожний пакет – можна вважати аналогом модуля програми, написаної іншою мовою програмування. В пакетах можна розташовувати бібліотеки. В обох випадках для застосування функцій та змінних зовні пакетів, необхідно зробити імена певних функцій та змінних зовнішніми символами.

Розглянемо взаємодію між пакетом COMMON-LISP-USER і пакетом користувача LOC на прикладах.

Приклад 1. Створити пакет LOC і зробити його поточним. Розмістити у ньому функцію VVOD, що вводить два числа та додає їх. Розмістити також в пакеті змінну D і привласнити їй значення.

```
CL-USER> (make-package 'LOC)
```

```
#<PACKAGE LOC>
```

```
CL-USER> (in-package "LOC")
```

```
#<PACKAGE LOC>
```

```
LOC> (defun vvod (x y)
```

```
(+ x y))
```

```
VVOD
```

```
LOC> (setq D 33)
```

```
33
```

Символи VVOD і D локальні в поточному пакеті LOC.

Приклад 2. Для виклику функції VVOD та змінної D з пакету CL-USER треба зробити символи VVOD та D зовнішніми у пакеті CL-USER, імпортуючи їх. Символ подають «назва пакету::символ». Перейти в пакет CL-USER, одержати значення змінної D та викликати функцію VVOD.

```
LOC> (import '(LOC::VVOD) 'CL-USER)
```

```
T
```

```
LOC> (import '(LOC::D) 'CL-USER)
```

```
T
```

```
LOC> (in-package "CL-USER")
```

```
#<PACKAGE COMMON-LISP-USER>
```

```
CL-USER> (VVOD 2 3)
```

```
5
```

```
CL-USER> D
```

```
33
```

Приклад3.Зробити символ присутнім в обох пакетах.

```
CL-USER> (EXPORT '(CL-USER::D) 'LOC)
```

T

Тепер символ можна не тільки одержати з обох пакетів, а і змінювати в обох пакетах.

5.3 Символи

Для подавання ідентифікаторів: змінних, імен функцій, макросів застосовують тип символ. Символ має зовнішнє і внутрішнє подавання.

Зовнішнє подавання – це набір будь-яких знаків (латинських та кирилических букв, цифр, спеціальних знаків без обмежень). Довжина зовнішнього подавання символу практично необмежена.

Для застосування цифри або спеціального знаку в якості символу, щоб відрізнити їх від макрознаків чи чисел, застосовують перед символом „\”. Наприклад, подамо символ цифрами \53.

```
CL-USER> (SETQ \53 25)
```

```
25
```

```
CL-USER> \53
```

```
25
```

Якщо зовнішнє подавання символу повинно містити розділові знаки, то щоб сприймати символ як ціле, його обмежують парними знаками “|”.

```
CL-USER> (SETQ | | 14e-1)
```

```
1.4
```

У прикладі символ містить 3 пропуски.

При вводі програми символи, що необмежені розділовими знаками автоматично переводяться у верхній регістр.

Наприклад:

```
CL-USER> (SETQ n 12.3)
```

```
12.3
```

```
CL-USER> N
```

```
12.3
```

Одержати Print - ім'я символу можна функцією SYMBOL-NAME:

CL-USER> (symbol-name 'R)

"R"

Мову LISP розробили для спілкування експертної системи з користувачем природною мовою та створення бази знань на основі мови.

Знання подаються в поняттях. Поняття - зміст слова. Слово – ідентифікатор поняття. Кожне поняття – передається групою тверджень. Кожне твердження вказує на певну властивість об'єкту. Наприклад, зміст слова «вишня» передається через властивості: «клас вишня фрукт», «смак вишня кислий» тощо.

Але змінна або функція теж мають імена і певні властивості. Здається, саме так виник у мові Лісп узагальнений об'єкт – символ.

Внутрішнє подання символу – це об'єкт, що розпізнається і складається з 4-х показників.

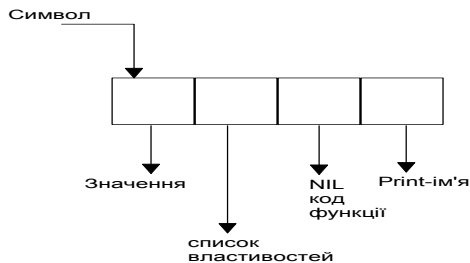


Рисунок 5.2 – Внутрішнє подання символу

Перше поле застосовується для показника на дане; друге для показника на властивість слова (символу) або іншого об'єкту мови Lisp; третє для показника на псевдокод функції; четверте для зовнішнього подання символу – Print-імені.

У середовищі слово подається символом. Для цього є спеціальна функція перетворення рядка у символ – INTERN. Для вказівки змісту слова існують властивості символу. Властивості символу подаються списком, а друге поле символу містить показник на цей список.

Експертна система повинна спілкуватися з експертом під час роботи програми, щоб одержувати від нього знання природною мовою.

Звідси виникла необхідність мати властивості створення символів під час завантаження у програму рядків.

Щоб утворити у середовищі символ, достатньо просто завантажити програму з файлу на вхід інтерпретатору або ввести вираз з клавіатури. У середовищі символ буде створено автоматично, якщо його не було. Для будь-якого символу створюється рядок – Print – ім'я символу.

Мова Common Lisp дозволяє генерувати унікальний символ у середовищі, якщо це необхідно. Для цього застосовується функція Gentemp.

```
CL-USER> (Gentemp "N")
```

```
|N2|
```

```
CL-USER> (Gentemp "N")
```

```
|N3|
```

Перевірити, чи є об'єкт символом, можна функцією Symbolp.

Наприклад:

```
CL-USER> (Symbolp 'a)
```

```
T
```

```
CL-USER> (Symbolp "a")
```

```
NIL
```

Перевірити чи є символ іменем функції та одержати тип функції можна функцією FBOUNDP. Перед перевіркою функції програміста вона повинна бути скомпільованою.

```
CL-USER> (fboundp '+)
```

```
#<Compiled-function + #x409FFC6>
```

```
CL-USER> (fboundp 'pop)
```

```
#(-227328000 #<Compiled-function POP Macroexpander #x491B68E>)
```

```
CL-USER> (fboundp 'setq)
```

```
#(-227328000 SETQ)
```

```
CL-USER> (defun a (x) (cdr x))
```

```
A
```

```
CL-USER> (fboundp 'a)
```

```
#<Compiled-function A #xC8B6E36>
```

5.4 Атоми і списки

Базовими типами мови є атом і список.

АТОМ

Атомами називають прості типи даних. З атомів будуються інші типи даних. До атомів відносяться символи, знаки, числа, логічні константи Т і NIL.

Перевірити, чи є об'єкт атомом, можна функцією ATOM:

```
CL-USER> (ATOM 263456789123456)
```

```
T
```

```
CL-USER> (ATOM 'ddd)
```

```
T
```

```
CL-USER> (ATOM NIL)
```

```
T
```

```
CL-USER> (ATOM "for")
```

```
T
```

ЛОГІЧНІ КОНСТАНТИ

Логічна константа Т служить для подавання значення «істина». Логічна константа NIL використовується в мові для подавання значення «лжа» і для подавання порожнього списку ().

У мові LISP всі значення, що не NIL вважаються істинними.

Нижче, подається приклад, що ілюструє твердження.

```
CL-USER> (NULL 234 )
```

NIL

Функція NULL перевіряє істинність свого аргументу. Якщо аргумент не істинний, то функція повертає T, інакше NIL:

Вказаний підхід дозволяє використовувати результат будь-якої функції, константу, значення змінної, як умову.

```
CL-USER> (IF (CAR '(2 5)) (PRINT "Істина"))
"Істина"
"Істина"
```

Перший раз слово друкує функція PRINT. Другий раз слово повертає функція IF, як результат останньої виконаної дії.

КОНСИ

Тип даного конс має наступне зовнішнє подавання:

(елемент1 . елемент2)

Конс складається з пари елементів, що пов'язані точкою і розміщені у круглих дужках. Елементами консами можуть бути будь-які об'єкти Lisp: список, символ, змінна, число тощо.

```
CL-USER> '(A . B)
(A . B)
```

Щоб відрізнити конс від дійсного числа, коли елементи числа, між елементами пари і точкою ставляться пропуски.

Наприклад,

```
CL-USER> (SETQ K '(3 . 4))
(3 . 4)
```

У середовищі конс – об'єкт, що розпізнається. Він має два покажчики, які вказують на елементи консу:

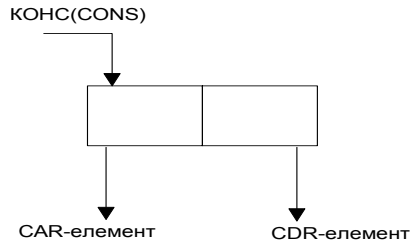


Рисунок 5.3-Внутрішнє подавання консу

Перевірити, чи є об'єкт консом, можна функцією CONSP:

```
CL-USER> (CONSP '(X . 30))
```

```
T
```

Конс (точкова пара) створюється функцією CONS із своїх аргументів:

```
CL-USER> ((CONS 'X 30)
```

```
(X . 30)
```

Функція CONS повертає побудовану точкову пару.

Функції CAR і CDR вибирають лівий і правий елементи першого консу списку відповідно. Сам конс при цьому не змінюється.

```
CL-USER> (CAR '(X . 30))
```

```
X
```

```
CL-USER> (CDR '(X . 30))
```

```
30
```

СПИСКИ

Список – будується на основі консів. CAR-елементи кожного консу списку вказують на елемент списку, CDR - елемент консу вказують на наступні конси. У останньому консу правий елемент має значення NIL, тобто покажчик відсутній.

Розглянемо, як у середовищі з консів подається список.

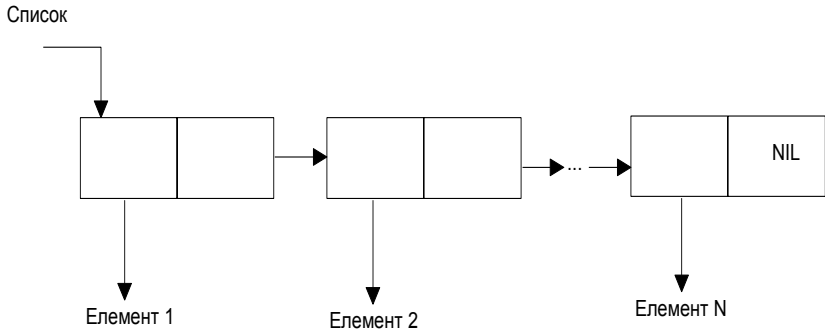


Рисунок 5.4-Внутрішнє подавання списку

Функції CAR і CDR працюють із першим консом списку.

```
CL-USER> (CAR '(1 2 3))
```

```
1
```

```
CL-USER> (CDR '(1 2 3))
```

```
(2 3)
```

Функція CAR одержує перший елемент списку і повертає його.

Функція CDR одержує хвіст списку (список без першого елемента) і також повертає його. Функції не змінюють сам список. Вони відносяться до функцій відбору.

Внутрішнє подавання консів і списків допомагає краще розуміти роботу з ними функцій. Конс тотожний списку з одним елементом.

$$'(1) \equiv '(1 . \text{NIL})$$

Запис списку можна трактувати, як запис із консів, тобто бінарним деревом: $(1\ 2\ 3) \equiv (1 . (2 . (3 . \text{NIL})))$.

На рисунку 5.5 поданий список $(1\ 2\ 3)$ консами у вигляді бінарного дерева.

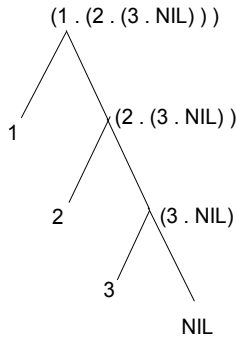


Рисунок 5.5 – Подання списку консами у вигляді бінарного дерева

Додати до списку перший елемент можна функції `CONS`. Функція будує нову точкову пару, лівий елемент якої містить покажчик на новий елемент, а правий елемент якої містить покажчик на перший елемент старого списку.

```
CL-USER> (setq L '(12 4 67))
(12 4 67)
```

```
CL-USER> (setq L (CONS 9 L))
(9 12 4 67)
```

Після додавання першого елементу до списку покажчик з символу `L` встановлюється на початок нового списку.

5.5 Інтерпретатор COMMON LISP

Вікно вводу в інтерпретатор називають «Стіл читання» - `ReadTable`. Інтерпретатор має стандартний синтаксис знакового зовнішнього подання виразу. Інтерпретатор також має засоби змінювати стандартний синтаксис під себе. Але за один сеанс роботи з `Common Lisp` може застосовуватися тільки один синтаксис.

По замовчанню в стандартному синтаксису всі символи перетворюються в великі. Увід залежить від аргументу функції `Readtable` – `Case`.

```
CL-USER> (READTABLE-CASE *READTABLE*)
:UPCASE
```

Змінна **READTABLE** повертає покажчик на своє значення:

```
CL-USER> *READTABLE*
#<READTABLE #xC19BB9E>
```

Інтерпретатор COMMON LISP має два процесори мови:

- процесор читання;
- процесор обчислення.

Процесор читання Lisp Reader одержує вхідні данні з клавіатури або з файлу. Він містить лексичний аналізатор і синтаксичний аналізатор.

Лексичний аналізатор процесору читання: читає потік знаків, виявляє в ньому S – вирази і передає їх синтаксичному аналізатору.

S - вираз є узагальненням типів атом і список.

Для перевірки, чи є вхідний об'єкт S – виразом, застосовують наступний алгоритм.

Надамо лічильнику дужок значення 0. Будемо називати об'єкт S - виразом, якщо при кожній відкритій дужці до лічильника додавати 1, а при кожній закритій дужці із лічильника віднімати 1. І при цьому всередині виразу значення буде весь час більше 0, а в кінці виразу значення лічильник дорівнює нулю.

Під це означення підходить атом і список. У атома немає дужок і тому лічильник дорівнює 0. У списку парні дужки.

Приклад 1.Розглянемо приклад вірного S – виразу:

```
CL-USER> (* (+ a b) (- c d))
Значення лічильника: 0,1,2,1,2,1,0.
```

Приклад 2.Подається вираз, у якого в кінці лічильник більше нуля (дужки непарні).

```
CL-USER> (* (+ a b) (- c d)
Значення лічильника: 0,1,2,1,2,1.
```

Якщо інтерпретатор, зустрічає такий вираз, то чекає вводу закритої дужки.

Приклад 3.

CL-USER> (+ (car '(2 4 6)) 2))

UnMatched – не відповідність дужок. Лічильник менше нуля.

Синтаксичний аналізатор процесору читання.

Аналізатор вводить знак поточного S-виразу і інтерпретує його відповідно синтаксису. Аналіз інтерпретатору базується на спецзнаках, коментаріях і макросимволах. Їх приклади подані нижче.

Спец.знаки:

«\» простий розділовий знак. Заперечує переворення символу великими буквами.

«| |» парні розділові знаки. Заперечує перетворення кількох символів великими буквами тощо.

Макросимволи:

«(», «)», «#» тощо.

Коментарії:

«;» - закоментувати від символу до кінця рядку або файлу;

«;;» - коментувати весь рядок;

«;;;» - закоментувати визначення функції;

«;;;;» - відноситься до коду.

Лексеми та числа виділяються за розділовими знаками і перевіряються за вмістом знаків. Числа вважаються лексемою, якщо перед числом стоїть розділовий знак «\» або парні знаки «| |».

Зовнішнє подавання S - виразів і одержану інформацію інтерпретують у внутрішнє за допомогою синтаксичних дерев. Древа передаються синтаксичному аналізатору процедури обчислення.

Синтаксичний аналізатор процесору обчислення: розглядає синтаксичні дерева і виявляє в них LISP - форми. Під формами у COMMON LISP розуміють вирази, що можуть обчислюватися.

Процесор обчислення, одержує форму, генерує її код, ініціює його виконання і повертає результат. Результат роботи відображується на екрані або поступає в файл.

ТЕМА 6: ФУНКЦІЇ, СПЕЦІАЛЬНІ ФОРМИ ТА МАКРОСИ

6.1 Базові функції

До базових функцій мови відносять функції, які ми розглянули при роботі із списками та консами. Це функції CAR, CDR, CONS.

СКЛАДЕНІ ФУНКЦІЇ

Часто потрібно застосовувати до списків комбінацію функцій CAR і CDR. LISP дозволяє коротко записати таку комбінацію функцій CAR і CDR одною складною функцією. Для цього між буквами C і R можна вставляти до 4-х символів «A» і / або «D».

Приклад 1:

```
CL-USER> (SETQ L (CAR (CDR '(5 3 1) ) ) )
3
```

Функція CDR одержує від списку (5 3 1) його хвіст (3 1). Функція CAR одержує від хвоста списку його голову 3.

Запишемо цей приклад складною функцією.

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

Внутрішні функції виконуються раніше, тому обчислення складних функцій виконується ліворуч.

Необхідними у роботі є також наступні функції

ФУНКЦІЯ SET

Функція (SET Символ Об'єкт) – обчислює значення аргументів і привласнює значенню 1-го аргументу значення 2-го аргументу. Функція повертає значення, яке привласнює значенню першого аргументу. Якщо після обчислення аргументів перший аргумент не символ, то фіксується помилка «Не символічний аргумент».

Приклад 1:

```
CL-USER> (SETQ b 21)
```

```

21
CL-USER> (SETQ k 'c)
C
CL-USER> (SET k b)
21
CL-USER> K
C
CL-USER> C
21

```

Привласнимо символу В 21, а символу К символ С. Функція SET обчислює обидва свої аргументи. Тому символ К має значення С, а символ В 21. В результаті символу С привласнюється 21.

Приклад 2:

```

CL-USER> (SETQ K 2)
2
CL-USER> (SETQ M 3)
3
CL-USER> (SET K M)

```

Помилка, видається повідомлення „Не символічний аргумент”.

ФУНКЦІЯ LIST

Функція може мати скільки завгодно фактичних параметрів. Функція обчислює значення своїх аргументів, будує з них список і повертає побудований список.

```
(LIST об'єкт1 об'єкт2 ... об'єктK)
```

Без функції LIST неможливо побудувати список із значень змінних.

Приклад 1. Нехай кільком змінним привласнюванні значення. побудувати список із значень цих змінних.

```
CL-USER> (SETQ A 1 B 4 C 5)
```

Створимо список із змінних із використанням дужок.

```
CL-USER> (SETQ L (A B C))
```

Обчислюються значення змінних. Виводиться повідомлення „Невідома функція А”.

Заборонимо обчислення списку функцією QUOTE, щоб інтерпретатор не розглядав список, як виклик функції:

```
CL-USER> (SETQ L '(A B C))
(A B C D)
```

Створюється список із символів.

Функція LIST дозволяє обчислювати значення змінних перед виконанням функції LIST.

```
CL-USER> (SETQ L (LIST A B C))
(1 4 5)
```

ФУНКЦІЯ SYMBOL-VALUE

Взагалі автоматично обчислюються змінні. Функція обчислює значення символу. Якщо символ не має покажчик на дане, то функція повертає помилку. Треба пом'ятати, що символи не можна записувати кирилицею.

```
(SYMBOL-VALUE Символ)
```

```
CL-USER> (setq s 'd)
D
```

```
CL-USER> (setq d 2)
2
```

```
CL-USER> (symbol-value 'd)
2
```

```
CL-USER> (symbol-value 's)
D
```

```
CL-USER> (symbol-value 'a)
Помилка. Змінна a не має значення.
```

ФУНКЦІЯ BOUNDP

Функція BOUND перевіряє, чи має символ посилку на дане. Якщо посилка є, функція повертає Т, інакше повертає NIL.

```
CL-USER> (setq 'a 6)
```

```
6
```

```
CL-USER> (IF (boundp 'a) (symbol-value 'a))
```

```
6
```

ФУНКЦІЯ LOAD

```
(LOAD "Шлях до файлу Ім'я файлу")
```

Функція застосовується для завантаження вмісту файлу на вхід інтерпретатору. По замовчанню вживається каталог із файлом LISP.BOX.

Приклад 1: Нехай в кореневому каталозі на диску D є файл із вмістом (Print "Доброго вам дня"). Тоді на екран виводяться коментарії початку завантаження із файлу та кінця завантаження із файлу і результат виконання функції Print.

```
CL-USER> (Load "D:\s.txt")
```

```
:: Loading file D:\s.txt ...
```

```
"Доброго вам дня"
```

```
:: Loaded file D:\s.txt
```

ФУНКЦІЯ COMPILE

Для компіляції програми (функцій) застосовуйте функцію компіляції.

```
CL-USER> (COMPILE-FILE-PATHNAME "D:\a.LSP")
```

```
#P"D:/a.wx32fsl"
```

Результати компіляції містять повністю розкриті макроси, що значно прискорює роботу функції.

6.2 Поняття спеціальної форми

Форма – загальна назва обчислювального виразу (константи, змінної, виклику функції, макросу тощо). Спеціальною формою називають таку конструкцію, яка порушує порядок виконання звичайний для функції (обчислити аргументи, обчислити за аргументами функцію).

Спеціальна форма SETQ виконує групове привласнювання значень символам.

(SETQ Символ1 Форма1 Символ2 Форма2 ... Символ N Форма N)

SETQ – може мати скільки завгодно пар аргументів.

Спеціальна форма SETQ поступово розглядає пари зліва направо. Для кожної форми виконуються дії: обчислюється форма, привласнюється символу значення форми.

Функція повертає останнє значення форми.

Спец.форма не просто обчислює аргументи, а виконує дії не властиві функції: розбиває аргументи на пари, заперечує обчислення кожного першого елементу пари - символу.

Наприклад:

```
CL-USER> (SETQ M 3 K (* M 4) N (- K 2))
```

```
10
```

```
CL-USER> K
```

```
12
```

```
CL-USER> N
```

```
10
```

Значення змінної K залежить від того, яке значення було привласнено символу M, а значення символу N залежить від того, яке значення було привласнено K.

Запис (SETQ M 3) тотожній запису (SET (QUOTE M) 30)

6.3 Макроси

Макросом називають скорочено макрокоманду. Макрокоманди не обчислюють свої аргументи, а передають їх в макровизначення, привласнюючи їх формальним параметрам. В результаті одержується макророзширення. Тобто, підставляючи замість формальних параметрів різні форми ми можемо створити групу функцій на базі одного макровизначення.

Розглянемо якими засобами можна створити макровизначення. Функції створюються на базі списків. Тому побудувати макровизначення можна теж застосовуючи функцію LIST та символи інших функцій.

Макровизначення компілюється функцією DEFMACRO.

Приклад 1.

```
CL-USER> (DEFMACRO SUM (x y) (List '+ x y))
```

```
SUM
```

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

```
13
```

Змінним X і Y привласнюються значення (* 3 3) і (* 4 4) відповідно.

Перетворення макровизначення:

```
(DEFMACRO SUM (X Y) (LIST '+ X Y))
```

```
(DEFMACRO SUM ((X Y) (LIST '+ (* 3 3) (* 4 4))))
```

Макрос обчислюється в два етапа.

1. Обчислюється функція LIST та її аргументи:

```
(DEFMACRO SUM (X Y) (+ 9 16))
```

2. Обчислюється одержана функція «+».

МАКРОС PSETQ

Аргументи макросу PSETQ діляться на пари, як у SETQ. Проте обчислення значень виразів макросу PSETQ виконується по-іншому. Спочатку обчислюються другі аргументи кожної пари, після чого виконуються привласнювання символів обчисленні значення. Макрос

завжди повертає NIL. Реалізація обчислювальної конструкції спец.формою або макросом залежить від розроблювача. Макроси можуть вкладатися один в одного.

Приклад 1:

```
CL-USER> (SETQ M 3 C 10)
```

```
CL-USER> (PSETQ C 4 M (+ 5 M) P (* C M M) )
```

```
NIL
```

```
CL-USER> M
```

```
8
```

```
CL-USER> P
```

```
90
```

Порівняємо результати обчислень тих же виразів функцією SETQ.

```
CL-USER>(SETQ M 3 C 10)
```

```
CL-USER> (SETQ C 4 M (+ 5 M) P (* C M M) )
```

```
256
```

6.4 Функції компаратору

Виконання програми може залежати від властивостей або значень даних. При певних умовах треба виконати одні дії, а при інших умовах інші дії. Для цього у COMMON LISP існує великий набір функцій порівняння значень даних. Слово „компаратор” виникло від англійського слова „compare”, що означає „порівняти”.

Всі функції класу «компаратор» мають два аргументи і повертають істину, якщо вони порівнюванні по їх тесту, інакше повертають NIL. Кожна функція працює із своїми типами об'єктів.

ФУНКЦІЯ EQ

Достоїнством мови LISP є подавання у середовищі однакових символів і цілих чисел один раз. За рахунок вказаного середовище невелике за розміром. В середовищі існують списки символів і цілих чисел (fixnum), у яких інтерпретатор перевіряє наявність вказаних

об'єктів. Якщо такі числа або символи вже існують, то їх не створюють в середовищі, а тільки вказують на них посилки.

До малих цілих чисел відносяться числа в діапазоні:

$-16777216 \leq N \leq 16777215$

Функція EQ перевіряє чи розташовані об'єкти середовища за однією адресою. Інакше кажучи, функція повертає Т, якщо вказані об'єкти представляють собою один об'єкт.

Приклад 1:

CL-USER> (EQ 'a 'A)

NIL

Перший символ має знак „\”, тому ім'я символу не переводиться у верхній регістр. Тому символи різні.

Приклад 2:

CL-USER> (EQ 'a 'A)

T

Ім'я першого символу переводиться у верхній регістр.

Функція не порівнює числа різних типів та числа, що подаються у різних формах.

Приклад 3:

CL-USER> (EQ 2 2)

T

Числа різних типів зберігаються по різному у середовищі. Дійсне число подається у вигляді скороченого дробу і має два покажчика на чисельник та знаменник.

CL-USER> (EQ 0.5 2/4)

NIL

Наприклад: 2/4 зберігається, як об'єкт, що має 2 покажчики, що вказують на чисельник 1 та знаменник 2. Дріб 2/4 автоматично скорочується, тому порівнюються числа 1/2 і 0.5.

Числа, що подаються у різних зовнішніх формах також не ідентичні об'єкти.

```
CL-USER> (EQ 2e+1 20.0)
NIL
```

У прикладі обидва числа, що порівнюються, дорівнюють 20.
Всі інші типи даних не розташовані за одною адресою.

```
CL-USER> (eq "r" "r")
NIL
```

ФУНКЦІЯ EQL

Функція EQL порівнює числа за величиною, не зважаючи на їх величину (fixnum або bignum). У прикладі 1 порівнюються 2 великих цілих.

```
CL-USER> (EQL 100000000 100000000)
T
```

Функція не може порівнювати числа різних типів (ціле і дійсне).

```
CL-USER> (EQL 2.0 2)
NIL
```

Функція порівнює числа записані у різних формах (з фіксованою точкою і з плаваючою точкою).

```
CL-USER> (EQL 2.0 0.2e+1)
T
```

Функція не може порівнювати складні дані.

```
CL-USER> (EQL '(1 2 3) '(1 2 3))
NIL
```

Функція не може порівнювати рядки.

```
CL-USER> (EQL "Q" "Q")
NIL
```

Функція може порівнювати символи і знаки.

```
CL-USER> (EQL 'Q 'Q)
T
```

```
CL-USER> (EQL '#Q '#Q)
T
```

ФУНКЦІЯ EQUAL

Функція порівнює дані будь-якого типу, але не порівнює числа різних типів.

```
CL-USER> (EQUAL '#Q '#Q)
```

```
T
```

```
CL-USER> (EQUAL 'Q 'Q)
```

```
T
```

```
CL-USER> (EQUAL "Q" "Q")
```

```
T
```

```
CL-USER> (EQUAL '(1 2 3) '(1 2 3))
```

```
T
```

```
CL-USER> (EQUAL 2.0 0.2E+1)
```

```
T
```

```
CL-USER> (EQUAL 2.0 2)
```

```
NIL
```

```
CL-USER> (EQUAL '(2.0 1) '(2 1))
```

```
NIL
```

Функція EQUALP

Функція універсальна, її використовують для порівняння будь-яких типів об'єктів і об'єктів, що подаються у різних формах.

```
CL-USER> (EQUALP '(2.0 1) '(2 1))
```

```
T
```

```
CL-USER> (EQUALP '(1.52e+1 (1 2)) '(15.2 (1.0 4/2)))
```

```
T
```

```
CL-USER> (EQUALP '(1.52e+1 (1 A)) '(15.2 (1.0 a)))
```

```
T
```

ФУНКЦІЯ =

Функція порівнює тільки числа

Для порівняння чисел, що подаються у різних формах запису використовують функцію „`=`”. Аналогічно працюють функції: `<`, `<=`, `>`, `>=`, `/=`.

Приклад 1:

У прикладі порівнюються числа різних типів ціле та дійсне.

CL-USER> (= 2 2.0)

T

Приклад 2:

У прикладі порівнюються числа різних типів звичайний дріб та дійсне.

CL-USER>> (= 1/2 0.5)

T

Приклад 3:

У прикладі порівнюються дійсні числа, що подаються різними формами запису: з фіксованою точкою і у експоненціальній формі.

CL-USER> (= 1.52e+1 15.2)

T

Приклад 4:

Функція не може порівнювати символи.

CL-USER> (= 'A 'a)

Помилка, повідомлення „Не число”.

Приклад 5:

Проте, наступний приклад вірний, бо A і B змінні.

CL-USER> (SETQ A 12.0 B 12)

CL-USER> (= A B)

T

Для порівняння рядків існують наступні функції: string=, string<, string>, string/=, string<=, string>=. Виконується лексиграфічне порівняння (за кодами).

CL-USER> (string/= "23aT" "23AT")

2

Рядки не порівнюються починаючи із другої позиції.

CL-USER> (string< "dsa" "asf")

NIL

```
CL-USER> (string< "axc" "bxc")
(string< "axc" "bxc")
0
CL-USER> (string> "as" "as")
NIL
```

Аналогічні є функції для порівняння знаків: CHAR=, CHAR< тощо.

```
CL-USER> (char< #\h #\a)
NIL
```

6.5 Функції предикати

Клас функцій, що перевіряють певний тип об'єкту називають функціями предикатами. Функції мають один аргумент і повертають Т або NIL. Як ознака предикату, більшість назв функцій закінчується буквою „P”.

ФУНКЦІЯ ATOM

Функція перевіряє, чи є об'єкт атомом.

```
CL-USER> (atom 23)
T
CL-USER> (atom 1000000000)
T
CL-USER> (atom "qwerty")
T
CL-USER> (atom 'ASD)
T
CL-USER> (atom '#\q)
T
CL-USER> (atom '(1 2 3))
NIL
CL-USER> (atom '(q . w))
NIL
```

ФУНКЦІЯ NUMBERP

Функція перевіряє, чи є об'єкт числом.

CL-USER> (Numberp 2.4)

T

CL-USER> (Numberp "12")

NIL

ФУНКЦІЯ INTEGERP

Функція перевіряє, чи є об'єкт цілим числом.

CL-USER> (INTEGERP 12)

T

CL-USER> (INTEGERP 12.0)

NIL

CL-USER> (INTEGERP "12.0")

NIL

ФУНКЦІЯ PLUSP

Функція перевіряє, чи є об'єкт додатнім числом.

CL-USER> (PLUSP 245)

T

CL-USER> (PLUSP -15)

NIL

CL-USER> (PLUSP 0)

NIL

CL-USER> (PLUSP "35")

Помилка. Не числовий аргумент.

ФУНКЦІЯ MINUSP

CL-USER> (MINUSP -245)

T

CL-USER> (MINUSP 15)

NIL

CL-USER> (MINUSP 0)

NIL

CL-USER> (MINUSP “-6”)

Помилка. Не числовий аргумент.

ФУНКЦІЯ ZEROP

Функція перевіряє, чи є число нулем.

CL-USER> (ZEROP 0)

T

CL-USER> (ZEROP 10)

NIL

CL-USER> (ZEROP “10”)

Помилка. Не числовий аргумент.

ФУНКЦІЯ ODDP

Функція перевіряє, чи є число непарним.

CL-USER> (ODDP 5)

T

CL-USER> (ODDP 10)

NIL

CL-USER> (ODDP “10”)

Помилка. Об’єкт не ціле число.

ФУНКЦІЯ EVENP

Функція перевіряє, чи є число парним.

CL-USER> (EVENP 10)

T

CL-USER> (EVENP 9)

NIL

CL-USER> (EVENP “10”)

Помилка. Об’єкт не ціле.

ФУНКЦІЯ NULL

Функція перевіряє, чи є значенням об'єкту NIL, або чи є об'єкт порожнім списком.

```
CL-USER> (NULL '())
```

```
T
```

```
CL-USER> (NULL '(1 3 5))
```

```
NIL
```

```
CL-USER> (NULL (PLUSP -24))
```

```
T
```

```
CL-USER> (NULL T)
```

```
NIL
```

```
CL-USER> (NULL NIL))
```

```
T
```

З останніх двох прикладів видно, що функція працює як логічна функція NOT:

```
CL-USER> (NOT T)
```

```
NIL
```

```
CL-USER> (NOT NIL)
```

```
T
```

6.6 Логічні функції

Функція AND

Функція може мати будь-яку кількість аргументів і повертає істинну, якщо всі її аргументи істинні. Функція працює за скороченим алгоритмом до першого ложного результату при обчисленні аргументів.

```
CL-USER> (AND (PLUSP 7) (+ 2 3))
```

```
5
```

Після обчислення аргументів функції AND її аргументи T і 5. У мові LISP всі значення крім NIL істинні. Результатом буде результат обчислення останнього аргументу.

ФУНКЦІЯ OR

Функція повертає істину, якщо хоча б один її аргумент істинен. Результатом буде результат обчислення істинного аргументу. Якщо всі аргументи невірні, то повертається NIL.

```
CL-USER> (OR 2 (STRINGP #C) (ATOM "A"))
2
```

ФУНКЦІЯ NOT

Функція повертає істиність протилежну істиності свого аргументу.

```
CL-USER> (NOT (+ 2 4))
NIL
```

ТЕМА 7: УЗАГАЛЬНЕННЯ ПОНЯТТЯ «ЗМІННА»**7.1 Застосування форми у якості «місця»**

У мовах нижнього та середнього рівня змінна - це вічко або група вічок з іменем (адресою), яким привласнюються значення. Такий підхід не дозволяє одній змінній привласнювати дані різних типів, бо дані різних типів займають різну кількість байтів і мають різну організацію. У алгоритмічних мовах вищого рівня, що не мали абстрактних типів цей підхід зберігається.

У мові LISP змінна подається символом, а значення прив'язується до символу покажчиком. Прив'язку виконують функцією SET або спеціальною формою SETQ. Символ і його значення це два різних об'єкти. Такий підхід дозволяє вводити динамічну типізацію. Змінна не має типу, тому до символу можна прив'язати значення будь-якого типу.

Крім вказаного поняття змінної, у мові COMMON LISP вводиться узагальнення поняття змінної - форма. Під формою ми завжди розуміли змінну або виклик функції, яка одержувала значення данного або його частину.

Наведемо, приклад форми, що одержує 2-гий елемент списку:

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

```
CL-USER> (CADR M)
4
```

У COMMON LISP застосування форми замість змінної називають «місце». Форму розуміють, як узагальнену змінну, а результат обчислення форми, як значення змінної. Форма може подаватися тільки у символьному виді і *вказує* на дане або частину даного.

Застосування форми, як «місця», дозволяє звертатися до вічок з даними за показчиками своїх символів. Форма одержує показчик від символу на ціле дане і знаходить через нього посилку на необхідну частину даного.

Для роботи з узагальненими змінними, що подаються формами, COMMON LISP застосовує макроси. В цьому розділі ми розглянемо два макроси роботи із «місцем».

7.2 Макроси роботи з «місцем»

МАКРОС SETF

«Привласнювання значення місцю», на яке вказує форма виконується макросом SETF. Треба пам'ятати, що насправді змінюється показчик форми на інше дане. Показчик форми обчислюється через показчик цілого даного. Макрос повертає значення, на яке змінився показчик.

Приклад 1:

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

```
CL-USER> (SETF (CADR N) 3)
3
```

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

У прикладі функція SETF змінює покажчик CAR - елементу другої точкової пари на число 3. Ясно, що запису за місцем не виконується, бо цілі числа і знаки створюються в середовищі один раз. Функція EQ повертає Т, якщо об'єкти знаходяться за одною адресою.

```
CL-USER> (EQ (CADDR N) (NTH 2 N))
```

```
T
```

Приклад 2. Покажемо, що рядок складається з посилань на знаки.

```
CL-USER> (setq S "Abcd")
```

```
"Abcd"
```

```
CL-USER> (setf (char s 2) #\A)
```

```
#\A
```

```
CL-USER> s
```

```
"AbAd"
```

```
CL-USER> (EQ (CHAR S 0) (CHAR S 2))
```

```
T
```

МАКРОС INCF

Макрос збільшує значення форми (число) на один. Форма подається у символьному виді. Макрос має необов'язковий параметр. Він показує на скільки треба збільшити число.

Приклад 1.

```
CL-USER> (setq V 2)
```

```
2
```

```
CL-USER> (incf v)
```

```
3
```

```
CL-USER> (incf v 111222333444555666777)
```

```
111222333444555666780
```

```
CL-USER> (setq n 111222333444555666780)
```

```
111222333444555666780
```

```
CL-USER> (eq v n)
```

```
T
```

МАКРОС PUSH

Макрос модифікує список, вставляє перед вказаною формою елемент і повертає список першим елементом якого є вставлений елемент. Макрос працює із списками за принципом стеку.

```
CL-USER> (setq n '(2 4 6))
```

```
(2 4 6)
```

```
CL-USER> (push 5 n)
```

```
(5 2 4 6)
```

```
CL-USER> n
```

```
(5 2 4 6)
```

```
CL_USER> (push 10 (cddr n))
```

```
(10 4 6)
```

```
CL-USER> n
```

```
(5 2 10 4 6)
```

МАКРОС POP

Макрос вилучає перший елемент вказаної форми. Повертається вилучений елемент. Макрос працює із списками за принципом стеку.

```
CL-USER> (setq m '(34 15 6 7))
```

```
(34 15 6 7)
```

```
CL-USER> (pop (cddr m))
```

```
6
```

```
CL-USER> m
```

```
(34 15 7)
```

```
CL-USER> (pop (cdr m))
```

```
15
```

ТЕМА 8: КОНСТРУКЦІЇ, ЩО КЕРУЮТЬ ВИКОНАННЯМ ПРОГРАМИ

Програма мовою LISP подається структурою з функцій користувача, що пов'язані між собою викликами. У програмі може бути головна функція, з якої викликаються інші функції, а можуть бути кілька окремих функцій, що користуються одною базою даних.

Будемо називати конструкціями, що керують виконанням програми, форми, які змінюють порядок виконання функцій програми.

8.1 Спеціальна форма QUOTE

Спеціальна форма QUOTE відноситься до типу конструкцій робота з контекстом. Спеціальна форма впливає на процедуру обчислення інтерпретатору, забороняючи обчислення певних функцій і об'єктів середовища.

8.2 Спеціальна форма PROGN

(Progn Форма1 Форма2 ... Форма N)

Форма застосовується як операторні дужки. Вона примушує виконуватися форми, що розташовані в її тілі послідовно зліва направо і повертає значення, яке обчислює остання форма.

Приклад 1: У прикладі обчислюється і друкується сума, повертається перший елемент списку.

```
CL-USER> (PROGN (SETQ K (+ 3 4)) (PRINT K) (CAR '(1 2 3))) )
```

```
7
```

```
1
```

8.3 Спеціальна форма IF

Спеціальна форма IF – залежно від істинності умови обчислює THEN- форму або ELSE-форму:

(IF умова THEN-форма ELSE-форма)

Слова THEN та ELSE не пишуться. Формою THEN або ELSE може бути тільки одна форма. Щоб виконати кілька форм застосовують операторні дужки PROGN.

Форма IF може мати будь-яку вкладенність.

Приклад 1. Вводиться дане. Якщо дане число, то воно збільшується в 1.5 рази і виводиться. Інакше списку записуються к зворотному порядку і повертається останній елемент вхідного списку.

```
CL-USER> (DEFUN SP (R)
(IF (NUMBERP R)
      (PROGN (INCF R (/ R 2)) (PRINC R) )
      (IF (LISTP R)
            (PROGN (NREVERSE R)(CAR R) ) )))
```

SP

```
CL-USER> (SP 4)
```

6

```
CL-USER> (SETQ A '(2 4 7))
(2 4 7))
```

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

7

```
CL-USER> A
(7 4 2)
```

Функція NREVERSE – модифікує вхідний список . Вона записує елементи списку в зворотному порядку.

8.4 Спеціальна форма COND

Спеціальна форма Cond – застосовується для виконання певних дій за умовами.

Спец. форма подається:

(Cond пара1 пара2 ...параN).

Кожна пара подається: (умова форма).

COND переглядає пари зліва направо, перевіряючи умови кожної пари. Якщо умова пари має місце, то обчислюється форма пари і результат повертається. Наступні пари не розглядаються.

Приклад 1. Функція перевіряє знак числа.

```
CL-USER> (defun s (R) (cond ((minusp R) 1)((plusp R) 2) ((ZEROP
R) 3) ) )
S
CL-USER> (s 8)
2
```

Вводиться додатнє число, тому працює друга пара.

Приклад 2. Функція перевіряє знак числа.

```
CL-USER> (defun s (R) (cond ((minusp R) 1)((plusp R) 2) ))
S
CL-USER> (s 0)
NIL
```

Жодна умова пари не виконується, повертається NIL.

Приклад 3. Розглянемо приклад, де остання пара COND є пастка, яка спрацює, якщо попередні умови не виконані.

```
CL-USER> (defun s (R) (cond ((minusp R) 1)((plusp R) 2) (T )))
S
CL-USER> (s 0)
T
```

Дія пари відсутня тому повертається теж саме значення T.

Загальний принцип мови: будь-яке значення крім логічної константи NIL вважається істинним значенням. Це дозволяє у конструкціях, що керують, результат будь-якої форми, що не повертає NIL застосовувати як логічне значення.

Приклад 4. Вводиться дане. Якщо вводиться NIL, то програма закінчує свою роботу інакше створюється і повертається список.

```
CL-USER> (Defun v () (setf r (read)))(Cond ((Null r) NIL) ((List r) )))
V
CL-USER> (v)
“Всесвіт”
( “Всесвіт” )
```

ТЕМА 9: ФОРМАЛЬНІ ТА ФАКТИЧНІ ПАРАМЕТРИ

9.1 Обов'язкові параметри

Формальні параметри означення функції локальні – вони працюють тільки в контексті означення функції. Означення функції може мати будь-яке число формальних параметрів або не мати їх. Виклик функції повинен мати стільки фактичних параметрів скільки є формальних. Такий тип параметрів називають обов'язковими параметрами.

9.2 Необов'язкові ключові параметри

Функція може мати необов'язкові параметри (параметри які не треба вказувати у фактичних параметрах). Такі параметри необхідні для роботи функції, але користувач не повинен знати про внутрішню роботу функції. Неможна вимагати від користувача допоміжні параметри! У означенні функції такі формальні параметри розміщуються після обов'язкових формальних параметрів і починаються з &Optional.

(Обов'язкові параметри &Optional ((Параметр Значення) ... (Параметр Значення) Параметр ... Параметр)

При виклику функції обов'язковим формальним параметрам надаються значення фактичних параметрів. Необов'язковим параметрам привласнюються їх значення. Якщо у необов'язкових формальних параметрів немає значення, то їм привласнюється NIL.

Приклад 1. Треба створити списки, що містять відомості про студентів групи 413: № групи, прізвище, і відомості про борги. Номер групи занести за замовчанням. Відсутність боргів за замовчанням позначимо константою NIL.

```
CL-USER> (Defun Stud (Pr &Optional Borg (Gr 413)) (List Gr Pr Borg))
```

```
STUD
```

```
CL-USER> (Stud 'Коваль)
```

(413 Коваль NIL)

Замовчання можна замінити на інші значення, але не заповнювати можна тільки останні параметри.

CL-USER> (Stud 'Петренко 2)

9.3 Іменовані ключові параметри

Для роботи із стандартними функціями треба знати не тільки вміст параметрів, а і їх порядок. За бажанням розробника ПЗ можна не фіксувати порядок фактичних параметрів, а фіксувати назви параметрів, зневажаючи їх порядок, і навіть не вводити всі параметри.

Такий підхід реалізується за допомогою іменованих ключових параметрів. Функція може мати довільну кількість фактичних параметрів і довільний порядок їх розташування, але не більше вказаної у формальних параметрах.

На початку формальних параметрів функції розміщують &Key, а за ними назви формальних параметрів або пари (назва параметру – замовчання). Формальним параметрам, що не мають значення по замовчанню привласнюється NIL.

(Обов'язкові_параметри &Key (Параметр Значення) ...
(Параметр Значення) Параметр ...Параметр)

Фактичні параметри не є позиційними, їх можна задавати не в тому порядку, як вони записані у формальних параметрах.

У виклику функції вказують тільки необхідні фактичні параметри. Перед кожним ім'ям фактичного параметру ставлять двокрапку і через пропуск його значення. Якщо фактичний іменований параметр відсутній, то він містить значення NIL.

Приклад 1: Формальні параметри подаються за замовченням NIL. При наявності фактичного параметру застосовується його значення.

```
CL-USER> (defun K1 (&key a b c) (List a b c))
K1
```

```
CL-USER> (K1)
```

```
(NIL NIL NIL)
```

```
CL-USER> (K1 :a 5 :b 6 :c 7)
```

```
(5 6 7)
```

```
CL-USER> (K1 :c 5 :a 7)
```

```
(7 NIL 5)
```

```
CL-USER> (K1 :a 5 )
```

```
(5 NIL NIL)
```

```
CL-USER> (K1 :b 5 )
```

```
( NIL 5 NIL)
```

Приклад 2: Формальні параметри подають у дужках разом із значенням по замовчанню.

```
(DEFUN K1 (&key (a 10) (b 20) (c 30 )) (LIST a b c ))
```

```
K1
```

```
CL-USER> (K1)
```

```
(10 20 30)
```

Іноді треба задати значенням фактичного параметру NIL. Щоб розрізнити NIL по замовчанню і фактичний параметр NIL, для формального параметра вводиться додатковий параметр (символ), який буде містити T, якщо значення NIL було фактичним параметром.

Приклад 3. Параметр b містить додатковий символ.

```
CL-USER> (defun K1 (&key (a 10) (b 20 s) (c 30 )) (list a b s c ))
```

```
K1
```

```
CL-USER> (K1 :a 1 :b NIL :c 3)
```

```
(1 NIL T 3)
```

В нашому прикладі параметр b контролюється.

```
CL-USER> (K1 :a 1 :b 2 :c 3)
```

```
(1 2 T 3)
```

9.4 Остаточні ключові параметри

Застосування у визначенні функції `&REST` дозволяє застосовувати у виклику функції будь-яку кількість фактичних параметрів.

(ОБОВ'ЯЗКОВІ_ПАРАМЕТРИ `&REST` ЗМІННА)

Якщо за обов'язковими фактичними параметрами є параметр `&REST`, то фактичні параметри збираються у список і привласнюються змінній, що йде за параметром `&REST`.

У прикладі показано, як формується з фактичних параметрів список.

```
CL-USER> (DEFUN SUM (&REST L)(PRINT L))
SUM
```

```
CL-USER> (SUM 1 2 3)
```

```
(1 2 3)
```

```
(1 2 3)
```

Всі типи параметрів одночасно вживають рідко, бо між ними можуть бути конфліктні ситуації. Параметри записують у такій послідовності: обов'язкові, необов'язкові, іменовані, остаточні.

ТЕМА 10: РЕКУРСІЯ

Термін рекурсія походить з латинської *cursus* (рух). Рекурсивними можуть бути як дані, так і процедури. Ми будемо розглядати рекурсивні дані типу **SEQUENCE** (послідовності). Рекурсивні типи даних зручно оброблювати рекурсивними функціями.

Функція називається рекурсивною, якщо в її тілі знаходиться виклик себе безпосередньо або вона викликає себе через іншу функцію.

Рекурсивні функції обов'язково мають граничну умову і рекурсивну умову. Без граничної умови рекурсія буде не результативною. Аргументи рекурсивного виклику повинні обов'язково сходитися до аргументів граничної умови.

Рекурсії бувають різних типів. Проста рекурсія (*simple*) має у кожній гілці функції один рекурсивний виклик. Нижче подані два приклади простої рекурсії.

10.1 Висхідний метод рекурсії

Висхідна рекурсія виконується за один етап. Від початкової граничної умови до кінцевої граничної умови. Рекурсивні функції реалізуються формою **COND**.

Розглянемо приклад рекурсивної функції, що написана висхідним методом.

Приклад 1: Нехай змінна **R** посилається на список (1 3 5 7). Побудувати список **N**, елементи якого розташуються у зворотному порядку.

Необов'язковий формальний параметр **M** за замовчанням одержує значення **NIL**.

Перша пара макросу **COND** являє собою граничну умову - перевірку на кінець списку і повернення результату.

Друга пара містить рекурсивну умову – **T**, а у рекурсивного виклику функції вхідний список за кожним викликом зменшується на **CAR** – елемент і цей елемент приєднується до початку нового списку.

Дії повторюються до тих пір, поки вхідний список стане порожнім, а всі його елементи приєднуються до нового списку.

У програмі тільки одна рекурсивна гілка і в неї тільки один рекурсивний виклик тому рекурсія проста. Проста рекурсія відповідає циклу.

Порядок розташування гілок COND у функції важливий.

```
CL-USER> (SETQ N '(1 3 5 7))
(1 3 5 7)
CL-USER> (DEFUN REV (N &OPTIONAL M)
(COND ((NULL N) M)
(T (REV (CDR N) (CONS (CAR N) M))))))
REV
```

Відкомпільовану функцію можна трасувати. Для цього викликається директива відлагоджувача. Назва функції не заперечується, бо це не виклик функції. Перед трасуванням функція компілюється; після поточної компіляції директива запускається знову.

В деяких реалізаціях налагоджувач викликається із командного рядку із додатковим параметром GDB --annotate=3 lispbox.bat.

```
CL-USER> (Trace Rev)
T
```

```
CL-USER> (REV N)
(7 5 3 1)
```

Нижче показано трасування функції Rev.

На вхід подається список (1 3 5 7) і порожній список для реверсованого списку.

Формальні параметри рекурсивних викликів розташовуються в стеку, для кожного наступного рекурсивного виклику окремо. Під час трасування формальні параметри будуть змінюватися наступним чином:

	N	M
1.Trace:	(Rev '(1 3 5 7) '())	↓
2.Trace:	(Rev '(3 5 7) '(1))	
3.Trace:	(Rev '(5 7) '(3 1))	
4.Trace:	(Rev '(7) '(5 3 1))	
5. Trace:	(Rev '() '(7 5 3 1))	

10. 2 Низхідний метод рекурсії

Низхідний метод рекурсії реалізується двома етапами: редукції і одержання рішення.

Метод редукції – це спрощення задачі за рахунок декомпозиції її на більш прості задачі. На фазі редукції аргументи рекурсивного виклику сходяться до аргументів граничної умови.

Фаза одержання рішення поступово komponує рішення основної задачі з рішень більш простих задач.

Приклад 2. Нехай є узагальнений список чисел:
((5 3 7) (10 4 8 9) (6 8 2 5)).

Треба зробити список, елементами якого є суми внутрішніх списків, а порядок сум повинен відповідати порядку внутрішніх списків.

Програма містить дві функції. Функція Sum знаходить суму чисел кожного внутрішнього списку.

```
CL-USER> (Defun Sum (U)
(Cond ((Null U) 0)
(T (+ (Sum (Cdr U)) (Car U))))))
SUM
CL-USER> (Defun SP (U)
(Cond ((Null U) NIL)
```

```
(T (Cons (Sum (Car U))(SP (Cdr U)) ))))
```

```
SP
```

```
CL-USER> (Setq U '( (5 3 7) (10 4 8 9) (6 8 2 5)))
```

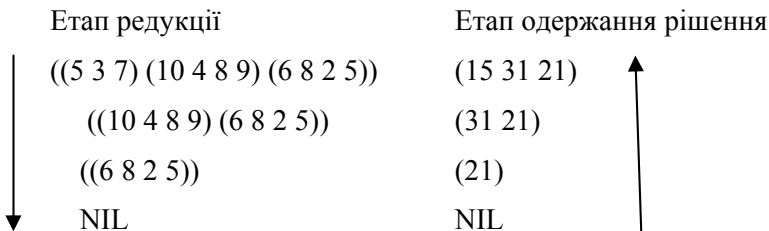
```
((5 3 7) (10 4 8 9) (6 8 2 5))
```

```
CL-USER> (SP U)
```

```
(15 31 21)
```

Програма містить дві функції, що написані низхідним методом. Функція Sum знаходить суму числового списку. В нашому прикладі вона буде знаходити суми внутрішніх списків.

Рекурсивний виклик функції SP із зменшеним списком на один елемент реалізує етап редукції. Етап закінчується, коли вхідний список порожній. Тоді покладається і вихідний список порожнім. На етапі одержання рішення друга функція SP будує вихідний список функцією Cons. Функція Cons викликає функцію Sum і одержану суму приєднує до вихідного списку.



10.3 Паралельна рекурсія

Рекурсію називають паралельною, якщо вона зустрічається одночасно в кількох аргументах однієї функції.

Приклад 3. Функція знаходить загальну суму всіх елементів числового списку. Якщо елемент підсписок, то спочатку знаходиться сума усіх чисел підсписку і додається до суми чисел зовнішнього списку.

Тіло функції містить функцію «+», яка має своїми аргументами два рекурсивних виклики. Тому рекурсія паралельна. Рекурсія написана низхідним методом.

```
CL-USER> (DEFUN SUM (L)
(COND ((NULL L) 0)
((Numberp L) L)
(T (+ (SUM (CAR L)) (SUM (CDR L))))))
SUM
```

```
CL-USER> (SUM '( (1 2) (3 4) 2) )
12
CL-USER> (SUM '( (1 2 (3 4) 2) )
12
```

Якщо не досягається гранична умова і поточний елемент не число, то працює рекурсивна умова. Обчислюється значення першого доданку - знаходиться сума першого елемента списку за функцією (SUM (CAR L)). В нашому прикладі сума дорівнює 1.

Далі шукається значення другого доданку. Для цього викликається (SUM (CDR L)) і всі дії повторюються.

Якщо перший елемент поточного списку число, замість суми повертається число на місце доданку. Якщо перший елемент список, то знаходиться сума цього списку.

Напротязі всього зворотнього процесу змінна S нарощується.

Розглянемо схему роботи паралельної рекурсії для останнього прикладу:

```
1: (Sum L: (1 2 (3 4) 2) )
2: (Sum (Car L): 1 )
   Повертається перший доданок 1
3: (Sum (Cdr L): (2 (3 4) 2) )
4: (Sum (Car L): 2 )
   Повертається перший доданок 2
```

5: (Sum (Cdr L): ((3 4) 2)) ; Обробка вкладеного списку

6: (Sum (Car L): (3 4))

7: (Sum (Car L): 3)

Повертається перший доданок 3

8: (Sum (Cdr L): (4))

9: (Sum (Car L): 4)

Повертається перший доданок 4

10: (Sum (Cdr L): NIL)

Повертається другий доданок 0

Повертається сума (+ 4 0)

Повертається сума (+ 3 4)

Повертається сума вкладеного списку, як перший доданок 7

12: (Sum (Cdr L): (2))

13: (Sum (Car L): 2)

Повертається перший доданок 2

14: : (Sum (Cdr L): ())

Повертається другий доданок 0.

Повертається сума (+ 2 0)

Повертається сума (+ 7 2)

Повертається сума (+ 2 9)

Повертається сума (+ 1 11)

12

ТЕМА 11: ЦИКЛИ

11.1 Макроси DO

Макрос DO застосовується для реалізації операцій, що повторюються.

(DO ((Змінна Початкова_Форма Форма з Кроком) ... (Змінна Початкова_Форма Крок))
((Умова_закінчення) Результат) Форма1 ... ФормаN)

ДІЇ МАКРОСУ DO

Перед початком роботи макрос DO створює *свої локальні змінні*. Ці локальні змінні існують тільки в тілі макросу DO.

На першій ітерації для кожної змінної макрос обчислює їх початкові – форми і привласнює змінним одержані значення.

На початку кожної ітерації, крім першої, макрос обчислює «Форми з кроками» змінних. Обчислені значення «Форм з кроками» *привласнюються їх змінним*. Кожна «Форма з кроком» містить свою локальну змінну (не обов'язково ту, для якої форма застосовується як крок). Вибір змінної для форми залежить від логіки задачі. Макрос DO застосовує для «форми з кроком» значення своєї змінної із попередньої ітерації.

Програміст може змінювати локальні змінні макросу також сам.

Перша форма в тілі макросу є умовою закінчення роботи циклу. Коли умова закінчення циклу стане істиною, тоді повертається результат. Інакше обчислюються наступні форми тіла макросу DO.

Приклад 1. Ввести числовий список. Одержати суму елементів списку. Зверніть увагу на те, що робота з локальними змінними повинна відповідати роботі у тілі макросу DO. В нашому прикладі – це відповідність між функціями CAR і CDR.

```
CL-USER> (DEFUN SUM (L) (DO ((S (Car L) (+ (Car L) S)))  
  ((NULL (CDR L)) S)  
  (SETQ L (CDR L))))  
SUM
```

```
CL-USER> (SUM '(5 2 3))
```

```
10
```

У прикладі є два типи локальних змінних: формальний параметр L, який діє у тілі функції; локальний параметр S, що працює тільки в макросі DO. Перед першою ітерацією обчислюється початкова форма (CAR L) і змінній S привласнюється 5. Починаючи з другої ітерації обчислюється «форма з кроком» (+ (CAR L) S), де змінна S береться з попередньої ітерації, а L – має поточне значення. Кроком являється (CAR L).

Нижче показана робота макросу DO:

Ітерація 1: L= (5 2 3): Початкове значення (CAR L) = 5; S=5.

Ітерація 2: L= (2 3) S = 5; Крок = 2; S = 2+5.

Ітерація 3: L= (3) S = 7; Крок = 3; S = 3+7.

L=(3) ознака кінця S=10.

У виразі ((NULL (CDR L)) S) дві дужки підряд означають неявну форму COND.

Макрос DO* Працює як DO, але крок додається до значення змінної взятої з поточної ітерації.

Приклад2: Порівняємо роботу DO і DO* на одному прикладі.

```
CL-USER> (DO ( (N1 1 (1+ N1)) (N2 0 (1+ N1))) )
```

```
((= N2 3) N1)
```

```
(PRINC N1) (PRINT N2))
```

```
1 0
```

```
2 2
```

```
3 3
```

```
3
```

```
CL-USER> (DO* ( (N1 1 (1+ N1)) (N2 0 (1+ N1))) )
```

```
((= N2 3) N1)
```

```
(PRINC N1) (PRINT N2))
```

```
1 0
```

```
2 3
```

```
3
```

Крім вказаного загального макросу DO існують спеціалізовані макроси DOTIMES і DOLIST.

DOTIMES

Макрос *DOTIMES*: у загальному вигляді подається:

(DOTIMES (змінна форма результат) форма1 форма2 ...формаN)

Змінна автоматично нарощується від 0 до вказаного цілого значення форми. Якщо не вказан результат, то макрос завжди повертає NIL. Цикли можна вкладувати.

У тілі макросу можна передбачити передчасний вихід при певній умові.

Приклад 1: Вивести на екран таблицю множення.

```
1*1 1*2 1*3 1*4 1*5 1*6 1*7 1*8 1*9 1*10
2*1 2*2 2*3 2*4 2*5 2*6 2*7 2*8 2*9 2*10
3*1 3*2 3*3 3*4 3*5 3*6 3*7 3*8 3*9 3*10
4*1 4*2 4*3 4*4 4*5 4*6 4*7 4*8 4*9 4*10
5*1 5*2 5*3 5*4 5*5 5*6 5*7 5*8 5*9 5*10
6*1 6*2 6*3 6*4 6*5 6*6 6*7 6*8 6*9 6*10
7*1 7*2 7*3 7*4 7*5 7*6 7*7 7*8 7*9 7*10
8*1 8*2 8*3 8*4 8*5 8*6 8*7 8*8 8*9 8*10
9*1 9*2 9*3 9*4 9*5 9*6 9*7 9*8 9*9 9*10
10*1 10*2 10*3 10*4 10*5 10*6 10*7 10*8 10*9 10*10
```

```
CL-USER> (DOTIMES (X 10)
```

```
(DOTIMES (Y 10)
```

```
(PRINC (+ 1 X )) (PRINC *) (PRINC (+ 1 Y))
```

```
(PRINC '\ ) ) (TERPRI))
```

Функція PRINC залишає курсор після виведених символів.
Функція TERPRI переводить курсор в початок наступного рядку.

DOLIST

Макрос *DOLIST*: у загальному вигляді подається:

(DOLIST (змінна список результат) форма1 форма2 ...формаN)

Макрос послідовно привласнює змінній елементи списку, поки не скінчиться список, і виконує дії, що вказані у формах. Макрос повертає NIL, якщо не вказан результат. У тілі макросу можна передбачити передчасний вихід при певній умові.

Приклад 1: Знайти суму непарних елементів числового списку. Дії виконувати до першого парного числа.

```
CL-USER> (DEFUN SUM_NP(L &KEY (S 0))
  (DOLIST (X L S)
    (IF (EVENP X)(RETURN S))
    (INCF S X)))
SUM_NP
```

```
CL-USER> (SUM_NP '(11 3 5 4 7))
```

```
19
```

```
CL-USER> (SUM_NP '(11 3 5 9 7))
```

```
35
```

11.2 Макрос LOOP

Макрос LOOP застосовують для виконання форм в циклі. Макрос LOOP – буває двох типів простий і розширений.

ПРОСТИЙ ТИП МАКРОСУ LOOP

Макрос LOOP виконує форми в своєму тілі послідовно зліва направо. Цикл закінчує свою роботу за формою з умовою. Для виходу з циклу застосовують спеціальну форму IF і функцію RETURN або RETURN-FROM.

Приклад 1. Знайти максимальний елемент списку.

```
CL-USER> (DEFUN MAX1 (L &OPTIONAL ( M (CAR L)) )
  (LOOP (IF (NULL L) (RETURN M))
    (IF (> (CAR L) M) (SETQ M (CAR L))
      (POP L))))
```

```
MAX1
```

```
CL-USER> (MAX1 '(12 45 6 34))
```

```
45
```

Функція має один обов'язковий параметр *L* і один необов'язковий параметр *M*. Для знаходження максимального елементу числового списку по замовчання змінній *M* привласнено (*CAR L*). Вихід з циклу виконується за умовою список порожній.

Існують різні способи знаходження максимального числа. Стандартна функція *MAX* може мати довільну кількість аргументів, але вона не працює із списками.

```
CL-USER> (MAX 34 23 99 23 67)
99
```

Для знаходження максимального елементу списку можна використати функцію *REDUCE*, застосувавши до списку функцію *MAX*.

```
CL-USER> (REDUCE 'MAX '(34 23 99 67))
99
```

РОЗШИРЕНИЙ LOOP

Розширений макрос *Loop* являє собою «мову в мові». Він містить в собі різні форми використання циклів.

Для цього макрос має спеціальні ключові слова для виконання дій:

- *minimize* (знайти мінімум);
- *maximize* (знайти максимум);
- *collecting* (зібрати);
- *counting* (підрахувати);
- *summing* (підсумувати);

- and (логічне і);
- repeat (повторити);
- finally (наприкінці);
- across (через, скрізь).

Межі дій:

- from (від);
- for (для);
- then (тоді);
- to (до);
- in (в).

Приклад 1. Підрахувати кількість голосних у тексті.

```
CL-USER> (LOOP FOR X ACROSS "COMMON LISP"
COUNTING (FIND X "AOUEI"))
```

3

Для усіх X з рядку підрахувати скільки разів X знайдеться серед голосних. Функція FIND застосовується для послідовностей.

Приклад 2. Знайти суму елементів списку.

```
CL-USER> (DEFUN SUM (L)
  (LOOP FOR I FROM 0 TO (- (LENGTH L) 1)
    SUMMING (NTH I L)))
```

SUM

```
CL-USER> (SUM '(23 17 30))
```

70

Для і від 0 до довжини списку мінус 1 просумувати всі елементи списку.

ЛІТЕРАТУРА

Основна

1. Peter Seibel "Practical Common Lisp" APRES. 2006г.
2. "Руководство по GNU Emacs" Тринадцатая редакция, обновлено для Emacs версии 20.7 Ричард Столмен, Издано: АНО «ИЛКиРЛ" 1999г.
3. Хювёнен Э., Сеппянен И. Мир Лиспа. Т. 1. Введение в мова Лисп и функциональное програмування.
4. Хювёнен Э., Сеппянен И. Мир Лиспа. Т2. Методы и системы программирования.

Додаткова

- 1 Хендерсон П. «Функциональное программирование. Использование и реализация». - М.: Мир. 1983. - 349 с.
- 2 А. Филд, П. Харрисон «Функциональное программирование»: Пер. с англ. - М.: Мир. 1993.-637с.