# Вступ до функціонального програмування. Методичні вказівки з виконання лабораторних робіт

# Лабораторна робота 1. Обробка списків з використанням базових функцій

**Мета лабораторної роботи**: ознайомитись із базовими типами даних та функціями Common Lisp, отримати практичні навички роботи зі списками.

Опис базових типів даних, базових функцій, а також особливостей роботи з REPL та внутрішньої організації списків наведено в розділах 2-5 навчального посібника.

**Реченець** виконання роботи (aka **дедлайн**) визначається викладачем під час видачі завдання на лабораторну роботу.

## 1.1 Завдання

Завдання складається з двох частин: загальної для всіх та завдання за варіантом.

### 1.1.1 Загальне завдання

Загальне завдання складається з кількох пунктів, наведених нижче. Для зручності перевірки результатів, у звіті перед фрагментом лістингу кожного з пунктів загального завдання варто додати коментар з позначенням пункту. Наприклад:

```
;; Пункт 1
CL-USER> ...

;; Пункт 2
CL-USER> ...
```

Або ж розділити лістинг по пунктам і додати підзаголовки (див. п. 1.2 для детальнішого пояснення).

Завдання:

1. Створіть список з п'яти елементів, використовуючи функції `LIST` і `CONS`. Форма створення списку має бути одна — використання `SET` чи `SETQ` (або інших допоміжних форм) для збереження проміжних значень не допускається. Загальна кількість елементів (включно з підсписками та їх елементами) не має перевищувати 10-12 шт. (дуже великий список робити не потрібно). Збережіть створений список у якусь змінну з `SET` або `SETQ`. Список має містити (напряму або у підсписках):
    - хоча б один символ
    - хоча б одне число
    - хоча б один не пустий підсписок

- хоча б один пустий підсписок

2. Отримайте голову списку.
3. Отримайте хвіст списку.
4. Отримайте третій елемент списку.
5. Отримайте останній елемент списку.
6. Використайте предикати `ATOM` та `LISTP` на різних елементах списку (по 2-3 приклади для кожної функції).
7. Використайте на елементах списку 2-3 інших предикати з розглянутих у розділі 4 навчального посібника.
8. Об'єднайте створений список з одним із його непустих підсписків. Для цього використайте функцію `APPEND`.

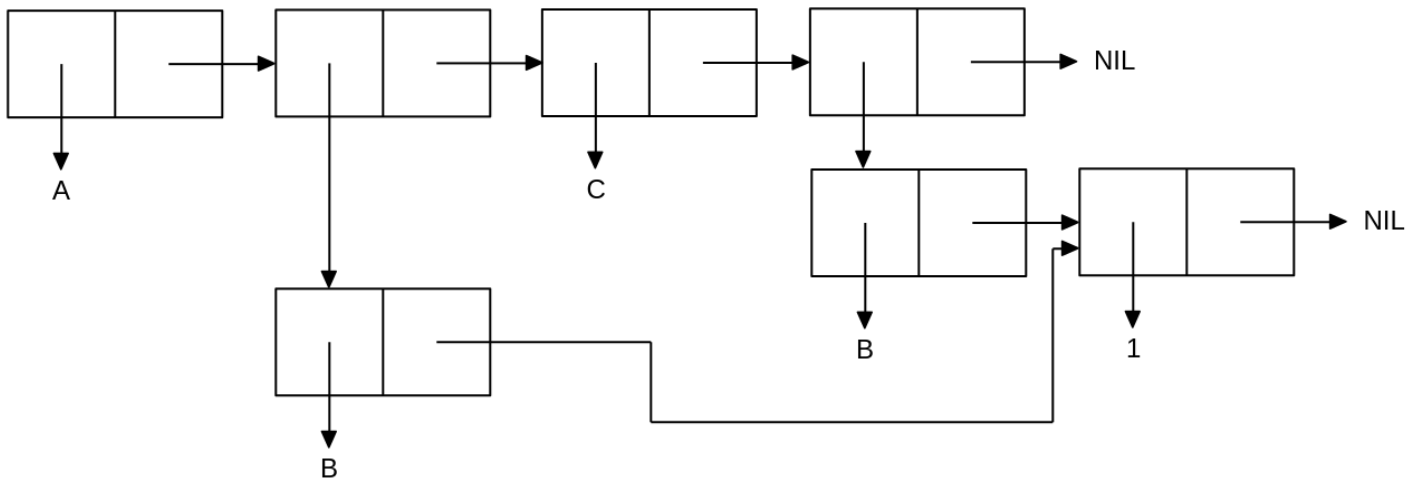
### 1.1.2 Завдання за варіантом

Створіть список, що відповідає структурі списку, наведеній на рисунку (за варіантом). Для цього **допускається використання не більше двох форм конструювання списку на "верхньому рівні"**. Але аргументами цих форм можуть бути результати інших викликів форм конструювання списків. Наприклад:

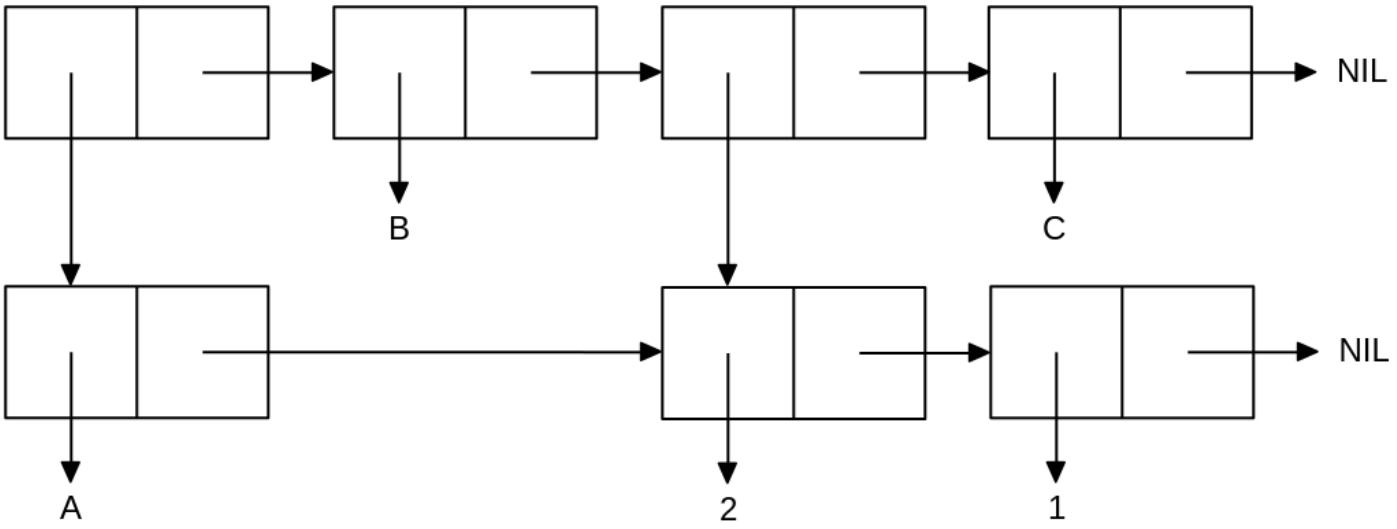
```
(let ((x (list 1 2)))  ; (list ...) - перша форма
  ;; далі - друга форма (list ...), що містить підсписки
  (list x (list 3) 4 (cons (cdr x) '(6))))
```

Номер варіанту обирається як номер у списку групи, який надсилає викладач на початку семестру (на випадок, якщо протягом семестру стануться зміни в складі групи), за модулем 8: 1 -> 1, 2 -> 2, ..., 9 -> 1, 10 -> 2, ...

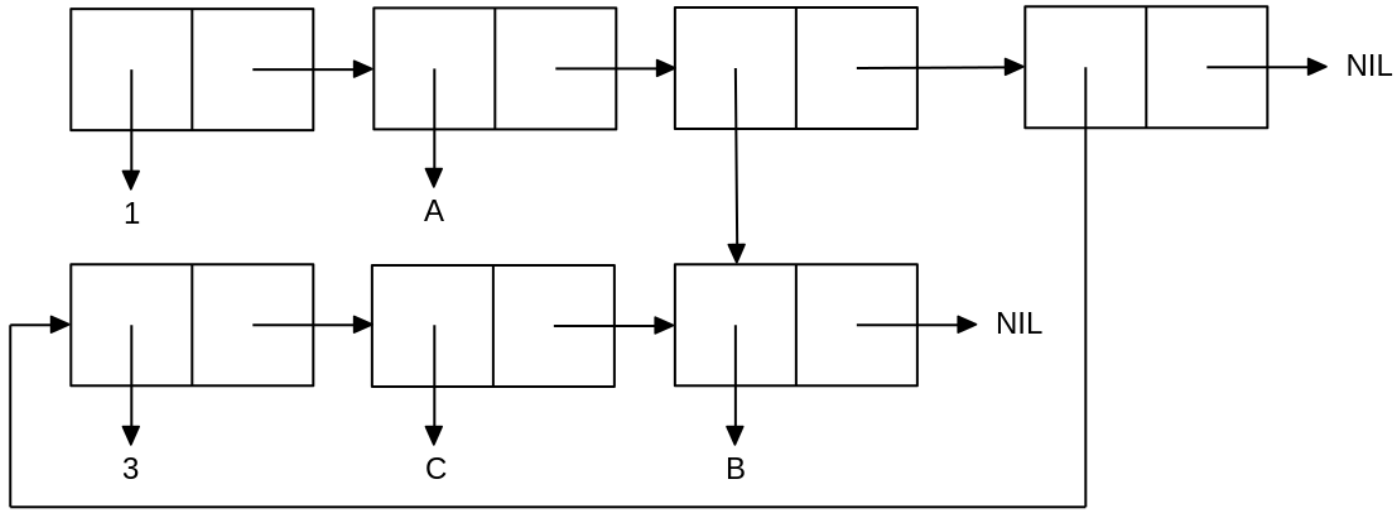
**Варіант 1**



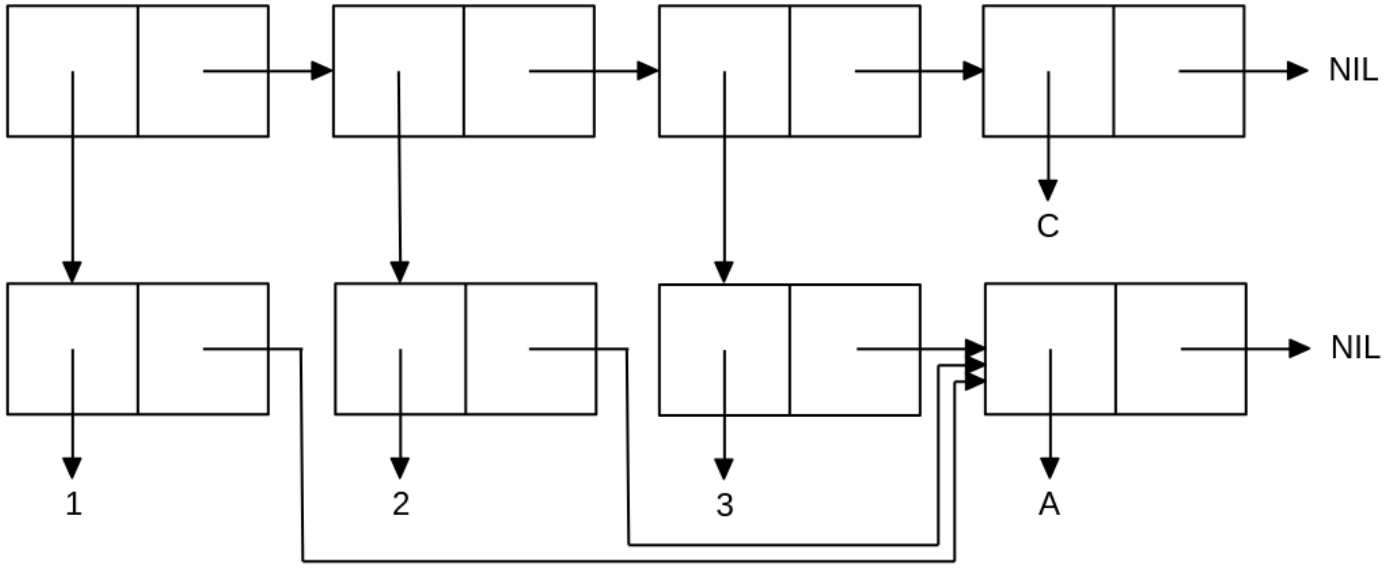

**Варіант 2**



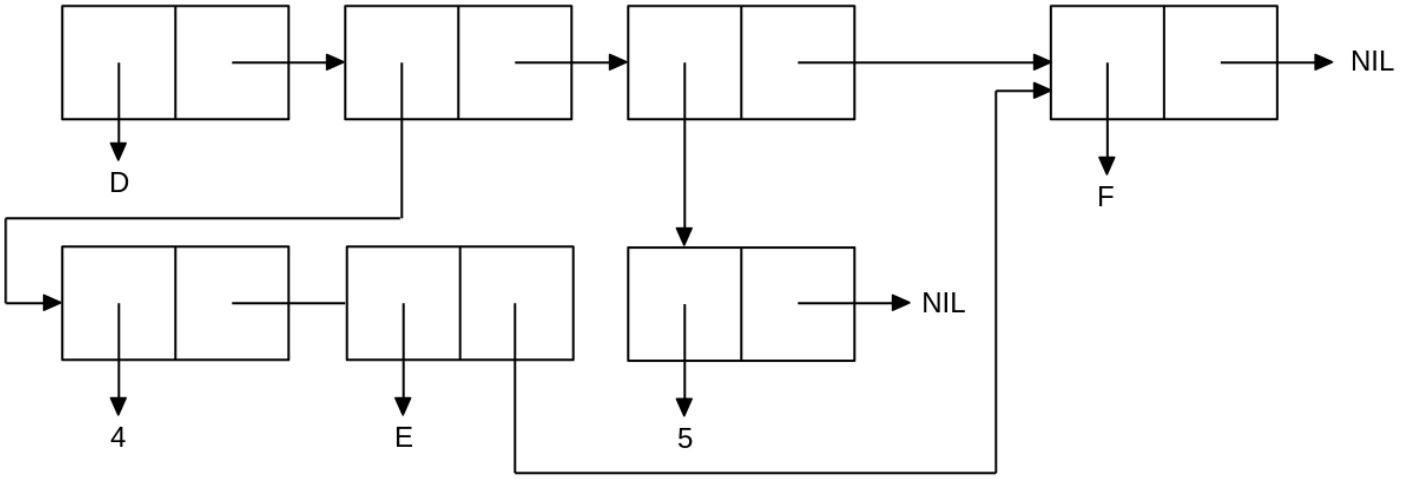

**Варіант 3**



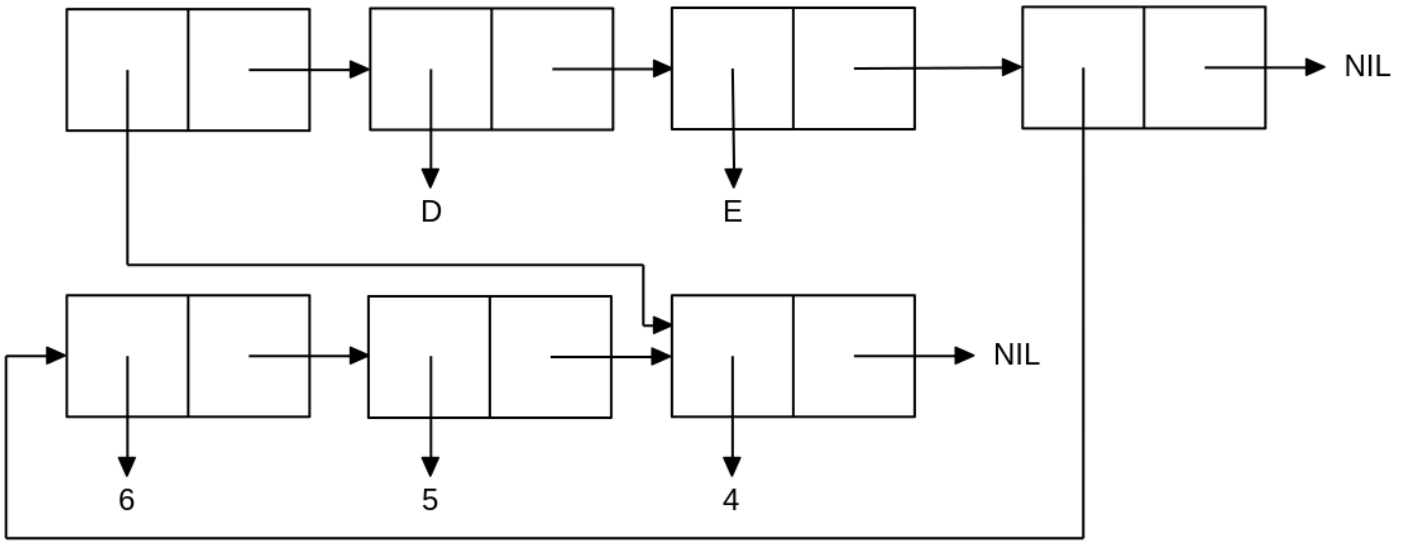

**Варіант 4**



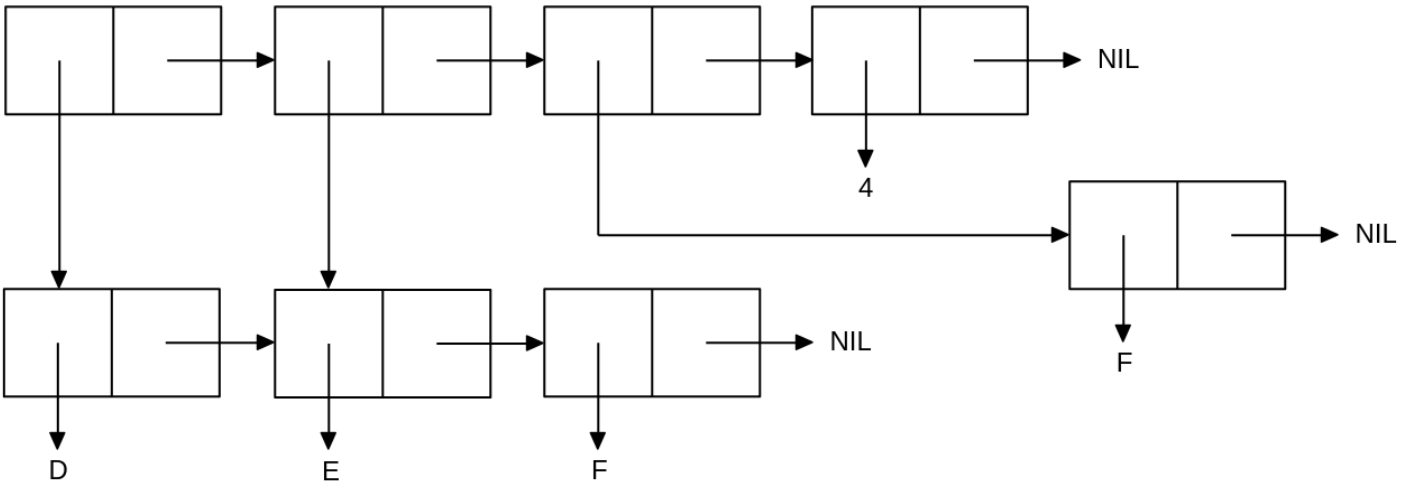

**Варіант 5**



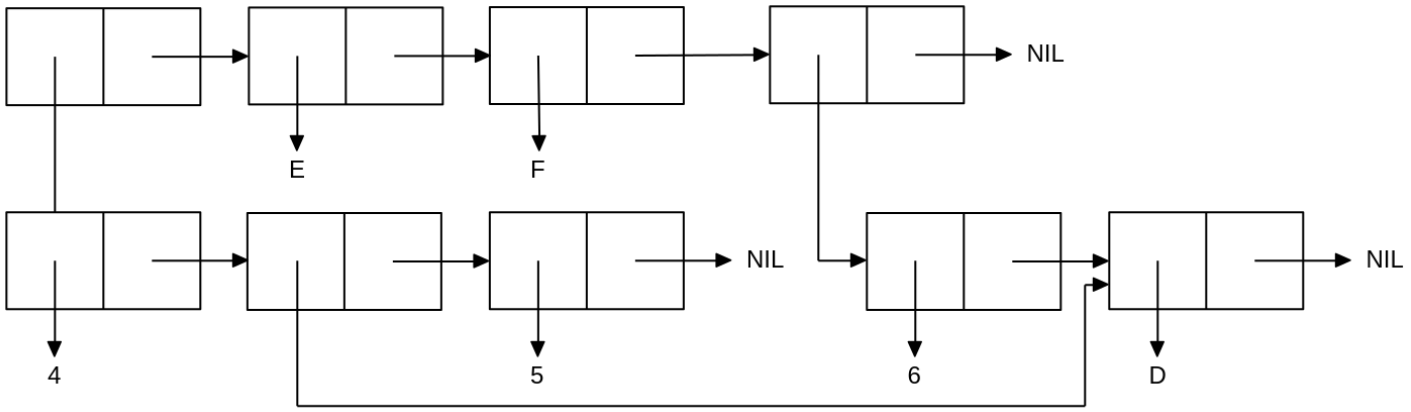

**Варіант 6**




**Варіант 7**




**Варіант 8**




## 1.2 Формат звіту та захисту

В результаті виконання лабораторної роботи необхідно підготувати звіт у форматі Markdown. Файл звіту (з розширенням `.md`) та файл із зображенням структури списку за варіантом (що використовуватиметься у звіті) або посилання на файл звіту у репозиторії (GitHub, Bitbucket, тощо.) необхідно надіслати викладачу на електронну пошту.

Звіт має містити титульні заголовки (див. шаблон), лістинг викликів в REPL з виконанням загального завдання, номер варіанту та лістинг виконання завдання за варіантом.

Шаблон звіту такий:

````
<p align="center"><b>МОНУ НТУУ КПІ ім. Ігоря Сікорського ФПМ СПіСКС</b></p>

<p align="center">
<b>Звіт з лабораторної роботи 1</b><br/>
"Обробка списків з використанням базових функцій"<br/>
дисципліни "Вступ до функціонального програмування"
</p>

<p align="right"><b>Студент(-ка)</b>: Прізвище Ім'я По-батькові група</p>
<p align="right"><b>Рік</b>: рік</p>

## Загальне завдання

<!--лістинг пунктів загального завдання можна навести в одному блоці коду із ком
які позначають початок виконання окремих пунктів, або ж розділити весь лістинг н
блоки коду і додати для них підзаголовки (напр. ### Пункт 1)-->

```lisp
<Тут має бути лістинг (текст) виконання загального завдання. Оскільки
 завдання вимагає лише виконання простих команд,  достатньо буде додати
 лістинг у вигляду тексту виконання команд і отриманого результату з REPL>
```

## Варіант <номер варіанту>

<p align="center">
<img src="lab-1-variant.png">
</p>

```lisp
````
````

```
<тут має бути лістинг (текст) виконання завдання за варіантом>
```
```

*Примітка*: за потреби опис правил форматування Markdown можна знайти за посиланнями:

- https://markdown.rozh2sch.org.ua/
- https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax

На захисті лабораторної роботи необхідно продемонструвати роботу окремих форм, відповісти на питання викладача по виконаній роботі та теорії.

### 1.3 Контрольні питання

1. Схарактеризуйте базові типи Common Lisp.
2. Що таке REPL?
3. Які є базові функції для роботи зі списками?
4. Які є базові функції для виконання арифметичних операцій?
5. Які є предикати для порівняння об'єктів в Common Lisp?
6. Як в Common Lisp реалізовані логічні операції?
7. Що таке список, спискова комірка, точкова пара?
8. Що таке псевдофункія? Наведіть приклади псевдофункцій.
9. В чому різниця між функцією, макросом та спеціальним оператором (формою)?
10. Наведіть приклади спеціальних операторів.
11. В чому різниця між конструктивним та руйнівним підходами до обробки списків?
12. Як можна відмінити обчислення або ж "виконати" форму?

# Лабораторна робота 2. Рекурсія

**Мета лабораторної роботи**: навчитись застосовувати рекурсію при реалізації простих програм мовою Common Lisp.

Різні види рекурсії розглядаються в розділі 10 навчального посібника. Визначення функції та форми для керування потоком виконання розглядаються в розділах 6 і 9 навчального посібника. Зв'язування змінних та контекст дії зв'язування розглядається в розділі 7 посібника.

**Реченець** виконання роботи (aka **дедлайн**) визначається викладачем під час видачі завдання на лабораторну роботу.
```

## 2.1 Завдання

Реалізуйте дві рекурсивні функції, що виконують деякі дії з вхідним(и) списком(-ами), за можливості/необхідності використовуючи різні види рекурсії. Функції, які необхідно реалізувати, задаються варіантом (п. 2.1.1). Вимоги до функцій:

1. Зміна списку згідно із завданням має відбуватись за рахунок конструювання нового списку, а не зміни наявного (вхідного).
2. Не допускається використання функцій вищого порядку чи стандартних функцій для роботи зі списками, що не наведені в четвертому розділі навчального посібника.
3. Реалізована функція не має бути функцією вищого порядку, тобто приймати функції в якості аргументів.
4. Не допускається використання псевдофункцій (деструктивного підходу).
5. Не допускається використання циклів.

Кожна реалізована функція має бути протестована для різних тестових наборів. Тести мають бути оформленні у вигляді модульних тестів (див. п. 2.3).

*Додатковий бал* за лабораторну роботу можна отримати в разі виконання всіх наступних умов:

- робота виконана до дедлайну (включно з датою дедлайну)
- крім основних реалізацій функцій за варіантом, також реалізовано додатковий варіант однієї чи обох функцій, який працюватиме швидше за основну реалізацію, не порушуючи при цьому перші три вимоги до основної реалізації (вимоги 4 і 5 можуть бути порушені), за виключенням того, що в разі необхідності можна також використати стандартну функцію `copy-list`

### 2.1.1 Варіанти завдань

Номер варіанту обирається як номер у списку групи, який надсилає викладач на початку семестру, за модулем 15: 1 -> 1, 2 -> 2, ..., 16 -> 1, 17 -> 2, ...

#### Варіант 1

1. Написати функцію `reverse-and-nest-head`, яка обертає вхідний список та утворює вкладену структуру з підсписків з його елементами, починаючи з голови:

```
CL-USER> (reverse-and-nest-head '(a b c))
(((C) B) A)
```

2. Написати функцію `duplicate-elements`, що дублює елементи вхідного списку задану кількість разів:

```
CL-USER> (duplicate-elements '(a b c) 3)
(A A A B B B C C C)
```

**Варіант 2**

1. Написати функцію `remove-seconds-and-thirds`, яка видаляє зі списку кожен другий і третій елементи:

```
CL-USER> (remove-seconds-and-thirds '(a b c d e f g))
(A D G)
```

2. Написати функцію `list-set-intersection`, яка визначає перетин двох множин, заданих списками атомів:

```
CL-USER> (list-set-intersection '(1 2 3 4) '(3 4 5 6))
(3 4)  ; порядок може відрізнятись
```

**Варіант 3**

1. Написати функцію `group-pairs`, яка групує послідовні пари елементів у списки:

```
CL-USER> (group-pairs '(a b c d e f g))
((A B) (C D) (E F) (G))
```

2. Написати функцію `list-set-union`, яка визначає об'єднання двох множин, заданих списками атомів:

```
CL-USER> (list-set-union '(1 2 3) '(2 3 4))
(1 2 3 4)  ; порядок може відрізнятись
```

**Варіант 4**

1. Написати функцію `remove-seconds-and-reverse`, яка видаляє зі списку кожен другий елемент і обертає результат у зворотному порядку:

```
CL-USER> (remove-seconds-and-reverse '(1 2 a b 3 4 d))
(D 3 A 1)
```

2. Написати функцію `list-set-difference`, яка визначає різницю двох множин, заданих списками атомів:

```
CL-USER> (list-set-difference '(1 2 3 4) '(3 4 5 6))
(1 2)  ; порядок може відрізнятись
```

**Варіант 5**

1. Написати функцію `remove-seconds`, яка видаляє зі списку кожен другий елемент:

```
CL-USER> (remove-seconds '(1 2 a b 3 4 d))
(1 A 3 D)
```

2. Написати функцію `list-set-symmetric-difference`, яка визначає симетричну різницю двох множин, заданих списками атомів (тобто, множину елементів, що не входять до обох множин):

```
CL-USER> (list-set-symmetric-difference '(1 2 3 4) '(3 4 5 6))
(1 2 5 6)  ; порядок може відрізнятись
```

**Варіант 6**

1. Написати функцію `merge-lists-spinning-pairs`, яка групує відповідні елементи двох списків, почергово змінюючи їх взаємне розташування в групі:

```
CL-USER> (merge-lists-spinning-pairs '(1 2 3 4 5) '(a b c d))
((1 A) (B 2) (3 C) (D 4) (5))
```

2. Написати предикат `list-set-intersect-p`, який визначає чи перетинаються дві множини, задані списками атомів, чи ні:

```
CL-USER> (list-set-intersect-p '(1 2 3) '(4 5 6))
NIL
CL-USER> (list-set-intersect-p '(1 2 3) '(3 4 5))
T
```

**Варіант 7**

1. Написати функцію `merge-lists-pairs`, яка групує відповідні елементи двох списків:

```
CL-USER> (merge-lists-pairs '(1 2 3 4 5) '(a b c d))
((1 A) (2 B) (3 C) (4 D) (5))
```

2. Написати предикат `sublist-after-p`, який перевіряє, чи знаходиться після визначеного елемента списку атомів визначена послідовність елементів (список):

```
CL-USER> (sublost-after-p '(1 a 2 b 3 c 4 d) 'b '(3 c))
T
CL-USER> (sublost-after-p '(1 a 2 b 3 c 4 d) 'b '(3 c d))
NIL
```

**Варіант 8**

1. Написати функцію `reverse-and-nest-tail`, яка обертає вхідний список та утворює вкладену структуру з підсписків з його елементами, починаючи з хвоста:

```
CL-USER> (reverse-and-nest-tail '(a b c))
(C (B (A)))
```

2. Написати функцію `compress-list`, яка заміщає сукупності послідовно розташованих однакових елементів списку двоелементними списками виду (кількість-повторень елемент):

```
CL-USER> (compress-list '(1 a a 3 3 3 b))
((1 1) (2 A) (3 3) (1 B))
```

**Варіант 9**

1. Написати функцію `remove-even-pairs`, яка видаляє зі списку кожен третій та четвертий елементи:

```
CL-USER> (remove-even-pairs '(1 a 2 b 3 c 4))
(1 A 3 C)
```

2. Написати функцію `find-deepest-list`, яка поверне "найглибший" підсписок з вхідного списку:

```
CL-USER> (find-deepest-list '(1 2 3 4 5))
(1 2 3 4 5)
CL-USER> (find-deepest-list '(1 (2 (3) 4) 5))
(3)
```

**Варіант 10**

1. Написати функцію `group-triples`, яка групує послідовні трійки елементів у списки:

```
CL-USER> (group-triples '(a b c d e f g))
((A B C) (D E F) (G))
```

2. Написати функцію `list-set-intersection-3`, яка визначає перетин трьох множин, заданих списками атомів:

```
CL-USER> (list-set-intersection-3 '(1 2 3 4) '(3 4 5 6) '(1 3 4 6))
(3 4)  ; порядок може відрізнятись
```

**Варіант 11**

1. Написати функцію `remove-thirds`, яка видаляє зі списку кожен третій елемент:

```
CL-USER> (remove-thirds '(a b c d e f g))
(A B D E G)
```

2. Написати функцію `list-set-union-3`, яка визначає об'єднання трьох множин, заданих списками атомів:

```
CL-USER> (list-set-union-3 '(1 2 3) '(2 3 4) '(nil t))
(1 2 3 4 NIL T)  ; порядок може відрізнятись
```

**Варіант 12**

1. Написати функцію `remove-thirds-and-reverse`, яка видаляє зі списку кожен третій елемент і обертає результат у зворотному порядку:

```
CL-USER> (remove-thirds-and-reverse '(a b c d e f g))
(G E D B A)
```

2. Написати функцію `list-set-difference-3`, яка визначає різницю трьох множин, заданих списками атомів:

```
CL-USER> (list-set-difference '(1 2 3 4) '(4 5 6) '(2 5 7))
(1 3)  ; порядок може відрізнятись
```

**Варіант 13**

1. Написати функцію `remove-even-triples`, яка видалить парні трійки послідовних елементів зі списку:

```
CL-USER> (remove-even-triples '(a b c d e f g h))
(A B C G H)
```

2. Написати функцію `decompress-list`, яка "розпакує" зі списку пар виду (кількість-повторень елемент) послідовність елементів визначеної кількості:

```
CL-USER> (decompress-list '((1 1) (2 a) (3 3) (1 4)))
(1 A A 3 3 3 4)
```

**Варіант 14**

1. Написати функцію `group-pairs-and-reverse`, яка групує послідовні пари елементів у списки і повертає обернений результат:

```
CL-USER> (group-pairs-and-reverse '(a b c d e f g))
((G) (E F) (C D) (A B))
```

2. Написати функцію `cyclic-shift-left`, яка виконує циклічний зсув елементів списку ліворуч на задане число позицій:

```
CL-USER> (cyclic-shift-left '(1 2 3 4 5 6) 2)
(3 4 5 6 1 2)
```

**Варіант 15**

1. Написати функцію `spread-values`, яка заміняє `nil` в списку на попередній не-`nil` елемент:

```
CL-USER> (spread-values ‘(nil 1 2 nil 3 nil nil 4 5))
(NIL 1 2 2 3 3 3 4 5)
```

2. Написати функцію `delete-duplicates`, яка видаляє всі послідовні дублікати тих елементів з вхідного списку атомів, послідовних дублікатів яких більше за задане число:

```
CL-USER> (delete-duplicates '(1 1 2 3 3 3 2 2 a a a b) 3)
(1 1 2 3 2 2 A B)
```

## 2.2 Формат звіту та захисту

В результаті виконання лабораторної роботи необхідно підготувати звіт у форматі Markdown. Файл звіту (з розширенням `.md`) та файл(и) з кодом (з розширенням `.lsp`

або `.lisp`), або посилання на файл звіту і файл(и) з кодом у репозиторії (GitHub, Bitbucket, тощо.) необхідно надіслати викладачу на електронну пошту.

Звіт має містити (див. шаблон):

- титульні заголовки
- загальне завдання
- номер варіанту
- завдання за варіантом
- лістинг реалізованих завдань
- лістинг реалізації тестових наборів
- результат виконання тестових наборів

Шаблон звіту такий:

```
<p align="center"><b>МОНУ НТУУ КПІ ім. Ігоря Сікорського ФПМ СПіСКС</b></p>

<p align="center">
<b>Звіт з лабораторної роботи 2</b><br/>
"Рекурсія"<br/>
дисципліни "Вступ до функціонального програмування"
</p>

<p align="right"><b>Студент(-ка)</b>: Прізвище Ім'я По-батькові група</p>
<p align="right"><b>Рік</b>: рік</p>

## Загальне завдання

<!-- Зазначається загальне завдання -->

## Варіант <номер варіанту>

<!-- Зазначається завдання за варіантом -->

## Лістинг функції <назва першої функції>

```lisp
<Лістинг реалізації першої функції>
```

### Тестові набори

```lisp
<Лістинг реалізації тестових наборів першої функції>
```

### Тестування

```lisp
<Виклик і результат виконання тестів першої функції>
```
```

```
```

## Лістинг функції <назва другої функції>

```lisp
<Лістинг реалізації другої функції>
```

### Тестові набори

```lisp
<Лістинг реалізації тестових наборів другої функції>
```

### Тестування

```lisp
<Виклик і результат виконання тестів другої функції>
```

На захисті лабораторної роботи необхідно продемонструвати роботу реалізованих функцій для різних тестових наборів, а також відповісти на питання викладача по виконаній роботі та теорії.

## 2.3 Просте модульне тестування

Для виконання тестування розроблених функцій можна написати одну функцію, що виконує перевірку фактичного результату з очікуваним і виводить повідомлення про те, чи пройшла перевірка, чи ні. Наприклад:

```
;;; Test function `my-reverse'

(defun check-my-reverse (name input expected)
  "Execute `my-reverse' on `input', compare result with `expected' and print
   comparison status"
  (format t "~:[FAILED~;passed~]... ~a~%"
          (equal (my-reverse input) expected)
          name))
```

Функція `format` в залежності від значення першого аргументу форматованого рядка (третій аргумент `format`) виводить "passed" або "FAILED", а також ім'я тесту, задане другим аргументом для форматованого рядка.

Тепер можна написати тестові набори:

```
(defun test-my-reverse ()
  (check-my-reverse "test 1" '(1 2 3) '(3 2 1))
  (check-my-reverse "test 2" nil nil)
  (check-my-reverse "test 3" '(1 (2) (3 (4))) '((3 (4)) (2) 1)))
```

І виконати тести:

```
CL-USER> (test-my-reverse)
```

```
passed test 1
passed test 2
passed test 3
NIL
```

### 2.4 Контрольні питання

1. Що таке рекурсія?
2. Які є види рекурсії? Опишіть їх.
3. Як визначаються функції в Common Lisp?
4. Які є форми розгалуження виконання в Common Lisp?
5. Яка область видимості змінних в Common Lisp?
6. Що таке хвостова рекурсія? Які її особливості?

# Додаток А. Поради щодо стилю написання коду мовою Common Lisp

Тут наведено деякі поради щодо оформлення коду мовою Common Lisp. Більш детальні описи конвенцій оформлення коду можна знайти за посиланнями:

- https://lisp-lang.org/style-guide/
- https://google.github.io/styleguide/lispguide.xml

## А.1 Стиль написання коду

### А.1.1 Пробіли між символьними виразами

При написанні програм в Common LIsp, слід дотримуватись наступних правил додавання пробілів і переносів рядка:

- після відкриваючої дужки перед першим елементом списку та після останнього елементу перед закриваючою дужкою пробіл не ставиться;
- між елементами списку ставиться по 1-му пробілу, або переносу рядка

Приклад:

```
(list (format nil "~A" arg) (1+ num) (- 10 k))
```

### А.1.2 Відступи вкладених форм

Вкладені форми (або елементи списку), що записуються у новому рядку, мають мати певний відступ відносно батьківської форми (початку списку, його відкриваючої дужки). Мінімальний відступ — 1 пробіл, але типова кількість відступів залежить від конкретної форми. Наприклад, для `defun`, `let` та ін. "тіло" форми має відступ у два символи

відносно початкового відступу батьківської форми.

Зазвичай кількість відступів регулюється Emacs-ом за допомогою комбінації C-M-q (Ctrl+Alt+q), застосованій на відкриваючій дужці форми (виправлення відбувається лише всередині форми, до якої ця комбінація застосована). Це правило стосується всіх вкладених форм.

Наприклад:

```
(defun foo (arg k)
  "Some comment about what this function does"
  (if (eql arg k)
      (1+ arg)
      (case k
        (1 "ok")
        ((2 3)
         "better")
        (otherwise "bad"))))
```

### A.1.3 Перенесення дужок, гілок та вкладених форм

Переносити дужки не потрібно. Коли реалізація деякої форми дописана і в кінці є кілька закриваючих дужок, вони мають бути розміщені одна за одною без переносів і пробілів:

```
(when a
  (when b
    (when c
      (+ a b c)))
```

Гілки форм керування потоком виконання та вкладені форми слід переносити на наступний рядок, якщо всі вони не поміщаються в один.

Нема стандарту “максимальної довжини рядка”, й взагалі його довжина не є обмеженою. Але краще писати код так, щоб довжина його рядків не перевищувала 80-100 символів — так, щоб ширина рядка не перевищували "половину екрана". Якщо ж форми доведеться переносити, то переносити потрібно всі, окрім однієї-двох перших. Якщо частину форм переносити, а частину — ні, то код може виглядати гірше або ж менш очевидно, бо деякі елементи "загубляться":

```
(if (eql (do-some-stuff (+ 2 3) (list 5)) (do-another-stuff (- 3 2))) (list 'good)
    (list 'bad))
```

У прикладі вище може здатись, що `(list 'bad)` — це T-гілка оператора `if`. Проте насправді це NIL-гілка. А T-гілка знаходиться рядком вище — `(list 'good)`. Тож варто T-гілку також перенести на новий рядок:

```
;;; don't do this:
(if cond t-branch
    nil-branch)

;;; do this (when needed)
(if cond
```

```
    t-branch
    nil-branch)
```

`IF` може бути записаний в один рядок у випадку, якщо його загальна довжина невелика. Наприклад:

```
(if (eql x 1) 0 (error "X != 1"))
```

Якщо `IF` має дві гілки і перша перенесена на новий рядок рядок після умови, тоді й другу теж треба перенести. Тобто ось так писати погано:

```
(if cond
    t-branch nil-branch)  ; bad: nil-branch should be moved to new line
```

Крім того, не потрібно робити кілька перенесень. Достатньо одного. Тобто, наступний приклад є прикладом того, як робити не слід:

```
(if (eql arg k)
                  ; empty line is bad
    (1+ arg)
                  ; the same
    (case k
      (1 "ok")))
```

### A.1.4 Іменування символів

Інтерпретатор дає змогу писати код в будь-якому регістрі — літери символів будуть приведені у верхній регістр автоматично (звісно, крім спеціального формату запису ( `|a b|` )). Проте загальноприйнято писати програми в нижньому регістрі. Тому не варто писати `(Car ...` , `(aPpEnd ...` , і подібні варіанти з використанням змішаного регістру. Суто верхній регістр зазвичай не використовується при написанні коду.

Символи, що складаються з декількох слів записуються з дефісами між словами. Наприклад: `do-something-interesting` , `my-reverse` , `lab-1` . Це так званий стиль kebab-case.

При написанні програм на Lisp (як, загалом, і інших мовах програмування) слід уникати таких імен змінних, як `l` (маленька літера "L"), `O` (велика літера "o"), оскільки їх досить просто переплутати з одиницею `1` і нулем `0` .

### A.1.5 Визначення функцій

Між окремими визначеннями функцій має бути відступ в один порожній рядок. Також можна їх розділяти за допомогою коментарів, наприклад, розділяючи групи функцій, що належать до різних завдань чи різних логічно пов'язаних груп. Наприклад:

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

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

```
;;; ------------------------------------------------------

(defun do-something (arg)
  ...)
```

При визначенні функції варто уникати занадто великих функцій або функцій, логіка яких неочевидна з коду. Але, якщо, все ж, функція виявилась складнішою, ніж варто, тоді варто додати документаційний коментар, в якому зазначено, що саме робить функція. Наприклад:

```
(defun reverse-list (lst)
  "Reverse input list"  ; just as an example (since logic is obvious
                        ; from the function's name)
  ...)
```