МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

ВИКОРИСТАННЯ РЕКУРСІЇ У ФУНКЦІОНАЛЬНОМУ ПРОГРАМУВАННІ

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторної роботи № 2
з курсу “Декларативне програмування” для студентів
спеціальності 121 «Інженерія програмного забезпечення»
(напряму 6.050103 «Програмна інженерія»)

Затверджено
на засіданні кафедри
програмного забезпечення
Протокол № 12 від 08.06.2017

Львів-2017

Використання рекурсії у функціональному програмуванні: Методичні вказівки до лабораторної роботи № 2 з курсу “Декларативне програмування” для студентів спеціальності 121 «Інженерія програмного забезпечення»/ Укл.: Є.В. Левус, Р.Р. Шкраб. – Львів: Видавництво Національного університету “Львівська політехніка”, 2017. – 24 с.

Укладачі Левус Є.В., канд. техн. наук, доц.
Шкраб Р.Р., асистент

Відповідальний за випуск Яковина В.С., завідувач кафедри ПЗ

Рецензенти Тушницький Р.Б., к.т.н., доц., доцент кафедри ПЗ
Марікуца У.Б., к.т.н., доц., доцент кафедри САПР

Вступ

Методичні вказівки містять теоретичні відомості та приклади стосовно рекурсивних обчислень, які використовуються у декларативному програмуванні для організації складних обчислювальних процесів.

Метою виконання лабораторної роботи студентами є освоєння нової техніки мислення, відмінної для традиційного програмування, в той же час поглиблення знань з фундаментальних понять програмування та теорії обчислювальних процесів та структур.

Лабораторна робота полягає у написанні функціональної програми однією з функціональних мов програмування. Зокрема, пропонуються мови Scheme, Lisp, F#. Після здачі лабораторної роботи за комп'ютером студент повинен підготувати звіт.

Тема. Рекурсія як основний спосіб реалізації складних обчислювальних процесів у функціональному програмуванні.
Мета. Навчитися описувати повторювані обчислювальні процеси у формі рекурсивних функцій з допомогою функціональної мови програмування.

Теоретичні відомості.

1. Рекурсивний метод у програмуванні. Рекурсивний метод у програмуванні передбачає розв'язання задачі, ґрунтуючись на властивостях рекурсивності окремих об'єктів. При цьому вихідна задача зводиться до вирішення аналогічних підзадач, які є простішими і відрізняються іншим набором параметрів. При виконанні рекурсивної функції здійснюється багаторазовий перехід від деякого поточного рівня організації алгоритму до нижнього рівня послідовно доти, поки, не буде отримано тривіальне рішення поставленого завдання.

Рекурсивний алгоритм − це алгоритм, в описі якого прямо або непрямо міститься звернення до самого себе. Рекурсивний алгоритм завжди розбиває задачу на частини та класифікується, залежно від того, які функції можна визначити і обчислити з використанням різних форм рекурсії.

Розробці рекурсивних алгоритмів передує рекурсивна тріада – етапи моделювання завдання, на яких визначається набір параметрів і співвідношень між ними. Рекурсивну тріаду складають *параметризація, виділення бази і декомпозиція*.

На етапі *параметризації* з постановки задачі виділяються параметри, які описують вихідні дані. При цьому деякі подальші розробки можуть вимагати введення додаткових параметрів, які не обумовлені в умови. Необхідність в

додаткових параметрах часто виникає також при вирішенні задач оптимізації рекурсивних алгоритмів, в ході яких скорочується їхня часова складність.

Виділення бази рекурсії передбачає перебування в розв'язуваної задачі тривіальних випадків, результат для яких очевидний і не потребує проведення розрахунків. Вірно знайдена база рекурсії забезпечує завершеність рекурсивних звернень.

Декомпозиція – це процес послідовного розбиття задачі на підзадачі двох типів: тих, які ми вирішувати вміємо і тих, які в чомусь аналогічні до вихідної задачі. В останньому випадку кожна з отриманих підзадач повинна бути спрощеним варіантом попередньої задачі. При цьому декомпозицію слід здійснювати так, щоб можна було довести, що при будь-якому допустимому наборі значень параметрів рано чи пізно вона призведе нас до одного з виділених тривіальних випадків, тобто до задачі з набором параметрів, що є індикатором завершення рекурсивних викликів.

Рекурсивні алгоритми відносяться до класу алгоритмів з високою ресурсомісткістю, так як при великій кількості самовикликів рекурсивних функцій відбувається швидке заповнення стекової області. На трудомісткість рекурсивних алгоритмів впливає і кількість переданих функцією параметрів.

Важливою характеристикою рекурсивного алгоритму є глибина рекурсивних викликів – найбільше одночасне кількість рекурсивних звернень функції, що визначає максимальну кількість шарів рекурсивного стека, в якому здійснюється зберігання відкладених обчислень. При розробці рекурсивних програм необхідно враховувати, щоб глибина рекурсивних викликів не перевищувала максимального розміру стеку обчислювального середовища.

Як і будь-який алгоритм, рекурсія має свої переваги і недоліки.

Переваги рекурсії:

- простота і компактність запису алгоритмів;
- рекурсивна процедура показує, що потрібно робити, а нерекурсивна – як потрібно робити;
- легко програмувати обчислення за рекурентними формулами;
- легко доводиться коректність програми – її еквівалентність тим формулам, за якими розв’язується задача;
- з допомогою кінцевого виразу можна визначити нескінчену множину об’єктів.

Недоліки рекурсивних визначень:

- неефективне витрачання оперативної пам’яті. Для кожного рекурсивного виклику однієї і тієї ж функції виділяються нові комірки пам’яті. Кожного разу створюється нова копія цієї функції і всім локальним

змінним цієї копії потрібно нові місця в пам’яті. Тому необхідно уникати описання великої кількості локальних змінних в процедурі;

- низька швидкодія роботи програм. Програми, які містять рекурсивні визначення, як правило, працюють повільніше програм, які вирішують туж саму задачу без використання рекурсії.

2. Рекурсія і функціональне програмування. Одним з найважливіших ознак класифікації мов програмування є приналежність їх до одного зі стилів, основними з яких є процедурний, об’єктно-орієнтований, функціональний (аплікативний), логічний (реляційний). До категорії функціональних мов відносяться, наприклад Lisp, F#, Erlang, Scheme та ін. Функціональна програма складається із сукупності визначень функцій. Функції, в свою чергу, представляють собою виклики інших функцій і оголошень, які управляють послідовністю викликів. Обчислення починаються з виклику деякої функції, яка в свою чергу викликає функції, які входять в її визначення і т.д. у відповідності до ієрархії визначень і структури умовних оголошень. *Функція є рекурсивною*, якщо в її визначенні міститься виклик цієї самої функції. В декларативних мовах програмування (Prolog, Lisp) рекурсія є незамінним способом організації повторювальних процесів. Функції можуть прямо або опосередковано (підпорядковано) викликати самі себе. Кожний виклик повертає деяке значення у викликану функцію, обчислення якої після цього продовжується; цей процес повторюється до того часу доки функція не поверне кінцевий результат користувачу.

Основні правила побудови рекурсивних функцій:

- визначити кількість і вид аргументів;
- визначити характер результату;
- задати, щонайменше, одну умову закінчення рекурсії;
- визначити формування результату функції;
- описати формування нових значень аргументів для рекурсивного виклику.

Рекурсію, в першу чергу, розрізняють як пряму чи непряму, просту чи складену.

Прямою рекурсією називається рекурсія, при якій всередині тіла деякої функції міститься виклик тієї ж функції.

Непрямою рекурсією називається рекурсія, що здійснює рекурсивний виклик функції шляхом ланцюга викликів інших функцій. При цьому всі функції ланцюга, що здійснюють рекурсію, вважаються рекурсивними. Види рекурсивних викликів представлено на рис. 1.

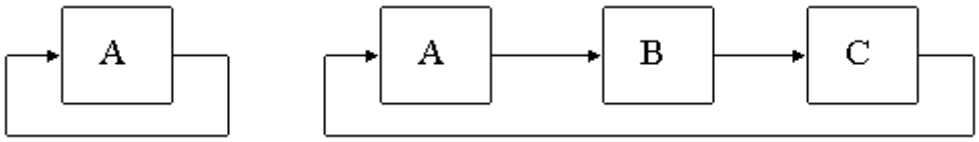

Рис. 1. Пряма і непряма рекурсії.

Проста рекурсія передбачає, що одиничний виклик функції самої себе зустрічається в одній чи декількох гілках. Така рекурсія відповідає організації циклічних процесів. Проста рекурсія матиме схематичний вигляд:

(define F (lambda (.....) (.... (F....) ... (F...)...)))

Класичним прикладом простої рекурсивної функції є функція обчислення факторіала числа. Потрібно визначити деяку функцію *F(n*), яка буде обчислювати значення *n!* через саму себе. Рекурсивне визначення цієї функції має вигляд:

$$F(n) = \begin{cases} 1, n = 0 \\ n * F(n-1), n > 0 \end{cases}$$

Функція рекурсивного обчислення факторіала мовою Scheme має вигляд:

```
(define F   (lambda (n)    (cond (( eq? n 0) 1)
                           (#t ( * n (F (- n 1 )))))))
```

Обчислення функції F можна представити як ланцюжок викликів нових копій цієї функції з передачею нових значень аргументу і повернень значень функції в попередню копію (рис. 2).

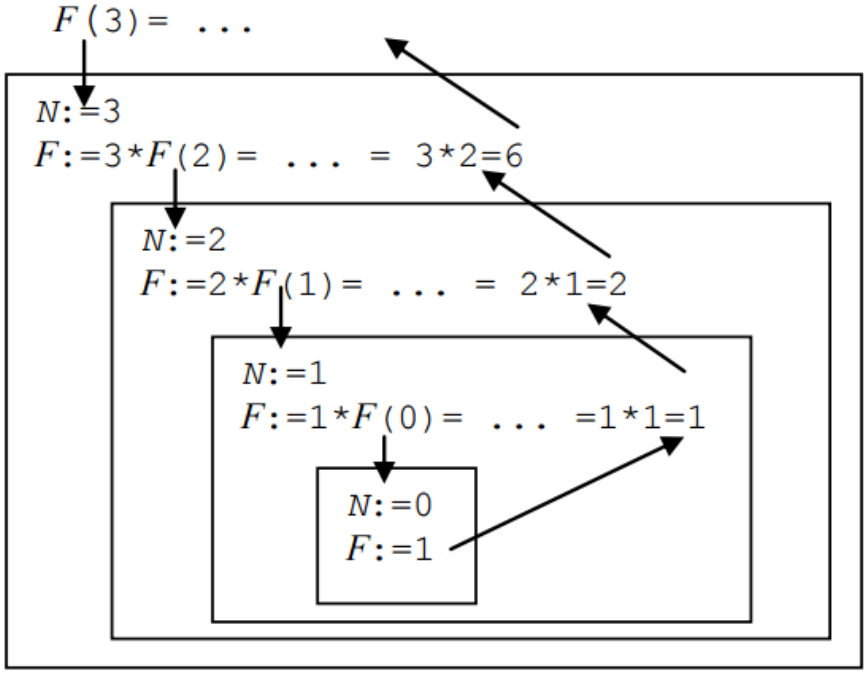

Рис. 2. Послідовні виклики функції F для обчислення 3!.

Як бачимо на прикладі, обчислення 3! функція F викликає точно такий же новий екземпляр самої себе. При цьому сама функція ще не завершилась, а новий її екземпляр вже починає працювати. Коли новий екземпляр завершить роботу, буде продовжена робота самої функції. Інформація про такі незавершені виклики рекурсивних підпрограм запам'ятовується в спеціальній області пам'яті – стеку. Важливою характеристикою рекурсивного алгоритму є глибина рекурсивних викликів. Кількість вкладених викликів функції називається глибиною рекурсії. При розробці рекурсивних програм необхідно враховувати, щоб глибина рекурсивних викликів не перевищувала максимального розміру стеку. У нашому випадку, при виконанні F(3) глибина рекурсії дорівнює 4, що відповідає кількості вкладених прямокутників на наведеному рисунку. Найбільший прямокутник відповідає першому рівню рекурсії, а найменший – останньому, тобто четвертому, рівню.

Рекурсія добре підходить для роботи зі списками, так як самі списки можуть складатися з підсписків, тобто мають рекурсивну будову. Підсписок – це список частина іншого списку.

Списки можна визначити за допомогою наступних правил Бекуса-Наура:

список → Nil; *список або порожній, або це*

список → (голова. хвіст); *точкова пара, хвіст якої є списком* (рекурсія "в ширину")

голова → атом

голова → список; рекурсія "в глибину"

хвіст → список.

Рекурсія є як у визначенні голови, так і у визначенні хвоста списку. Зауважимо, що наведене вище визначення безпосередньо відображає визначення функцій, що працюють зі списками, які можуть опрацьовуватися рекурсивним викликом голови списку, тобто "в глибину" (в напрямку CAR), і хвоста списку, тобто "в ширину" (в напрямку CDR).

Списки можна визначити і в наступній формі, яка підкреслює логічну рекурсивність побудови:

список → nil
список → (елемент ...)
елемент → атом
елемент → список; рекурсія

Розглянемо найпростіший приклад рекурсивного копіювання списку:

```
(define copy_list (lambda (A)
        (cond ((eq? list '()) '())
              (#t (cons (car A) (copy_list (cdr A)))))))
```

У рекурсивної функції дві або більше обчислювальних «гілки», причому одна обов'язково веде до завершення обчислень. В іншій – рекурсивній гілці (яка в даному випадку єдина) функція будує список з головного елемента (car A) і списку, який виходить в результаті виклику тієї ж функції copy_list з аргументом (cdr A) – «обезголовленим» вихідним списком. Так повторюється до тих пір, поки не буде істинним (null? A), тобто аргумент функції, що викликається не стане порожнім списком; в цьому випадку обчислення завершуються. Функція copy_list копіює тільки верхній рівень списку (дерева), адже черговим головним елемент (car A) цілком може сам бути списком, – але функція копіює його повністю. Це не дозволяє, наприклад, використовувати функції з такою структурою для пошуку окремих атомів, тому що вона знайде тільки (CD), але не C і D окремо.

Рекурсивна функція обов'язково повинна містити хоча б одну умову закінчення рекурсії, а також, значення аргументу при рекурсивних викликах має змінюватися. Дуже часто умовою закінчення рекурсії є порожній список-аргумент. У процесі роботи всі елементи вихідного списку обробляються послідовного до тих пір, поки не буде досягнутий кінець списку. Так вказану умову закінчення рекурсивної обробки можна записати:

((eq? list '()) < дія>)

Друга вимога до рекурсивної функції, що стосується зміни аргументу при рекурсивних викликах, найчастіше забезпечується рекурсивним викликом з аргументом-хвостом списку.

Розглянемо функцію, яка буде перевіряти рівність (еквівалентність) двох списків. Формальними параметрами у функції будуть списки (list1 і list2). Функція повинна повертати значення істина, якщо довжина списків однакова.

Функція має вигляд:

```
(define equal(lambda (list1 list2)
  (cond((and (eq? list '()) (eq? list '()) #t)
    ((and(not (eq? list '()) (not (eq? list '()))
      (equal (cdr list1) (cdr list2)))  (#t #f))))
```

Примітка 1. Для дотримання строго функціонального стилю програмування необхідно визначити на основі базових примітивів функції логічного множення *and* та логічного заперечення *not*.

3. Основні складові рекурсивного обчислення. Основна ідея рекурсивного визначення полягає в тому, що функцію можна за допомогою рекурентних формул звести до деяких початкових значень, до раніше визначених функцій чи до самої обумовленої функції, але з більш "простими" аргументами. Обчислення такої функції закінчується в той момент, коли воно зводиться до відомих початкових значень.

Рекурсивне визначення складається принаймні з двох обчислювальних гілок: термінальної і рекурсивної.

Термінальна гілка необхідна для закінчення обчислень. Без термінальної гілки рекурсивний виклик був би нескінченним. Термінальна гілка повертає результат, який є базою для обчислення результатів рекурсивних викликів.

Після кожного виклику функцією самої себе, обчислення повинні наближатися до термінальної гілки. Завжди повинна бути впевненість, що рекурсивні виклики ведуть до термінальної гілки.

Реалізація рекурсивних викликів функцій опирається на механізм стеку викликів. Адреса повернення і локальні змінні функції записуються в стек, кожен наступний рекурсивний виклик цієї функції користується своїм набором локальних змінних. При надмірно великій глибині рекурсії може наступити переповнення стеку викликів. Коли обчислювальний процес доходить до рекурсивної гілки, то поточний процес призупиняється, і новий такий же запускається з початку, але вже на новому рівні. Перерваний процес запам’ятовується і перебуває в режимі очікування доти, поки не закінчиться новий. Новий процес, в свою чергу, може призупинитися і чекати і так далі. Таким чином, якби утворюється стек перерваних процесів, з яких виконується

лише останній в даний час процес (рис.3). Після його закінчення виконуватиметься попередній до нього процес. Цілком весь процес виконано, якщо стек перерваних процесів буде порожній.

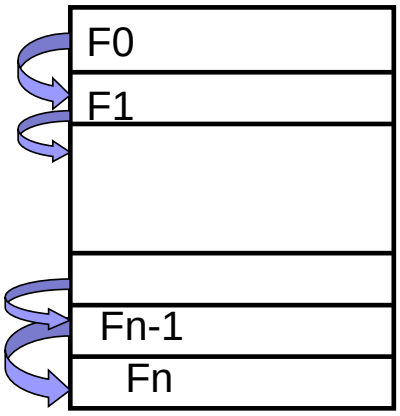

Рис.3. Стек обчислювальних процесів при рекурсивному виклику.

Формування стеку перерваних процесів в рекурсивних функціях називають розгорткою рекурсії, а вилучення елементів з цього стеку – згорткою рекурсії.

Звичайно, що прослідкувати подумки обчислення в рекурсії надзвичайно складно. Це практично неможливо для функцій більш складних. Таким чином, потрібно вміти писати рекурсивні функції, без того щоб представляти точно порядок обчислень.

При написанні рекурсивних функцій потрібно планувати термінальні і рекурсивні гілки.

Для планування термінальної гілки рекурсивної функції потрібно вирішити, коли функція може повернути значення без рекурсивного виклику. Планування рекурсивної гілки передбачає визначення рекурсивного виклику функції зі спрощеним аргументом і використання результату для обчислення значення при поточному аргументі. Таким чином, потрібно вирішити:

1. Як спростити аргумент, наближуючи його крок за кроком до кінцевого значення.
2. Крім того, необхідно побудувати форму, яку називають рекурсивним відношенням, яка пов’язує правильне значення поточного виклику зі значенням рекурсивного виклику.

Інколи просто знайти це відношення, а якщо є труднощі у пошуку, то рекомендується виконати таку послідовність кроків:

a. Визначити значення деякого простого виклику функції і її відповідного рекурсивного виклику.

b. Визначити відношення між цими викликами функцій.

Порядок розміщення термінальних та рекурсивних гілок є важливим. Спочатку варто розміщувати прості термінальні гілки, далі – складніші (якщо вони є), за ними мають слідувати рекурсивні гілки. Неправильне розміщення обчислювальних гілок, відсутність термінальних гілок або помилки в означенні термінальних та рекурсивних гілок приводять до нескінченної рекурсії.

Приклади рекурсивного визначення обчислень.

1) Визначити функцію power. Вона має два числових аргумента m і n та обчислює значення m в степені n.

Тобто (power 2 3) повертає результат 8 .

Спочатку складемо рекурсивну таблицю.

Крок 1. Завершення (Термінальна гілка).

Якщо n=0 – аргумент, то (power 2 0) = 1 – значення.

Крок 2. Рекурсивна гілка.

Пошук рекурсивних відношення між (power m n) і (power m (- n 1))

2a. Приклади рекурсії:

(power 5 3) =125

(power 5 2) = 25

(power 3 1)=3

(power 3 0)=1

2b. Характеристичні рекурсивні відношення

(power m n) може бути отримано з (power m (- n 1) множенням на m

Дамо визначення мовою Scheme:

```
(define power  (lambda (m n)
        (cond ((eq? n 0) 1)
                (#t (* m (power  m (- n 1))))))
```

2) Побудувати Scheme-функцію, яка обчислює кількість елементів на верхньому рівні у списку.

Хід міркувань полягає в тому, що потрібно послідовно застосовувати функцію відкидання голови списку і рахувати скільки разів можна застосовувати функцію-селектор, аж поки не дійдемо до порожнього списку. Якщо досягнуто кінець списку, необхідно повернути значення 0.

Отже, мовою Scheme текст функції такий:

```
(define ln (lambda (list)
        (cond ((eq? list '()) 0) (#t (+ (ln (cdr list)) 1)))))
```

Вище було розглянуто випадок рекурсії з одною термінальною і одною рекурсивною гілками.

Проте у визначенні рекурсивної функції може бути декілька термінальних гілок.

Дві термінальні гілки будуть в тому випадку, коли ведеться пошук цілі в послідовності значень і бажано отримати результат, як тільки ціль знайдено.

Наприклад, можливі такі випадки.

Гілка 1. Ціль знайдено і потрібно повернути відповідь.

Гілка 2. Ціль не знайдено і немає більше елементів.

Декілька рекурсивних гілок можуть бути потрібними, якщо функція опрацьовує всі елементи в структурі, але використовує деякі елементи інакше, ніж інші. У цьому випадку складаються принаймні два рекурсивних відношення.

4. Типи рекурсій. Стосовно структури обчислювального процесу рекурсивних функцій розрізняють просту та складну рекурсії. Складна рекурсія буває таких форм: паралельна, взаємна, вищого порядку.

Паралельна рекурсія передбачає, що тіло визначення деякої функції містить виклик іншої функції, декілька аргументів якої є рекурсивними викликами першої функції. Така рекурсія матиме схематичний вигляд:

(define F (lambda (.....) (...(G (F....) ... (F...)...))))

Паралельність рекурсії полягає не в сенсі часу, а в сенсі логіки. У такому випадку кажуть про текстуальну паралельність рекурсії.

Паралельна рекурсія буде одночасно рекурсією за значенням, так як виклики рекурсивної функції обчислюють аргументи другої функції.

Розгляним для прикладу копіювання списків в глибину (тобто вкладених списків).

```
(define full_copy (lambda (list)
  (cond ((eq? list '()) '()) ((atom list) list)
(cons (full_copy(car list)) (full_copy (cdr list))))))
```

Примітка 2. Для дотримання строго функціонального стилю програмування необхідно визначити на основі базових примітивів допоміжну функцію *atom* для перевірки чи S-вираз є атом.

Паралельну рекурсію ще називають рекурсією по дереву. Прикладом паралельної рекурсії є також функція обчислення чисел, що належать послідовності Фібоначчі. Послідовність чисел, кожен елемент якої дорівнює сумі двох попередніх, два перші елементи якої дорівнюють 0 і 1 відповідно, називається послідовністю Фібоначчі.

Загальне правило для чисел Фібоначчі можна сформулювати так:

$$Fib(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ Fib(n-2) + Fib(n-1) \end{cases}$$

Обчислення n-го члена ряду Фібоначчі наведено нижче:

```
(define Fib (lambda(n) (cond ((= n 0)1) ((= n 1)1)
                             ((+ (Fib (- n 2))(Fib (- n 1)))))))
```

Розглянемо схему цього обчислення. Для того, щоб обчислити Fib (5), ми спочатку обчислюємо Fib (4) і Fib (3). Щоб обчислити Fib (4), ми обчислюємо Fib (3) і Fib (2). В загальному, отриманий процес схожий на дерево, як показано на рис. 4.

На кожному рівні (окрім дна) гілки розділяються надвоє; це відображає той факт, що функція Fib при кожному виклику звертається до самої себе двічі. Число раз, які ця функція викличе Fib (1) або Fib (0) в точності рівна Fib (n+1). Таким чином, число кроків нашого процесу росте експоненціально.

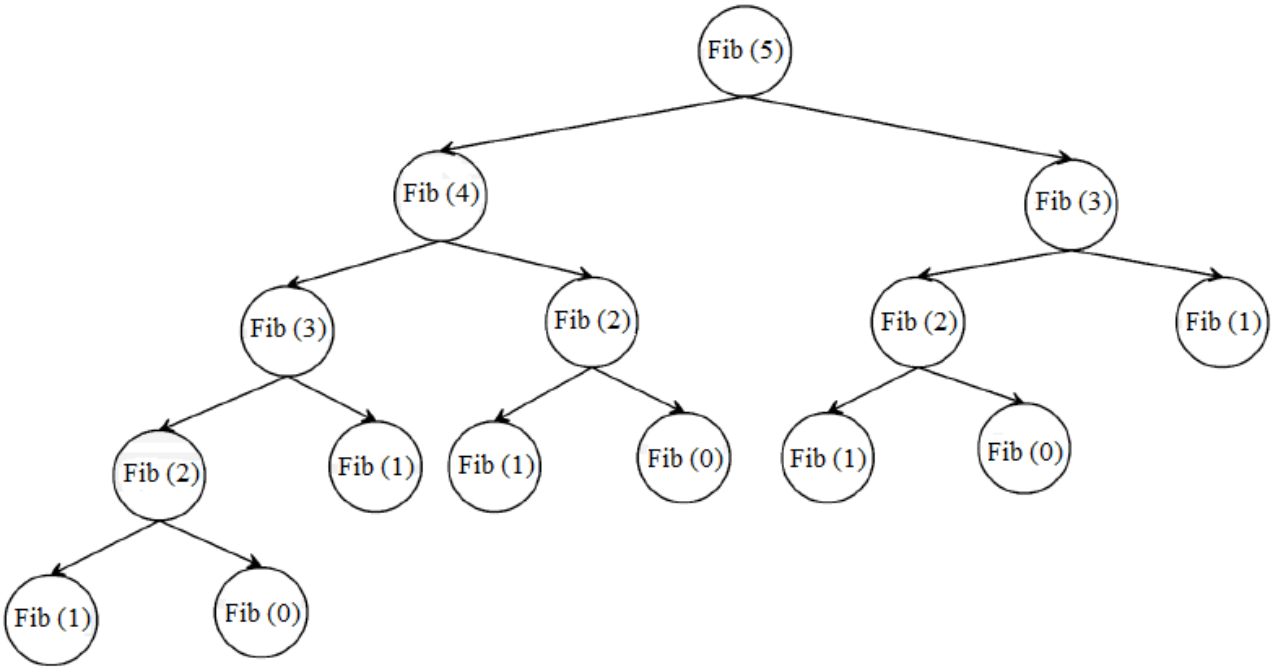

Рис. 4. Дерево рекурсії для обчислення п’ятого члена послідовності Фібоначчі.

У рекурсивних підпрограм часто зустрічається недолік – неефективність. Так, порівнюючи швидкість обчислення чисел Фібоначчі з допомогою ітеративної і рекурсивної функції побачимо, що ітеративна функція виконується миттєво, незалежно від значення n. При використанні рекурсивної функції уже при n=40 помітна затримка при обчисленні, а при більших n результат появиться не скоро. Однак коли ми будем розглядати процеси не з

числами, а з ієрархічними структурованими даними, побачимо, що деревовидна рекурсія є потужнім інструментом.

Взаємна рекурсія передбачає, що у визначенні деякої функції викликається інша функція, яка, у свою чергу, містить виклик першої функції. Інша назва такої рекурсії – перехресна. У такому випадку рекурсія матиме схематичний вигляд:

```
(define F (lambda (.....)
                    (...(G ...)...)))
(define G  (lambda (.....)
                    (...(F....) ...)))
```

Для прикладу розглянемо наступну задачу. Потрібно визначити предикат, який перевірятиме чи містить список парне число елементів на верхньому рівні, і предикат, який перевірятиме список на непарність числа елементів.

Декларативне описання задачі:

- пустий список парний;
- у парного списка непарний хвіст;
- у непарного списка парний хвіст.

```
(define even1 (lambda(l)
(cond ((eq? l '()) #t) ( #t (number1 (cdr l))))))

(define number1 (lambda(l)
(cond ((eq? l '()) '()) (#t (even1 (cdr l))))))
```

Рекурсія вищого порядку є в тому випадку, якщо аргумент рекурсивного виклику функції є рекурсивний виклик цієї ж функції. Порядок рекурсії визначається рівнем, на якому знаходиться вкладений рекурсивний виклик. Проста, паралельна, взаємна – це рекурсії нульового порядку.

Якщо функція викликає саму себе, а аргумент формується рекурсивно нею ж, це рекурсія першого порядку. Така рекурсія матиме такий схематичний вигляд:

```
(define F (lambda (.....) (...(F (F....) ...))))
```

Якщо функція викликає саму себе, а аргумент формується рекурсивно нею ж, де знову аргументом є рекурсивний виклик – це рекурсія другого порядку. І такі міркування можна продовжити далі. Рекурсія другого порядку матиме такий схематичний вигляд:

```
(define F (lambda (.....) (...(F (F(F....) ...)))))
```

Приклад функції з рекурсією другого порядку – функція Аккермана.
Математично функція Аккермана визначається в такий спосіб:

ackerman (x, 0) = 0
ackerman (0, y) = 2 y
ackerman (x, 1) = 2
ackerman (x, y) = ackerman ((x-1), ackerman (x, (y-1))

Наведеному математичному опису відповідає наступне визначення функції:

```
(define (ackerman x y)
    (cond ((eq? y 0) 0) ((eq? x 0) (* 2 y))
    ((eq? y 1) 2)  ((ackerman (- x 1) (ackerman x (- y 1))))))
```

При роботі з функціоналами корисно вміти специфікувати відображувальну функцію, дане, що представляє відображувана множина, а також результат виділення відображуваної множини. Вся ця інформація може бути корисною при налагодженні програм, що містять функції вищих порядків.

Стосовно місця виклику розрізняють рекурсію за аргументом і рекурсію за значенням. Рекурсія за аргументом – це якщо у якості результату повертається значення іншої функції, аргумент якої формується на основі рекурсивного виклику.

Приклад рекурсії за аргументом (об'єднання списків):

```
(define union (lambda (list1 list2)
    (cond ((eq? list1 '()) list2) ((member (car list1) list2) (union (cdr list1) list2))
                                  (#t (cons (car list1) (union (cdr list1) list2))))))
```

Примітка 3. Для дотримання строго функціонального стилю програмування необхідно визначити на основі базових примітивів допоміжну функцію *member* для перевірки чи елемент належить списку.

Рекурсія за значенням – це якщо рекурсивний виклик є виразом, який формує результат функції.

Приклад рекурсії за значенням (визначення входження заданого елемента в списку):

```
(define memberall (lambda (element list)
  (cond ((eq? list '())  #f)
    (#t (or (eq? (car list) element)
            (memberall  element (cdr list)))))))
```

Примітка 4. Для дотримання строго функціонального стилю програмування необхідно визначити на основі базових примітивів допоміжну функцію логічного додавання *or*.

5. Рекурсія та ітерація

Оскільки обчислювальні можливості операторних і функціональних програм однакові, кожне обчислення можна записати будь-яким з цих двох способів. Однак програми, написані різними способами, за своїми властивостями помітно відрізняються одна від одної. Тому потрібно зважувати, який спосіб у якій ситуації кращий.

Факторами, що впливають на розв’язок, можуть бути ефективність обчислень в плані часу чи обсягу пам'яті, довжина програми, її прозорість і зрозумілість і т.д. На розв’язок може вплинути, наприклад, наскільки використовувана обчислювальна система забезпечує опрацювання різних видів рекурсії. Архітектура більшості сучасних машин розрахована в першу чергу на операторне програмування, а перетворення рекурсивних функціональних обчислень в ефективні ітераційні обчислення вдається не завжди.

З рекурсивної будови списків випливає, що рекурсивне визначення функцій, що обробляють їх, буде коротше і наочніше відіб'є суть обчислень. У свою чергу ітеративна програма може бути з погляду обчислень більш ефективною. Це є наслідком того, що обчислення рекурсивної функції вимагає при кожнім виклику деякого обліку рівнів і значень параметрів на різних рівнях. Ця робота робиться впусту, якщо дані, що запам'ятовуються, в обчисленнях участі не беруть. Однак транслятори Ліспа, а часто навіть і інтерпретатори здатні аналізувати код, перетворити, наприклад, кінцеву рекурсію в звичайну ітерацію. У цьому випадку різниці в ефективності немає.

Перетворення ітерації в рекурсію звичайно простіше, ніж навпаки. Це однак потрібно рідко. Зворотний напрямок перетворення необхідний при трансляції програми в іншу мову, що не має рекурсії, наприклад при трансляції Ліспа в машинну мову. Якщо рекурсія складна, то і задача стає складною. Для перетворення рекурсивних обчислень в ітеративну форму розроблені спеціальні методи, на яких ми тут однак зупинятися не будемо.

Найкраще рекурсивні методи виправдані в задачах, у яких зустрічається формальна і насамперед природна рекурсія в структурах і процесах. Рекурсивні процеси в практичному програмуванні звичайно не є чисто рекурсивними, а з ними часто зв'язані різні побічні ефекти. У таких випадках доречно

використовувати рекурсивне операторне програмування, іншими словами компромісне об'єднання рекурсивного й операторного програмування.

Наприклад, рекурсію у хвостовій частині можна перетворити в ітерацію, але зберегти рекурсію у головній частині. Такий розв’язок підкреслює інтерпретацію списку у вигляді упорядкованої множини, до кожного елемента якої рекурсивно застосовується деяка дія.

Незважаючи на відсутність однозначних критеріїв вибору між ітерацією та рекурсією, можна запропонувати ряд правил:

• при однакової складності рекурсивної і нерекурсивною версій алгоритму перевагу слід віддавати нерекурсивною версії, як більш ефективної за витратами пам'яті і процесорного часу (приклад з функціями звернення списку);

• якщо застосування нерекурсивною версії алгоритму значно ускладнює його логіку або вимагає застосування більш складних структур даних, то перевагу слід віддати рекурсивному алгоритму.

Операторне програмування особливо корисно, якщо під час рекурсії потрібно одержати побічні ефекти, наприклад, для запису чи друку проміжних результатів. Реалізуємо для прикладу гру Ханойські вежі.

Гра полягає в наступному. Використовується три вертикальних стержні A, B, C та сукупність N круглих дисків різної величини з отвором (рис. 4.). У початковому стані диски нанизані один по одному у відповідності зі своїм розміром на стрижень A. Метою є перенесення всіх дисків на стержень B. Однак диски переносяться не в довільному порядку, при переносі потрібно виконувати наступні правила:

1. За один раз можна перенести тільки один диск.
2. Більший по розмірі диск не можна покласти на менший.

Третій стрижень можна використовувати як допоміжний (проміжний). Якщо він вільний чи там лежить більший диск, то на нього можна перекласти черговий диск на той час, поки переноситься диск, що лежить нижче. У цій ідеї міститься розв’язок гри. Ідею потрібно лише узагальнити. Перенесемо спочатку N-1 дисків на допоміжний стрижень C, щоб можна було перенести нижній диск на стрижень B. Після цього ми можемо N-1 дисків перенести назад на місце. Ми в такий спосіб розділили задачу на три підзадачі, з яких перша й остання зводяться до первинної, але меншої на один диск задачі і середня – до простого переносу диска.

Як результат ми одержимо наступні рекурсивні правила:

1. Перенести зі стрижня A N-1 дисків на допоміжний стрижень C (задача Ханойські вежі для N-1).

2. Перенести нижній диск зі стрижня A на стрижень B.
3. Перенести зі стрижня C N-1 дисків на стрижень B (задача Ханойські вежі для N-1).

При проведенні скорочених ігор можна вільно використовувати всі стержні, оскільки більший диск не є перешкодою для переносу інших дисків. Потрібно лише пам'ятати, з якого стержня на який переносяться диски на поточному рівні, оскільки вихідний, допоміжний і цільовий стержні міняються місцями. Внаслідок рекурсії первинна проблема так само, як і всі підзадачі зводяться зрештою до легко розв'язуваної задачі з одним диском.

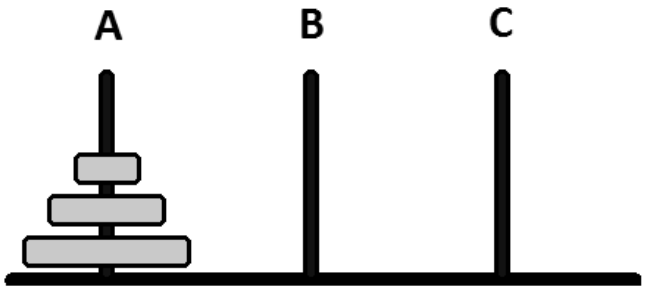

Рис. 5. Задача “Ханойські вежі”.

Хоча інтуїтивно алгоритм здається ясним, проте не очевидно, як реалізуються переноси і зміна ролей стрижнів на різних рівнях рекурсії. Це, однак, автоматично враховується механізмом рекурсії. У цьому ми можемо переконатися, запрограмувавши алгоритм:

```
(define (move-disks num-disks from spare to)
        (cond ((= num-disks 1) (list (cons from to)))
              (#t (append (move-disks (- num-disks 1) from to spare)
                  (move-disks 1 from spare to)
                  (move-disks (- num-disks 1) spare from to)))))
```

Задачі для веж, що складаються з одного чи двох дисків, розв’язуються просто. Задача для трьох дисків вимагає вже більше переносів:

- (move-disks 3 'A 'B 'C)

A -➢ C A -➢ B C -➢ B A -➢ C B -➢ A B -➢ C A -➢ C

Легко переконатися в правильності переносів і в тім, що розв’язок підходить для вежі довільної висоти. Однак кількість переносів для більш високих веж швидко зростає. За допомогою рекурентних формул можна показати, що в загальному випадку гра рівня N вимагає 2^{N-1} переносів.

Контрольні запитання

1. Що таке рекурсія?
2. Що таке пряма рекурсія? Що таке непряма рекурсія?
3. Що таке проста рекурсія? Що таке складна рекурсія?
4. Які основні правила побудови рекурсивних функцій?
5. Дайте визначення рекурсії за аргументом?
6. Дайте визначення рекурсії за значенням?
7. Сформулюйте правила Бекуса-Наура?
8. Що таке рекурсивна тріада?
9. Назвіть переваги і недоліки рекурсивних визначень?
10. В яких завданнях доцільно використовувати рекурсивні функції?
11. Яке значення термінальної гілки в рекурсивному визначенні функції?
12. Чи істотний порядок розміщення обчислювальних гілок у рекурсивному визначенні функцій?
13. Як реалізуються циклічні обчислювальні процеси у функційному програмуванні?
14. Як виглядає рекурсія «в ширину» і в «глибину» у списку?
15. Поясніть поняття “нескінченна рекурсія”. Наведіть приклади. Як виглядає нескінченна рекурсія на практиці?
16. Як розрізняють рекурсію за місцем використання? Наведіть приклади.
17. Які форми рекурсії розрізняють за структурою обчислювального процесу?
18. Чому списки зручно опрацьовувати з допомогою рекурсивних функцій?
19. Що таке рекурентність обчислень?
20. Що таке текстуальна паралельність рекурсії?
21. Чи можливим є наявність трьох термінальних гілок у рекурсивному визначенні функції? Відповідь обґрунтуйте.
22. Доведіть, що список є рекурсивною структурою.
23. Обгрунтуйте, що рекурсія є природнім обчисленням у декларативному програмуванні.
24. Як рекурсія пов’язана з ітераціями?

Хід виконання роботи

1. Ознайомитися з теоретичним матеріалом за темою лабораторної роботи.
2. Отримати індивідуальне завдання .
3. Визначити функцію відповідно до свого варіанта.
4. Продемонструвати роботу функціональної програми на різних тестових даних.
5. Підготувати та здати звіт про виконання лабораторної роботи.

Варіанти завдань до лабораторної роботи №2.

Варіант 1. Визначити функцію СУМА(X), де X − список довільної довжини, який складається з цілих чисел, а результат функції – сума цих чисел.

Варіант 2. Побудувати функцію предикатного типу НАЛЕЖИТЬ(E, X), яка перевіряє чи заданий елемент E входить до списку X.

Варіант 3. Визначити функцію ДОБУТОК(X), де X − список довільної довжини, який складається з цілих чисел, а результат функції – добуток цих чисел.

Варіант 4. Визначити функцію ОСТАННІЙ(X), яка видає як результат останній елемент списку X.

Варіант 5. Визначити функцію ОБЕРНУТИ(X), яка видає список X в оберненому порядку на верхньому рівні.

Варіант 6. Визначити функцію ВИДАЛИТИ(E, X), яка видаляє перше входження заданого елемента E зі списку X.

Варіант 7. Визначити функцію ДОВЖИНА(X), яка обчислює кількість елементів списку X на верхньому рівні.

Варіант 8. Визначити функцію ВИДАЛИТИ_ОСТАННІЙ (X), яка вилучає останній елемент зі списку X.

Варіант 9. Визначити функцію НОВИЙ(X, У), яка чергуючи елементи двох списків, утворює новий список.

Варіант 10. Визначити функцію ОБ’ЄДНАТИ(X, У), яка об’єднує два списки X і У в один.

Варіант 11. Визначити функцію, яка вилучає кожен другий елемент зі списку X.

Варіант 12. Визначити функцію предикатного типу, яка визначає чи її аргумент є однорівневим списком.

Варіант 13. Визначити функцію СПИСОК(X), яка впорядковує атоми у багаторівневому списку в один рівень.

Варіант 14. Визначити функцію предикатного типу, яка визначає чи два S-вирази еквівалентні.

Варіант 15. Визначити функцію, яка вилучає зі списку перший та останні елементи.

Варіант 16. Визначити функцію, яка обчислює кількість атомів у списку на всіх рівнях вкладеності.

Варіант 17. Визначити функцію, яка чергуючи елементи двох введених списків утворює новий список.

Варіант 18. Визначити функцію, яка обчислює кількість вкладених списків у введеному списку.

Вимоги до звіту

1. Титульний аркуш.
2. Тема і мета лабораторної роботи.
3. Теоретичні відомості, в яких даються відповіді на контрольні запитання (номери вказуються викладачем).
4. Формулювання завдання.
5. Текст програми.
6. Результати роботи програми.
7. Змістовні висновки за результатами роботи.

Список літератури

1. Introduction: Why Lisp? [Electronic resource]. – Web access: http://www.gigamonkeys.com/book/introduction-why-Lisp.html (2015).
2. Lisp (programming language) [Electronic resource]. – Web access: http://en.wikipedia.org/wiki/Lisp_%28programming_language%29 (2015).
3. LISP [Electronic resource]. – Web access: http://www.nist.gov/Lispix/doc/Lispix/Lisp-new.htm (2015).
4. LISP [Електронний ресурс]. – Веб-доступ до сторінки: http://lurkmore.to/LISP (2015).
5. List of programming languages by type [Electronic resource]. – Web access: http://en.wikipedia.org/wiki/List_of_programming_languages_by_type (2015).
6. PLT Scheme is a Racket [Electronic resource]. – Web access: http://racket-lang.org/new-name.html (2015).
7. Бадаєв Ю.І. Теорія функціонального програмування. Мови Common LISP та Auto LISP. Навчальний посібник. – К.:КПІ. – 1999.–150с.
8. Большакова Е.И., Груздева Н.В. Основы программирования на языке Лисп: Учебное пособие. – 2010.
9. Грэм П. ANSI Common Lisp. – Пер. с англ. – СПб.: Символ-Плюс, 2012.
10. Дехтяренко И.А. Декларативное программирование. – 2003.
11. Душкин Р.В. Текст лекций по курсу «Функциональное программирование». – Москва, 2001.
12. Заєць В.М. Функційне програмування. – Л.: 2002.
13. Медведєв М.Г. Функціональна мова програмування ЛІСП: Навчальний посібник для студентів. – Київ – 1999.
14. Морозов М.Н. Функциональное программирование. – 1999.
15. Новицкая Ю.В. Основы логического и функционального программирования. – 2006.
16. Сайбель Питер Практическое использование Common Lisp. – 2015.
17. Хювёнен Э, Сеппянен И. Мир Лиспа. Т.1: Введение в язык Лисп и функциональное программирование / Э.Хювёнен, И.Сеппянен. – М.: Мир, 1990. – 458 с.
18. Хювёнен Э, Сеппянен И. Мир Лиспа. Т.2: Методы и системы программирования / Э.Хювёнен, И.Сеппянен. – М.: Мир, 1990. – 332 с.

Термінологічний покажчик

НАВЧАЛЬНЕ ВИДАННЯ

ВИКОРИСТАННЯ РЕКУРСІЇ У ФУНКЦІОНАЛЬНОМУ ПРОГРАМУВАННІ

МЕТОДИЧНІ ВКАЗІВКИ
до лабораторної роботи № 2 з курсу “Декларативне програмування”
для студентів спеціальності 121 «Інженерія програмного забезпечення»

Укладачі Левус Євгенія Василівна
Шкраб Роман Романович

Редактор

Комп’ютерне верстання