В.І. Месюра

Функціональне та логічне програмування
Частина 1
Основи функціонального програмування

Вінниця ВДТУ 2001

Міністерство освіти і науки України
Вінницький державний технічний університет

В.І.Месюра

Функціональне та логічне програмування
Частина 1
Основи функціонального програмування

Затверджено Ученою радою Вінницького державного технічного університету як навчальний посібник для студентів спеціальностей: "Інтелектуальні системи прийняття рішень", "Програмне забезпечення автоматизованих систем". Протокол №5 від 27 грудня 2000 року.

Вінниця ВДТУ 2001

УДК 519.68
М53

Р е ц е н з е н т и:
А.М.Петух, доктор технічних наук, професор
В.П.Кожем'яко, доктор технічних наук, професор
О.М.Мельников, кандидат технічних наук, професор

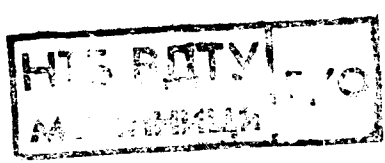
Рекомендовано до видання Ученою Радою Вінницького державного технічного університету Міністерства освіти і науки України

Месюра В.І.

М53 Функціональне та логічне програмування. Частина 1. Основи функціонального програмування. Навчальний посібник. – Вінниця. ВДТУ, 2001, 90 с.

Посібник містить теоретичний матеріал та практичні завдання з функціонального програмування, та перелік основних питань до лабораторних робіт, що відносяться до першої частини навчальної дисципліни "Основи функціонального та логічного програмування".

Розрахований на студентів комп'ютерних спеціальностей всіх форм навчання.



УДК 519.68

ЗМІСТ

ВСТУП

Швидкий розвиток технологій штучного інтелекту відкрив перспективи ефективного застосування обчислювальної техніки не тільки для автоматизації суто технологічних процесів, розв'язання задач рутинного, обчислювального характеру, але і для вирішення проблем, які завжди вважалися прерогативою людини і потребують для свого розв'язання інтелектуальних зусиль.

Однією з найважливіших проблем в автоматизації розв'язку інтелектуальних задач є забезпечення ефективної обробки символьної інформації, оскільки придатні мисленню і мові людини об'єкти, події і ситуації реального світу неможливо природно відобразити за допомогою чисел і масивів. Саме тому вже в 1956 р. основоположники штучного інтелекту Ньєл, Шоу і Саймон розробили першу мову програмування ІПЛ, що була призначена для обробки списків [1].

Поняття "список" виявилося дуже вдалим. У вигляді списків зручно зображати найрізноманітнішу інформацію: алгебраїчні вирази, графи, елементи кінцевих груп, множини, правила виведення та інші складні об'єкти. Список є найбільш гнучкою формою представлення інформації в пам'яті сучасних комп'ютерів.

Тому саме на основі обробки списків і були створені два основних класи мов програмування штучного інтелекту: функціональні мови, найбільш яскравим представником яких є ЛІСП, і логічні мови, серед яких перше місце по праву займає ПРОЛОГ.

Сучасний програміст повинен обов'язково бути обізнаним з основами функціонального і логічного програмування, які подають альтернативні до традиційних процедурних мов (Паскаль, Си та інші) методи та підходи до розв'язання практичних задач. Програміст не знайомий з ЛІСПом і Прологом залишається позбавленим повноти сприйняття будь-якої проблеми, яка виникає завдяки можливості подивитися на проблему з різних точок зору.

Особливістю функціональних мов програмування є їх функціональна спрямованість, тобто подання програм у вигляді функцій. Причому функція розуміється як правило, що зіставляє елементи певного класу з відповідними елементами іншого класу. Сам процес зіставлення не впливає на роботу програми, важливим є тільки його результат – значення функції. Це дозволяє відносно просто писати і відлагоджувати великі програмні комплекси [3].

Даний навчальний посібник присвячений викладенню основ функціонального програмування на базі знайомства з мовою функціонального програмування ЛІСП, відповідно до першого розділу програми дисципліни "Функціональне і логічне програмування", в другому розділі якої розглядаються питання логічного програмування.

1 ВВЕДЕННЯ ДО ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ

1.1 Програмування за допомогою функцій

Концепція функції є одним з фундаментальних понять математики. Функція *f(x)* є правилом, за допомогою якого здійснюється зіставлення кожного елемента деякого класу (області визначення - *domains*) з відповідним йому єдиним елементом з іншого класу (області значень - *range*), як це показано на рисунку 1.1.

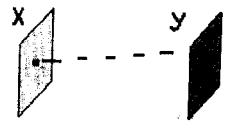


Рисунок 1.1 - Графічне відображення функціонального відношення *y=f(x)*

Кажуть, що *f* відображає *x* у *f(x)* або що f*(x)* є результатом застосування *f* до аргументу *x*. Функція має багато форм запису. Наприклад,

$$Y=F(x),\ F(x)\rightarrow Y,\ F: x\rightarrow Y$$

та ін.

Взагалі функція може мати скільки завгодно аргументів. Функція без аргументів має постійне значення. Прикладами функцій є:

- abs(-3) → 3 абсолютна величина;
- + (2 , 3) → 5 додавання;
- union ((a , b), (c , b)) → (a , b , c) об'єднання множин;
- кількість_букв (слово) → 5.

Функція може задаватися аналітичним виразом, таблицею, правилом та ін. Наприклад, функцію ***макс(x,y)*** можна задати за допомогою такого правила [6]:

макс(x,y) = *x*, якщо *x*>= *y*, в іншому випадку *y*.

Більш компактний запис можна отримати, якщо використати знайому з інших мов програмування конструкцію **якщо...то...інакше**:

макс(x,y) = **якщо** *x* >= *y* **то** *x* **інакше** *y*.

При цьому форма **якщо...то...інакше** вимагає спочатку обчислити предикат *x* >= *y* і в залежності від результату (істина або хибність), обрати підвираз *x* або y в якості результату конкретного застосування функції.

Наприклад, ***макс***(1,3), тобто результат застосування функції ***макс*** до аргументів (1,3) обчислюється за таким правилом:

1) конструкція **застосування**, яка називається в програмуванні також **викликом функції**, потребує, щоб параметрам x і y було зіставлене значення 1 і 3 відповідно;
2) обчислюється вираз $x >= y$, тобто 1 >= 3, внаслідок чого буде отримане значення **хибність**.
3) відповідно до отриманого значення предикату **(хибність),** в якості результату застосування функції обирається значення y, тобто 3.
4) функція ***макс***(1,3) повертає значення 3.

Для забезпечення можливості програмування за допомогою функцій, необхідно визначити деякий набір базових функцій (наприклад, таких як ***макс)***, і викликаючи їх з відповідними аргументами, обчислювати їх результати.

Функція по суті сама по собі вже є програмою, яка сприймає вхідну інформацію (аргументи) і повертає вихідну інформацію (результат застосування функції до аргументів). Для конструювання більш потужних програм, необхідно вміти визначати нові функції через старі. При цьому нові функції мають посилатись при їх обчисленні тільки на основні, базові функції, чого можна досягти побудувавши ієрархію визначень [5,6].

Розглянемо, наприклад, таке визначення:

найбільший(x,y,z)=***макс***(***макс***$(x,y),z)$

Бачимо, що нова функція обчислює більший з трьох її аргументів. Вона робить це з використанням функції ***макс***, спочатку застосовуючи її до двох перших аргументів, а потім, до проміжного результату і третього аргументу. Даний приклад ілюструє один з фундаментальних способів побудови нових функцій з старих з використанням композиції функцій. Вкладенням одного застосування функції ***макс*** у інше здійснюється композиція функції ***макс*** з собою ж. Взагалі функції можуть бути вкладені одна в одну на довільну глибину і складати таким чином довільно глибокі композиції. Наприклад, найбільше з шести чисел a,b,c,d,e,f може бути визначене за допомогою однієї з таких композицій:

макс(найбільший(a,b,c),***найбільший***$(d,e,f))$
найбільший(***макс***(a,b),***макс***(c,d), ***макс***$(e,f))$
макс(***макс***(***макс***(a,b),***макс***$(c,d))$,***макс***$(e,f))$

Для програмування за допомогою функцій треба лише визначити достатньо велику множину базових функцій і потім використовувати композицію для визначення нових функцій на основі базових.

До числа основних функцій необхідно включити і арифметичні операції, які слід визначити як функції.

Наприклад:

- ***додати***$(x,y)=x+y$;
- ***перемножити***$(x,y)=x*y$;

- ***відняти***$(x,y)=x-y$;
- ***поділити***$(x,y)=x/y$.

При цьому за допомогою введених функцій звичні записи алгебраїчних виразів перетворяться на такі:

$$a + b * c \Rightarrow \textbf{додати}(a, \textbf{перемножити}(b,c));$$

$$(2 * x + 3 * y)/(x-y) \Rightarrow$$

поділити(***додати***(***перемножити***(2,x),***перемножити***(3,y)),***відняти***(x,y)).

Запис виразів в цій формі називається аплікативною структурою виразів в математичній мові (мові програмування). Кожний вираз в такій мові можна розбити на складові частини, кожна з яких є або операцією або операндом. Операнд позначує значення, в той час як операція позначує функцію. Структура виразів є дуже простою, вона складається з операцій, які застосовуються до операндів.

Важливою властивістю аплікативної структури є те, що значення виразу єдиним чином визначається за значеннями його складових частин. Тобто, якщо деякий вираз двічі зустрічається в одному й тому ж самому контексті, він має одне й те ж саме значення в обох випадках. Мова, в якій ця властивість зберігається для всіх виразів, називається аплікативною або чисто функціональною мовою.

1.2 ЛІСП – основна мова функціонального програмування

Найбільш поширеною мовою функціонального програмування є ЛІСП, назва якого є абревіатурою виразу List Processing, тобто обробка списків [1,4]. З іншого боку, слово Lisp - означає "шепелявити". З появою цієї мови комп'ютер хоча і не став ще розмовляти але вже отримав можливість шепелявити по-людськи.

ЛІСП в 1960 р. запропонував Джон Маккарті для розробки програм спрямованих на розв'язання задач нечислового характеру. Такі задачі мають справу із складними структурами даних і базами знань, що містять правила прийняття рішень і різні інші об'єкти. Символьна обробка дозволяє ефективно працювати з такими структурами, як речення природної мови, значення слів і рішень, нечіткими поняттями і т.ін., та здійснювати на їх основі прийняття рішень, проведення міркувань і інші придатні людині засоби роботи з даними.

ЛІСП базується на алгебрі спискових структур, лямбда-численні та теорії рекурсивних функцій. Тривалий час ЛІСП використовувався лише вузьким колом спеціалістів у галузі штучного інтелекту. Широке розповсюдження він отримав наприкінці 70-х – на початку 80-х років з появою обчислювальних машин необхідної потужності та відповідного кола задач. Сьогодні ЛІСП є одним з головних інструментів штучного інтелекту. Він визнаний як одна з двох основних мов програмування

Міністерства оборони США і поступово витісняє другу мову програмування АДА. Наприклад, на мові програмування ЛІСП розроблено систему AutoCAD.

ЛІСП має кілька основних особливостей:

- форми подання програм і даних, які вони обробляють, в ЛІСПі є однаковими. І перші і другі подаються списковою структурою, що має однакову форму. Тобто, програми можуть обробляти і перетворювати інші програми і навіть самі себе, що відкриває дуже широкі можливості в програмуванні. Універсальний, за єдиним зразком і простий синтаксис ЛІСПу дозволяє просто визначати нові форми запису, подання і абстракцій. Завдяки цьому сама структура мови є дуже зручною для розширення, що не властиво традиційним мовам програмування;
- ЛІСП як правило є мовою-інтерпретатором, подібно до BASIC;
- списки, які подають в ЛІСПі програми і дані, складаються з спискових комірок, порядок розташування яких в пам'яті не є суттєвим. Структура списку визначається логічно, на основі імен символів та покажчиків, які створюють ланцюжки спискових комірок. Додавання або видалення елементів з списків може виконуватися без переносу списків до інших комірок пам'яті. Резервування та звільнення можуть здійснюватись динамічно, комірка за коміркою. При цьому користувачу не треба думати про урахування пам'яті. Система резервує і звільнює її автоматично, відповідно до потреб програми. Коли пам'ять вичерпується, запускається спеціальний прибиральник сміття, який повторно включає в роботу символи і списки, які вже не використовуються програмою;
- ЛІСП є безтиповою мовою. Тобто символи в ній не зв'язані за замовчуванням з будь-яким типом. Типи в ЛІСПі не зв'язані з іменами об'єктів даних, а супроводжують самі об'єкти. Тобто, змінні в різні моменти часу можуть представляти різні об'єкти, на відміну від типизованих мов програмування, де типи закріплюються за певними змінними (Фортран), або визначаються на етапі написання програми або її трансляції. В ЛІСПі тип визначається в ході виконання програми. Це забезпечує можливість різноманітного використання символів і гнучкої модифікації програм. Платою за динамічні типи є необхідність перевірки типів на етапі виконання програми. Але в ЛІСП-машині така перевірка здійснюється апаратно і не потребує багато часу;
- взагалі ЛІСП нагадує машинну мову тим, що дані і програма подаються в ній в однаковій формі. Вона чудово пристосована для написання інтерпретаторів і трансляторів, як для неї самої, так і для інших мов програмування. Програми, що написані на ЛІСПі, є

в багато разів коротшими ніж програми, що написані на процедурних мовах програмування. Наприклад, ядро ЛІСПу, яке написано на самому ЛІСПі, займає близько двох сторінок коду. Такий же обсяг потрібен і для інтерпретатора Прологу. Найбільш короткий інтерпретатор Прологу займає біля шістдесяти рядків ЛІСПу;

- ЛІСП має незвичайний синтаксис. Через велику кількість дужок LISP розшифровують як **L**ots of **I**diotic **S**illy **P**arentheses (Велика кількість Ідіотських Дурних Круглих дужок);
- В ЛІСПі для відображення функцій прийнята, як і в математиці, префіксна нотація. Але якщо в математиці ім'я функції передує аргументам, що розміщуються у дужках *(f(x), g(x,y), ...)*, то у ЛІСПі ім'я функції або дії передує аргументам і розміщується в середині дужок *((f x), (g x y), ...)*. Префіксна нотація використовується у ЛІСПі і для запису арифметичних виразів *((+ 2 3), (/ 6 2), ...)*, в той час як у математиці для цього використовується інфіксна форма запису ((2 + 3), (6/2), ...).

До основних переваг, які надає використання префіксної форми запису відносяться:

1) спрощення синтаксичного аналізу виразів, оскільки вже по першому символу поточного виразу система знає, з якою структурою має справу;
2) надання можливості запису функції у вигляді списків, тобто дані (списки) і програма (списки) подаються в єдиному форматі.

1.3 Типи даних в ЛІСПі

Основними типами даних в ЛІСПі є ***атоми*** і ***точкові пари***.

Атоми бувають двох типів: символьні і числові. Символьний атом може складатися з кількох символів, в тому числі і цифрових (але повинен мати хоча б одну букву), і завжди розглядається як єдине ціле. Символьний атом (або символ), це не ідентифікатор змінної як у звичайній мові програмування. Символ, як правило, позначає певний предмет, об'єкт, річ або дію. Символьний атом розглядається як єдине ціле. До символьних атомів застосовується лише одна операція – порівняння. Спеціальними атомами є атоми T і NIL –які водночас є логічними константами (істина і хибність, відповідно).

Точкова пара синтаксично визначається так [3]:

<точкова пара> ::= (<атом>.<атом>) | (<атом>.< точкова пара>) | (<точкова пара>.<атом>) | (<точкова пара> . < точкова пара>)

Тобто, точковими парами є такі вирази:

(Ivan.Maria), (sin.(Y.(PLUS.Y))), (A.((B.C).NIL)).

Використання дужок є дуже важливим, оскільки усуває можливу неоднозначність. Але дозволяється і бездужковий запис точкових пар. Наприклад, однаковий зміст мають такі записи: A.B.C і (A.(B.C)). Але точкову пару, першим елементом якої є пара A.B, а другим – атом C, вже неможливо записати без дужок: (A.B).C.

Атоми і точкові пари мають загальну назву – S-вираз. Спеціальним випадком S -виразу є список, який визначається так:

<список> =:: NIL | (< S -вираз>.<список>).

Якщо визначити NIL як пустий список, то це визначення можна переписати з використанням понять голови і хвоста списку:

<список> =:: <пустий список> | (<голова>.<хвіст>)
<пустий список> =:: NIL
<голова> =:: < S -вираз>
<хвіст> =: :<список>.

Згідно з цим визначенням будь-який непустий список є точковою парою. В якості прикладів списків можна навести такі S-вирази:

(КІТ.(ЛЮБИТЬ.(М'ЯСО.NIL))),
(ЄДИНИЙ.NIL).

Не є списками такі вирази:

(КІТ. (ЛЮБИТЬ.М'ЯСО)) ,
(ЄДИНИЙ.Т).

Список, який складається з елементів S1, S2,…,Sn відповідно до правил запису S-виразів, можна записати в такому вигляді:

(S1, S2,…,Sn).

Пустий список записується як ( ). Тобто маємо такі правила запису списків:

NIL ↔ ( )
(S1. (S2. (…,Sn.NIL))) ↔ (S1, S2,…,Sn)

S-вираз, який не є списком може бути записаний так:

(S1. (S2. (…,Sn.Sn+1))) ↔ (S1, S2,…,Sn.Sn+1)

Наведені правила можна підсумувати так. Точка, за якою безпосередньо іде ліва дужка, може бути опущена разом з цією дужкою і відповідною правою дужкою. Точка, за якою безпосередньо йде NIL також може бути опущена разом із самим NIL.

Розглянутий раніше список (КІТ.(ЛЮБИТЬ.(М'ЯСО.NIL))) можна відобразити як (КІТ ЛЮБИТЬ М'ЯСО).

Прикладами списків є і такі:

- (РОМАН БОЙКО 35 РОКІВ)

- ((ОЛЕГ 17) (МАРИНА 19) (ТЕТЯНА 6))
- ((МІЙ ДІМ) МАЄ (ВЕЛИКІ (СВІТЛІ ВІКНА))).

Отже в ЛІСПі список є послідовністю елементів (атомів або списків), які розділяються пропусками. Список береться в дужки. Список – це багаторівнева, або ієрархічна структура даних, в якій ліві і праві дужки знаходяться у строгій відповідності.

Елементом інших списків може бути і пустий список. Наприклад, () – пустий список; (NIL) – список, що складається лише з атома NIL, те ж саме, що і (()). Список ((NIL)) еквівалентний списку ((())). Список ((NIL)()) складається з двох пустих списків.

Звичайний запис S-виразів за допомогою дужок і точок може використовуватися в комбінації із записом у вигляді списків. Наприклад, структура виду (A.(B.(C.NIL))) може бути послідовно зведена до таких структур:
(A B.(C.NIL))) (A B C.NIL) (A B C). На рисунку 1.2 наведені всі S-вирази, які мають те ж саме значення, як і (A.(B.(C.NIL))):

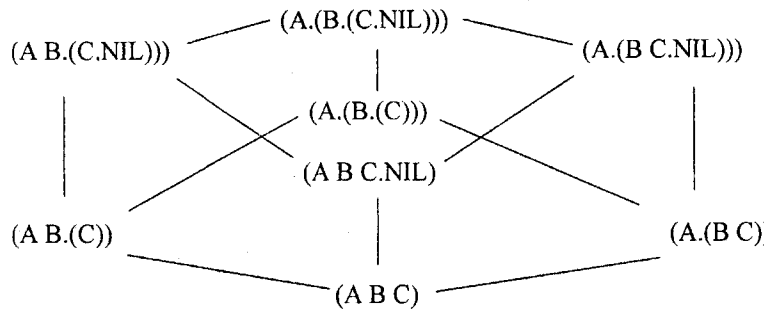


Рисунок 1.2 - Еквівалентні форми запису виразу (A.(B.(C.NIL)))

Розглянемо приклади запису даних у форматі S-виразів. Наприклад, необхідно подати дані для програми, яка обчислює суму послідовності цілих чисел. Скоріше за все в цьому випадку найзручніше подати дані у вигляді списку цілих чисел, наприклад:

(5 12 –3 4 67),

оскільки така структура буде найпростішою при обробці. Розглянемо можливі зображення алгебраїчних формул (враховуючи, що алгебраїчні операції задаються спеціальними символьними іменами). Наприклад формулу $x^2+2x-31$ можна зобразити у вигляді:

(ДОДАТИ (СТЕПІНЬ X 2) (ВІДНЯТИ (ПОМНОЖИТИ 2 X) 31))

або

(ВІДНЯТИ (ДОДАТИ (СТЕПІНЬ X 2) (ПОМНОЖИТИ 2 X)) 31)

та іншими можливими способами.

Шахова позиція може бути подана таким списком:

(((БІЛИЙ КОРОЛЬ) (D 7)) ((ЧОРНИЙ КОРОЛЬ) (B 1))
((БІЛИЙ ПІШАК (C 3)) ((ЧОРНИЙ ПІШАК) (A 2)))

Співвідношення основних типів ЛІСПу можна зобразити у вигляді діаграми, наведеної на рисунку 1.3:

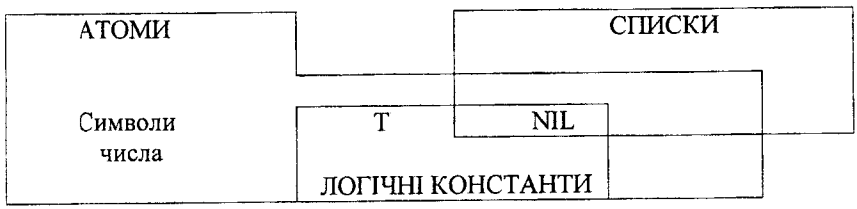


Рисунок 1.3 - Графічне зображення типів даних ЛІСПУ

1.4 Внутрішнє подання списків в ЛІСПі

Оперативна пам'ять комп'ютера, на якому працює ЛІСП, логічно розбивається на маленькі області – комірки. Кожен атом займає одну комірку. Списки є сукупностями атомів, що зв'язуються разом спеціальними елементами пам'яті, які називають списковими комірками (list cell, cons cell або просто cell). Спискова комірка складається з двох частин: полів CAR і CDR. Кожне з полів містить показчик (pointer), який може посилатися на іншу спискову комірку, або деякий ліспівський об'єкт, наприклад, атом. Показчики об'єднують комірки в ланцюжок, по якому можна з попередньої комірки потрапити до наступної і так далі, аж до атомарних об'єктів. Кожний відомий системі атом записується в пам'яті лише один раз (в окремих діалектах існують і винятки). Графічно спискова комірка позначається прямокутником, поділеним на поля CAR і CDR (рисунок 1.4).

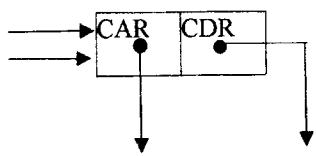


Рисунок 1.4 - Графічне зображення спискової комірки в ЛІСПі

Показчик зображується у вигляді стрілки, яка починається з однієї з частин прямокутника і закінчується на зображені іншої комірки або атому, на який посилається показчик. Спискова структура наочно відображає точкову пару. При цьому у першому полі міститься адреса першого елемента пари, а у другому полі – адреса другого елемента. У випадках, коли елементом точкової пари є, в свою чергу, інша точкова пара, стрілка на діаграмі вказує на відповідну комірку. Таким чином, наприклад, пара:

РОМАН.БОЙКО і пара PLUS. (1.X)

зображується так, як показано на рисунку 1.5:

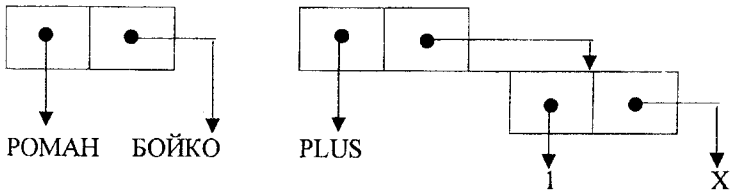


Рисунок 1.5 - Графічне зображення точкових пар РОМАН.БОЙКО і PLUS.(1.X)

За допомогою спискових комірок можна відобразити структури, які при запису за допомогою точок і дужок мають вигляд нескінчених. Наприклад, S-вираз виду

A. (B. (A. (B. (A. ..

буде відображений так, як показано на рисунку 1.6:

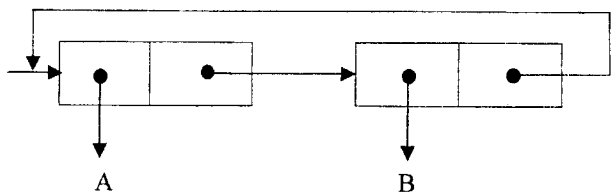


Рисунок 1.6 - Графічне зображення нескінченного списку

Звичайний список (A B C D) буде мати вигляд, відображений на рисунку 1.7.

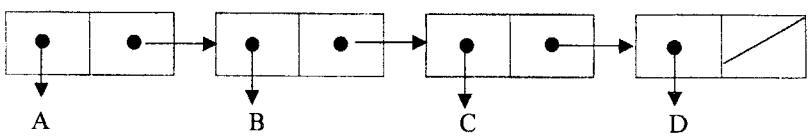


Рисунок 1.7 - Графічне зображення звичайного ЛІСП-списку

На одну комірку може вказувати більш ніж одна стрілка з інших спискових комірок, але з кожного поля комірки може виходити лише одна стрілка. Якщо на одну комірку вказує кілька покажчиків, це означає, що вона описує загальний підвираз. Наприклад у спискові

(ХТОСЬ ПРИЙШОВ ХТОСЬ ПІШОВ)

символ ХТОСЬ є загальним підвиразом на який посилаються покажчики з поля CAR першої та третьої комірок списку (рисунок 1.8):

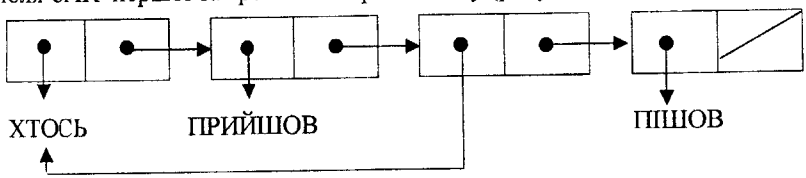


Рисунок 1.8 - Зображення списку з символами, що повторюються

Якщо елементами списку є не атоми, а підсписки, то на місці атомів будуть розташовані перші комірки підсписків. Наприклад, список

((B C) A B C)

буде відображено так, як показано на рисунку 1.9:

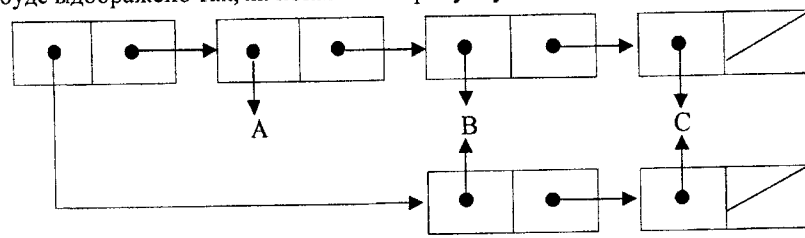


Рисунок 1.9 - Складний список, елементами якого є підсписки

Тобто, логічно ідентичні атоми містяться в системі один раз, але логічно ідентичні списки можуть бути подані різними списковими комірками. Наприклад, список (B C) може складатися і з тих же самих комірок що і підсписок (B C) списку (A B C), як показано на рисунку 1.10:

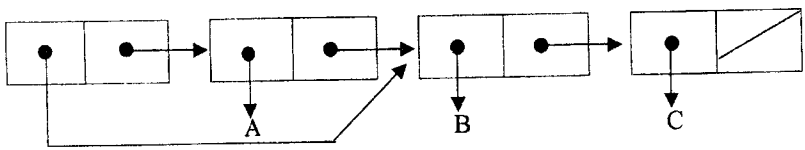
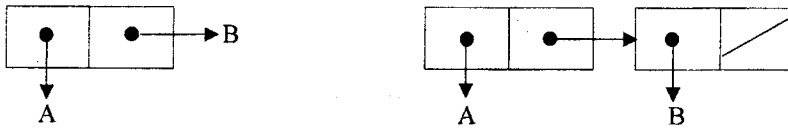


Рисунок 1.10 - Можливий варіант зображення списку ((B C) A B C)

На рисунку 1.11а і 1.11б наведені зображення, відповідно, точкової пари (A.B) та списку (A B):



а) б)

Рисунок 1.11 - Графічне зображення: а) точкової пари; б) списку

Повторимо, що будь-який список можна записати в точковій нотації:

(a) <=> (a.nil)
(a b c) <=> (a.(b.(c.nil)))

Вираз поданий у точковій нотації завжди можна привести до спискової нотації, якщо поле CDR є списком:

(a.(b c)) <=> (a b c)
(a.(b.c)) <=> (a b.c)

1.5 Контрольні питання

1. Визначіть поняття функції та наведіть її основні властивості.
2. Поясніть поняття композиції функцій. Наведіть власні приклади.
3. Що покладено в основу ЛІСПу?
4. Стисло охарактеризуйте основні особливості мови ЛІСП.
5. Які типи даних використовуються в ЛІСПі?
6. Наведіть синтаксичні визначення точкової пари та списку.
7. Поясніть поняття спискової комірки.
8. Подайте наступні спискові записи як точкові:
   a) (w (x));
   b) ((w) x);
   c) (nil nil nil);
   d) (v (w) x (y z));
   e) ((v w) (x y) z);
   f) (((v) w x) y z).
9. Подайте наступні точкові записи як спискові:
   a) (a . (b . (c . nil)));
   b) ((a . nil) . nil);
   c) (nil . (a . nil));
   d) (a . ((b . (c . nil)) . ((d . (e . nil)) . nil)));
   e) (a . (b . ((c . (d . ((e . nil) . (nil))) . nil)));
   f) ((a . (b . nil)) . (c . ((d . nil) . (e . nil)))).

2 БАЗОВІ ФУНКЦІЇ ЛІСПУ

2.1 Селекторні функції

Для роботи з S-виразами у ЛІСПі використовуються дуже прості базові функції, які називаються примітивами мови програмування. Можна говорити, що вони створюють систему аксіом мови, або алгебру обробки списків, до якої кінець-кінцем зводяться символьні обчислення. Базові функції обробки списків можна порівняти з основними операціями в арифметиці. Характерною рисою ЛІСПу є мала кількість базових функцій і їх простота.

Базові селекторні функції призначені для відокремлення певних частин S-виразів: функція CAR – здійснює вибір першого елемента точкової пари (або голови списку), а функція CDR – другого елемента точкової пари, або список, отриманий з початкового відкиданням його першого елемента. Тобто функції CAR і CDR доповнюють одна одну. Для атомів ці функції не визначені і їх застосування в даному випадку призведе до помилки. Оскільки в ЛІСПі аргументи функції записуються після її імені, CAR і CDR можуть бути визначені так:

(CAR (S1.S2))=S1;
(CDR (S1.S2))=S2.

Робота селекторів в графічному вигляді є очевидною. Результатом застосування функції CAR до списку буде вміст лівого поля спискової комірки, а застосування CDR – значення правого поля першої спискової комірки. Тобто,

(CAR (A B C))=(CAR (A.(B C)))=A;
(CDR (A B C))=(CDR (A.(B C)))=(B C).

Можна записати також:

(CAR <список>) = <голова >
(CDR <список>) = <хвіст >.

З цього правила є один важливий виняток – список NIL не містить жодного елемента, тому функції (CAR (NIL)) і (CDR (NIL)) є невизначеними, оскільки пустий список є атомом а не точковою парою.

Цікава історія виникнення імен функцій CAR і CDR. Автор ЛІСПу реалізовував свою першу систему на машині IBM 605. Для зберігання адреси голови списку він використовував регістр CAR (content of address registr), а для зберігання адреси хвоста списку - регістр CDR (content of decrement registr), на честь яких і дав відповідні назви створеним функціям.

За допомогою операції композиції з функцій CAR і CDR можна утворювати нові функції. Наприклад, селектор другого елемента списку

(CAR(CDR x)),

або функцію

(CAR(CAR(CAR x))),

де x – деякий S-вираз.

Функції, які є композиціями CAR і CDR застосовуються в ЛІСПі дуже часто, тому для них використовуються спеціальні позначення виду:

(CAR(CAR x)) = (CAAR x),
(CAR(CDR x))=(CADR x),
(CDR(CAR x))=(CDAR x),
(CAR(CAR(CDR(CDR x))))=(CAADDR x).

Максимальна кількість вкладених функцій не перевищує, як правило, 4.

2.2 Обчислення значень в ЛІСПі. Блокування QUOTE

Процес обчислень, який виконується ЛІСП-програмою прихований від користувача всередині так званої ЛІСП-машини. Користувачу доступні лише вхід до ЛІСП-машини, припустимо, через клавіатуру (вхідний потік), і вихід з неї, наприклад, відображення на моніторі (вихідний потік). Тобто елементарний цикл роботи ЛІСПу складається з операцій зчитування, обчислення і виведення результату у вихідний потік:

Read-eval-print цикл
loop { read
evaluate
prin}

В ЛІСПі завжди одразу читається, потім обчислюється (evaluate) значення функції і видається обчислене значення. Наприклад,

(* 5 3) ⇒ 15.

У функцію, що вводиться, можуть входити і функціональні підвирази:

(+ (* 5 3) (* 3 2)) ⇒ 21.

Програма на ЛІСПі складається з речень, під якими розуміються S-вирази або виклики функції на верхньому рівні. Наприклад, вираз (CAR(CDR x)) є реченням, оскільки виклик CAR здійснюється на верхньому рівні. Виклик CDR в даному прикладі є вкладеним і тому не є реченням. Одне речення може займати кілька рядків, а один рядок може містити кілька речень.

Підкреслимо, що виконання речення Ліспі-машиною завжди є обчисленням яке здійснюється в такому порядку. Спочатку зліва-направо обчислюються всі значення аргументів функції, після чого до обчислених

значень застосовується сама функція. Наприклад, в розглянутому вище прикладі спочатку обчислюються значення першого (15) і другого (6) аргументів, після чого до них застосовується функція додавання.

Але інколи бувають потрібні не обчислені значення виразів, а самі вирази. Наприклад, якщо безпосередньо ввести (+ 2 3) , то отримаємо значення 5. Але можна розуміти (+ 2 3) не як функцію, а як список. S-вирази, які не потрібно обчислювати, помічають для інтерпретатора апострофом
" ' " (quote). QUOTE є спеціальною функцією з одним аргументом, яка повертає в якості значення сам цей аргумент. Наприклад,

$$(+\ 2\ 3) \Rightarrow 5,$$

але

$$'(+\ 2\ 3) \Rightarrow (+\ 2\ 3);$$
$$'y \Rightarrow y;$$
$$(\text{quote}\ (+\ 2\ 3)) \Rightarrow (+\ 2\ 3),$$

замість апострофа можна безпосередньо використовувати ім'я quote;

$$''y \Rightarrow '(\text{quote}\ y);$$
$$'(a\ b\ '(c\ d)) \Rightarrow '(a\ b\ (\text{quote}\ (c\ d))),$$

апостроф автоматично перетворюється на quote.

Помітимо, що перед числовими константами ставити апостроф не треба, оскільки інтерпретатор вважає, що число і його значення збігаються. Але використання апострофу в даному випадку не заборонено. Не треба ставити апостроф і перед константами NIL і T.

Отже, для функції F1(F2(x)) спочатку обчислюється функція F2(x). Якщо її аргумент є атомом, то він може бути або константою, або ідентифікатором (символьним позначенням змінної, яка має власне значення). В першому випадку значенням аргументу є сама константа, а в другому – атом інтерпретується як змінна. Обчислення таких аргументів полягає в заміні ЛІСПом змінних на їх значення. Тобто, якщо аргументами функції CAR є змінні ПЕРШИЙ і ДРУГИЙ, то її виклик буде призводити до видачі повідомлення про помилку, доки ідентифікатори ПЕРШИЙ і ДРУГИЙ не будуть мати значень. Аналогічно, виклик атому НЕВИЗНАЧЕНИЙ, який є змінною з невизначеним значенням також викличе помилку. Але використання апострофа дозволяє в даному випадку працювати з ідентифікаторами не як із змінними, а як із символьними константами. Наприклад, виклик 'НЕВИЗНАЧЕНИЙ призведе не до помилки, а до виведення у вихідний потік рядка НЕВИЗНАЧЕНИЙ. Виклик (CAR 'ПЕРШИЙ.'ДРУГИЙ) дасть результат ПЕРШИЙ. Виклик CAR з попереднього прикладу можна переписати також у вигляді (CAR (QUOTE ПЕРШИЙ) . (QUOTE ДРУГИЙ)).

2.3 Конструктори

Якщо селектори здійснюють вибір частин з S-виразу, то для побудови S-виразу необхідно використовувати конструктор, який в ЛІСПі має ім'я CONS. Функція CONS будує новий список з переданих їй в якості аргументів голови і хвоста. Тобто, можна визначити її так:

$$(\text{CONS } (S1\ S2)) = (S1.S2).$$

Наприклад,

$$(\text{CONS 'a '(b c)}) \Rightarrow (a\ b\ c),$$
$$(\text{CONS '(a b) '(c d)}) \Rightarrow ((a\ b)\ (c\ d)),$$
$$(\text{CONS '(a b c) NIL}) \Rightarrow ((a\ b\ c)),$$
$$(\text{CONS NIL '(a b c)}) \Rightarrow (\text{NIL } a\ b\ c),$$
$$(\text{CONS NIL NIL}) \Rightarrow (\text{NIL}).$$

Як і селектори CONS є функцією, тому за допомогою операції функціональної композиції з неї можна отримувати нові функції, наприклад:

(CAR (CONS x y)),
(CDR (CONS xy))

та ін.

Селектори CAR і CDR є зворотними до конструктора CONS, як це показано на рисунку 2.1:

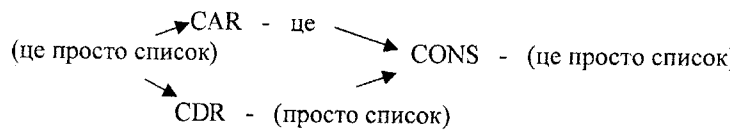


Рисунок 2.1 - Взаємозалежність селекторних і конструкторської функцій

Відмітимо, що CONS робить перший елемент головою другого елемента, який завжди є списком:

$$(\text{CONS 'a '(b c)}) \Rightarrow (a\ b\ c),$$
$$(\text{CONS '(a b) '(c d)}) \Rightarrow ((a\ b)\ c\ d),$$
$$(\text{CONS '(a b c) NIL}) \Rightarrow ((a\ b\ c)),$$

і дозволяє перетворювати елемент у список:

$$(\text{CONS 'a NIL}) \Rightarrow (a).$$

Відмітимо, що конструювання списків за допомогою CONS не є дуже зручним. Так, для отримання списку з чотирьох елементів, необхідно створити таку послідовність викликів CONS:

$$(\text{CONS x1 (CONS x2 (CONS x3 (CONS x4 NIL)))}) = (x1.(x2.(x3.(x4.\text{NIL})))).$$

В зв'язку з цим в ЛІСПі передбачено функцію LIST, значенням якої є список створений значеннями її аргументів, кількість яких не обмежується:

(LIST S1, ..., Sn)=(S1 ... Sn) = (S1.(S2.(...(Sn.NIL)))).

Таким чином, створити, наприклад, точкову пару

(ПЕРШИЙ.ДРУГИЙ)

можна за допомогою будь-якого з таких речень:

(CONS 'ПЕРШИЙ 'ДРУГИЙ);
'ПЕРШИЙ.'ДРУГИЙ;
'(ПЕРШИЙ.ДРУГИЙ);
(LIST 'ПЕРШИЙ 'ДРУГИЙ).

В третьому випадку точкова пара створюється не на кроці обчислення ЛІСП-машини, а вже на кроці зчитування.

Розглянемо ще одну функцію APPEND, яка дозволяє створювати новий список з двох чи більше вихідних списків:

_(append '(a) '(b) '(c d))
(a b c d).

APPEND об'єднує елементи не вносячи до них жодної зміни:

_ (append '(list) '('(a b) '(c)))
(list (quote (a b)) (quote (c))).

Звернемо увагу на відмінності, які існують між функціями CONS, LIST і APPEND:

_(list '(a b) '(c d)) ((a b) (c d))	LIST бере один або більше аргументів і створює список, поміщаючи кожний аргумент у дужки;
_(cons '(a b) '(c d)) ((a b) c d)	CONS завжди бере два аргументи і розміщує перший на початок другого;
_(append '(a b) '(c d)) (a b c d)	APPEND створює новий список, видаляючи дужки навколо аргументів і поміщаючи їх в один список.

Тобто CONS створює нову спискову комірку, поле CAR якої вказує на перший елемент, а поле CDR - на другий. Графічне зображення списку, створеного завдяки використанню функції (CONS 'x '(a b c)), показано на рисунку 2.2:

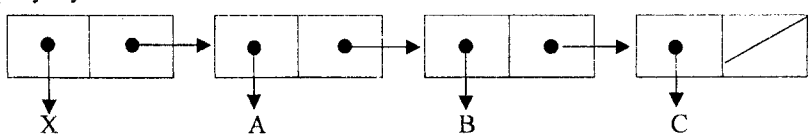


Рисунок 2.2 - Графічне зображення результату, що повертається функцією (CONS 'x '(a b c))

Результат дії функції LIST, наприклад (LIST 'a '(b c)) (a (b c)), може бути відображений у графічній форми так, як показано на рисунку 2.3:

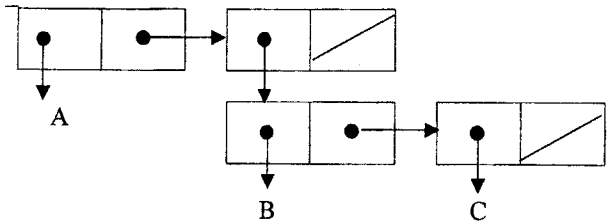


Рисунок 2.3 - Графічне зображення результату, що повертається функцією (LIST 'a '(b c)) (a (b c))

Він отримується в три кроки:

1) створюється спискова комірка для кожного аргументу функції;
2) у поле CAR поміщається покажчик на відповідний елемент;
3) у поле CDR поміщається покажчик на наступну спискову комірку.

2.4 Предикати

Для перевірки допустимості певних дій з функціями і запобігання помилкових ситуацій необхідно використовувати засоби розпізнавання виразів. Наприклад, перед застосуванням функції CAR необхідно впевнитися, що її аргумент є точковою парою. Крім того, розрізнювати S-вирази треба і в деяких інших ситуаціях. Наприклад, в задачі, в якій треба підрахувати кількість атомів в списку, треба вміти перевірити умову, чи є черговий елемент списком.

Функції, які дозволяють відрізняти S-вирази, називаються в ЛІСПі предикатами, оскільки визначають, чи має аргумент певну властивість, і повертають логічні значення хибність, тобто NIL, або істина, яке може бути подано атомом T або будь-яким іншим виразом, що відрізняється від NIL. Для ЛІСПу характерна "презумпція істинності". Тобто будь-який S-вираз відмінний від атому NIL інтерпретується як істина. Тому іноді кажуть, що ЛІСП використовує не двозначну, а півторазначну логіку.

Оскільки S-вирази можуть бути двох типів: атоми і точкові пари, існують предикати ATOM і PAIR, які визначаються так:

(ATOM x) = T,	якщо x – атом,	NIL – інакше;
(PAIR x) = T,	якщо x – точкова пара,	NIL – інакше.

Наприклад:

(ATOM 'x) ⇒ T;

(ATOM '(я програміст)) ⇒ NIL;

$$ATOM(CDR('(a\ b\ c)) \Rightarrow NIL$$

атом від списку – помилка;

$$(ATOM\ ()\ ) \Rightarrow T$$

() - це теж саме, що і NIL;

$$(ATOM\ '(NIL)) \Rightarrow NIL$$

це одноелементний список;

$$(ATOM\ (ATOM\ (+\ 2\ 3))) \Rightarrow T;$$

(+ 2 3)=5 це атом, і атом від Т – це теж атом.

Різні типи атомів, такі як: числа, рядки, NIL, ідентифікатори – можна розпізнавати за допомогою відповідних предикатів:

$$NUMBERP(x),\ STRINGP(x),\ NULL(x),\ IDP(x),\ CONSTANTP(x)$$

та ін.

В ЛІСПі використовуються також три предикати порівняння:

$$EQ,\ EQL,\ EQUAL.$$

Предикат EQ перевіряє тотожність двох символів або констант Т і NIL, але не чисел або списків. EQ все ж таки можна застосовувати до спискових або числових аргументів не отримуючи повідомлення про помилку, оскільки він не перевіряє логічної рівності чисел, рядків або інших об'єктів, а лише дивиться, чи подаються ліспівські об'єкти в пам'яті тією ж самою структурою. Наприклад:

$$(EQ\ 'x\ 'кіт) \Rightarrow NIL;$$
$$(EQ\ 'кіт\ (CAR\ '(кіт\ пес))) \Rightarrow T;$$
$$(EQ\ ()\ NIL) \Rightarrow T;$$
$$EQ\ (\ t\ 't) \Rightarrow T;$$
$$(EQ\ t\ (atom\ 'миша)) \Rightarrow T.$$

Але якщо списки логічно (зовнішньо) однакові, але складаються з різних ліспівських комірок, EQ поверне NIL. Не може він порівняти і числа, які є різними за типами, наприклад, 3 і 0,3E1.

Більш загальним є предикат EQL, який працює як EQ але додатково дозволяє порівнювати однотипні числа:

$$(EQL\ 3\ 3) \Rightarrow T;$$
$$(EQL\ 3.0\ 3.0) \Rightarrow T;$$
$$(EQL\ 3.0\ 0.3E1) \Rightarrow NIL.$$

Не слід використовувати EQL при порівнянні символів.

Незалежно від типів чисел і зовнішнього вигляду їх запису, значення Т у випадку рівності чисел поверне предикат = :

$$(=\ 3\ 3.0) \Rightarrow T;$$
$$(\ =\ 3.00\ 0.3E1) \Rightarrow T.$$

Предикат EQUAL працює як EQL, але крім того перевіряє рівність двох списків.

Хоча звичайну обробку списків можна завжди звести до трьох базових функцій CAR, CDR, CONS і двох предикатів ATOM і EQ, в ЛІСП включено багато додаткових базових функцій, які необхідні для спрощення опису різноманітних дій та ситуацій. Наприклад, перевірку чи не є аргумент пустим списком здійснює вбудована функція NULL:

(NULL'())⇒ T;
(NULL (CDDR'(a b c))) ⇒ NILL;
(NULL NIL) ⇒ T;
(NULL T) ⇒ NILL.

З останніх двох прикладів можна побачити, що NULL працює як логічне заперечення, для якого у ЛІСПі є спеціальна логічна функція (предикат) NOT:

(NOT (NULL NILL)) ⇒ NIL.

Функції NULL і NOT можна виразити через EQ:

(NULL x) ⇔ (EQ NIL x).

Для виділення n-го елементу зі списку у більшості діалектів ЛІСПу існують крім CAR CADR, CADDR, CADDDR та ін., спеціальні функції, такі як:

(NTH n список),

відокремлює n-ий елемент списку, починаючи з 0-го;

(FIRST список);
(SECOND список);
(THIRD (cons 1 (cons 2 (cons 3 NIL)))) ⇒ 3,
(FOURTH '(1 2 3)) ⇒ NIL,

та ін.

Останній елемент списку можна отримати за допомогою функції LAST(x).

Існує також багато арифметичних функцій і предикатів порівняння (наприклад (LESSP x y), (GREATERP x y) MAX і MIN з необмеженою кількістю аргументів.

2.5 Контрольні питання

1. Назвіть відомі Вам базові функції ЛІСПу.
2. Які типи мають аргументи базових функцій?
3. Які значення повертають базові функції?
4. Що таке предикат?
5. Назвіть основні відмінності предикатів EQ, EQL, EQUAL і =.

6. Чим відрізняються функції CONS і LIST?

7. Запишіть послідовності викликів CAR і CDR, які дозволяють виділити з наведених нижче списків символ «а». Спростіть ці виклики за допомогою функцій C...R.

а) (1 2 3 a 4)
б) (1 2 3 4 a)
в) ((1) (2 3) (a 4))
г) ((1) ((2 3 a) (4)))
д) ((1) ((2 3 a 4)))
е) (1 (2 ((3 4 (5 (6 a))))))

8. Яким буде значення кожного з таких виразів:

a) (ATOM (CAR (QUOTE ((1 2) 3 4))));
b) (NULL (CDDR (QUOTE ((5 6) (7 8)))));
c) (EQUAL (CAR (QUOTE ((7)))) (CDR (QUOTE (5 7))));
d) (ZEROP (CADDDR (QUOTE (3 2 1 0))));

9. Виконайте наведені обчислення за допомогою інтерпретатора ЛІСПу:

а) 3.234*(45.6+2.43)
б) 55+21.3+1.54*2.5432-32
в) (34-21.5676-43)/(342+32*4.1)

10. Визначте значення таких виразів:

а) '(+ 2 (* 3 5))
б) (+ 2 '(* 3 5))
в) (+ 2 (' * 3 5))
г) (+ 2 (* 3 '5))
д) (quote 'quote)
е) (quote 6)

3 ВИЗНАЧЕННЯ ФУНКЦІЙ

3.1 Ім'я і значення символу

Як і вираз, що є викликом функції, в разі відсутності у нього попереднього апострофа є значенням виразу а не самим виразом, так і атом можна використовувати для подання будь-яких значень. Наприклад, константи T і NIL представляють самі себе, як і числа. Тобто, якщо ми вводимо число, то інтерпретатор в якості відповіді повертає ту ж саму константу:

_t ⇒ T;
_'t ⇒ T;
_5 ⇒ 5.

Спочатку символи в ЛІСПі не мають значень. Значення мають тільки константи. Отож, спроба обчислити символ призведе до помилки. Значення символів зберігаються в комірках, що закріплені за кожним символом. Якщо в цю комірку занести значення, то символ буде зв'язано (bind) зі значенням. В процедурних мовах про це кажуть "буде присвоєно значення". При цьому для ЛІСП характерні такі особливості:

1) не оговорюється, що може зберігатися в комірці: ціле, атом, список, масив та ін. В комірці може зберігатися що завгодно;
2) з символом може бути зв'язана не тільки комірка із значенням, але і багато інших комірок, кількість яких не обмежена.

Для присвоєння значень (SET) або зв'язування символів використовуються три функції: SET, SETQ, SETF.

Функція SET зв'язує з символом деяке значення, попередньо обчисливши значення аргументів. При цьому вона обчислює обидва свої аргументи. Встановлений зв'язок залишається дійсним до кінця роботи, якщо цьому символу не буде присвоєно нове значення функцією SET:

_(SET 'a '(b c d)) ⇨ (b c d);
_a ⇨(b c d);
_(SET (CAR a) (CDR (o f g)) ⇨ (f g);
_a ⇨ (b c d);
_(CAR a) ⇨ b;
_b ⇨ (f g).

Інший приклад:

(SET 'функції '(CAR CDR CONS ATOM EQ)) ⇒
(CAR CDR CONS ATOM EQ).

Тепер інтерпретатор буде обчислювати значення символу ФУНКЦІЇ:

_функції ⇒ (CAR CDR CONS ATOM EQ).

Помітимо, що SET обчислює обидва аргументи. Тобто, якщо перед першим аргументом відсутній апостроф, то за допомогою функції SET можна присвоїти значення імені, яке отримується обчисленням. Наприклад, виклик

_(SET (CAR функції) '(зробити висновок))

присвоює змінній CAR вираз (ЗРОБИТИ ВИСНОВОК), оскільки виклик (CAR функції) повертає в якості значення символ CAR, який і використовується як фактичний аргумент виклику функції SET:

_(CAR функції) ⇒ CAR;
_CAR ⇒ (зробити висновок);
_функції ⇒ (CAR CDR CONS ATOM EQ).

Значення символів можна визначити за допомогою спеціальної функції SYMBOL-VALUE, яка повертає значення значенню символу, який є її аргументом:

_(SYMBOL-VALUE (CAR функція)) ⇒ (зробити висновок).

Якщо перед першим символом відсутній апостроф, то значення буде зв'язано зі значенням першого аргументу:

_(SET 'c (CAR '(f g))) ⇨ f;
_(SET c '(l m)) ⇨ (l m);
_f ⇨ (l m).

На значення символу можна послатися, записавши ім'я символу без апострофа.

Зв'язування імені символу без його обчислення виконує функція SETQ. Тобто, SETQ відрізняється від SET тільки тим, що обчислює лише свій другий аргумент:

_(SETQ d '(l m n)) ⇨ (l m n).

Про автоматичне блокування обчислення першого аргументу нагадує буква Q в імені функції. При використанні SETQ відпадає потреба в апострофі перед першим аргументом.

Перевірку зв'язаності атому можна здійснити за допомогою предикату BOUNDP, який буде істинним лише в тому випадку, коли з атомом зв'язане будь-яке значення:

_(BOUNDP 'без значення) ⇒ NIL;
_(BOUNDP 'функція) ⇒ T, BOUNDP ('t) ⇒T.

В ЛІСПі значення символу зберігається в комірці пам'яті (storage location), яка зв'язана з самим символом. Як на значення символів можна посилатися через їх імена, так і на комірки пам'яті можна посилатися викликом функції SYMBOL- VALUE або іншими способами, які залежать

від типу даних. Для занесення значення в комірку пам'яті існує узагальнена функція оновлення даних SETF, яка записує в комірку пам'яті нове значення: SETF(комірка пам'яті, значення). Наприклад,

(SETF список '(a b c)) ⇒ (A B C), _список ⇒ (A B C).

Розглянемо, як створюються зв'язки між змінними і списками. Наприклад, створення зв'язку між змінною Y та списком (a b c) за допомогою функції (setq y '(a b c)), можна графічно відобразити так, як показано на рисунку 3.1:

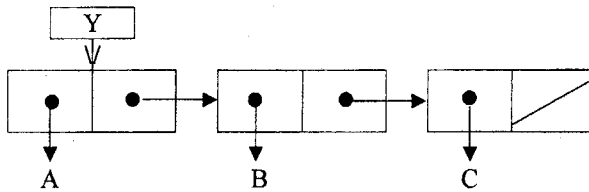


Рисунок 3.1 - Створення зв'язку між змінною Y та списком (a b c)

Зауважимо, що якщо у функції присвоювання список задається в явному вигляді, то під нього відводяться нові спискові комірки.

Використання змінної в функції, наприклад (setq x (cons 'd y)), забезпечує доступ до структури (рисунок 3.2):

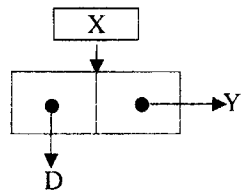


Рисунок 3.2 - Забезпечення доступу до структури даних

В даному випадку CONS не змінюючи структури збільшує список.

Ще раз повернемося до предикатів порівняння. Повторимо, що EQ перевіряє фізичну рівність списків, а EQUAL – логічну. Тобто, для EQ необхідно, щоб списки мали однакову структуру, а для EQUAL – однакові елементи (структура списку визначається списковими комірками).

Створимо дві структури:

_(setq lis1 '(a b))
_ (setq lis2 '(a b))

Тепер перевіримо рівність списків за допомогою функцій порівняння:

_ (setq lis3 lis1)

```
_(equal lis1 lis2)
t
_(equal lis1 lis3)
t
_(eql lis1 lis2)
nil
_(eql lis1 lis3)
t
_(eq lis1 lis2)
nil
_(eq lis1 lis3)
t
```

Але для окремого атома це не виконується, тобто нові комірки не відводяться:

```
_(setq m 'abc)
_(setq n 'abc)
_(eq m n)
t
```

Функції SET, SETQ, SETF відрізняються від раніше розглянутих тим, що крім значення вони мають побічний ефект. Побічний ефект функції полягає у створенні зв'язку між символом і його значенням, а значенням функції є значення, яке зв'язується. В ЛІСПі значення повертає будь яка функція, навіть та, основним призначенням якої є побічний ефект (наприклад такий, як зв'язування символу або виведення результатів на друк). Функції, що мають побічний ефект, називають ще псевдофункціями. Виклик псевдофункції може знаходитися на місці аргументу іншої функції (в процедурних мовах програмування це, як правило, не можливо):

```
_(LIST (+ (SETQ a 3) 4) a) ⇒ (7 3);  _a⇒3;
_(LIST  b (SETQ (b 3))) ⇒ ERROR UNBOUND ATOM,
```

аргументи обчислюються зліва-направо!

В практичному програмуванні псевдофункції корисні і часто необхідні, тоді як в чистому функціональному програмуванні вони не потрібні.

3.2 Способи визначення функції

Ми визначили функцію у вигляді *f(x1, x2, ..., xk)=e*, де *f* –функція, яка визначається, *x1, x2, ..., xk* – її параметри, *e* – вираз, який визначає функцію. Ім'я *f* розглядається надалі як глобальне (його можна викликати з будь якого місця), в той час як імена *x1, x2, ..., xk* є локальними (можуть використовуватися тільки у виразі *e*). Як же визначити функцію, яка б не

була глобальною? Вираз, значенням якого є функція являється λ-виразом. Помітимо, що визначення функції f, що приведено вище, присвоює f (або зв'язує з f) значення, яке є функцією. Значення, яке присвоюється наведеним вище визначенням, є значенням, яке подається за допомогою λ-виразу: $\lambda(x1, x2, ..., xk)e$. Функція є правилом для обчислення значення за деякими аргументами. λ-вираз містить в собі це правило, виділяючи імена параметрів (тут це $x1, x2, ..., xk$) і вираз, заданий в термінах цих параметрів. λ-вираз $\lambda(z)z+1$ обчислює функцію, яка, при виклику з конкретним значенням аргументу, видасть це значення збільшене на 1. В ЛІСПі лямбда вираз має такий вигляд:

(LAMBDA (x1, x2, ..., xk) fn).

Список, який утворюється параметрами, називається лямбда-списком. Тілом функції може бути довільний вираз, який може обчислюватися інтерпретатором ЛІСПу – наприклад, константа, пов'язана зі значенням символу, або композиція з викликів функцій. Наприклад, функція для обчислення суми квадратів двох чисел може бути визначена таким лямбда-виразом:

(lambda(x y) (+ (* x x) (* y y))).

Лямбда вираз є визначенням обчислень і параметрів функції у чистому вигляді, без фактичних параметрів або аргументів. Для застосування такої функції необхідно підставити на місце функції лямбда-визначення:

(лямбда-визначення a1, a2, ..., an),

де an – вирази, що задають фактичні параметри, які обчислюються за загальними правилами.

Наприклад дію додавання чисел 2 і 3:

_(+2 3)⇒5,

можна записати через виклик лямбда-виразу:

_((lambda (x y) (+ x y)) 2 3) ⇒ 5.

Наступний виклик будує список з аргументів A і B:

_((lambda (x y) (cons x (cons y nil))) 'a 'b) ⇒ (A B).

Така форма виклику називається лямбда-викликом. Обчислення лямбда-виклику здійснюється в два етапи. Спочатку обчислюються значення фактичних параметрів і відповідні формальні параметри зв'язуються з отриманими значеннями. Цей етап називається зв'язуванням параметрів. На другому етапі з урахуванням нових зв'язків обчислюється вираз, який є тілом лямбда-виразу і отримане значення повертається в якості значення лямбда-виклику. Формальним параметрам на закінчення

повертаються ті зв'язки, які в них можливо були до обчислення лямбда-виклику. Весь цей процес називається лямбда-перетворенням.

Лямбда-виклики можна вільно об'єднувати між собою та іншими виразами. Вкладені лямбда-виклики можна ставити як на місце тіла лямбда-виразу, так і на місця фактичних параметрів. Наприклад, вкладення лямбда-виклику в тіло лямбда-виразу можна здійснити так:

_((lambda (y) ((lambda (x) (list y x)) 'ВНУТРІШНІЙ)) 'ЗОВНІШНІЙ) ⇒
(ЗОВНІШНІЙ, ВНУТРІШНІЙ).

Приклад лямбда-виклику, який є аргументом іншого виклику, може бути таким:

__((lambda (x) (list 'ДРУГИЙ x))((lambda (y) (list y)) 'ПЕРШИЙ)) ⇒
(ДРУГИЙ, (ПЕРШИЙ)).

Запам'ятаємо, що лямбда-виклик без аргументів (фактичних параметрів) є лише визначенням, але не виразом, який можна обчислити. Сам по собі він не сприймається інтерпретатором:

_ (lambda (x y) (cons x (cons y nil))) ERROR: undefined function LAMBDA.

Лямбда-вираз є функцією без імені, яка зникає одразу після обчислення. Її важко викликати знову, оскільки до неї неможливо звернутися за ім'ям, хоча раніше вираз був доступним як списковий об'єкт. Тим не менше такі безіменні функції дуже зручно використовувати, наприклад, при передачі функції в якості аргументу іншій функції або при формуванні функції за результатами обчислень, тобто при синтезі програм.

Для того, щоб надати визначеній функції ім'я, використовується функція DEFUNE:

(DEFUNE ім'я лямбда-список тіло), або (DEFUN ім'я лямбда-вираз).

Для зручності використання тут вилучають зовнішні дужки лямбда-виразу і сам атом лямбда. Наприклад:

(defun list ((lambda(x y) (cons x (cons y nil))))) ⇒
(defun list (x y) (cons x (cons y nil)))

DEFUN поєднує символ з лямбда-виразом внаслідок чого він починає представляти (іменувати) обчислення, яке визначає даний лямбда-вираз. Значенням цієї форми є ім'я нової функції:

(defun list (x y) (cons x (cons y nil))) ⇒ LIST, (list 'a 'b) ⇒ (A B).
(defun lambdap (вираз) (eq (car вираз) 'lambda),

перевіряє на лямбда-вираз:

(lambdap '(list 'a 'b)) ⇒ NIL.

_(defun проценти (частина чого) (* (/частина чого) 100)) ⇒ ПРОЦЕНТИ
(проценти 4 20) ⇒ 20.

Визначимо ще одну функцію insert-second з двома аргументами new і oldlist, яка буде вставляти на другу позицію будь-якого списку елемент new. Тіло функції може мати такий вигляд:

(cons (car oldlist) (cons new (cdr oldlist))).

Отож, загальне визначення функції буде таким:

```
( defun insert-second (item oldlist) (cons (car oldlist) (cons item (cdr oldlist))).
                    _(insert-second 'b '(a c d ))
                    (a b c d).
```

Підкреслимо, що в ЛІСПі передача параметрів здійснюється тільки за значенням (call by value), тоді як в традиційних мовах програмування використовується також передача параметрів за посиланням (call by reference). Отже, формальний параметр у ЛІСП-функціях зв'язується з тим же значенням, що і значення фактичного параметра. Зміна значення формального параметра під час обчислення функції ніяк не впливають на значення фактичних параметрів. Тобто, інформацію можна передавати тільки процедурам, але не з них. Після обчислення функції створені на цей час зв'язки параметрів ліквідуються і здійснюється повернення до того стану, який був до виклику функції. Параметри функції є локальними змінними і мають значення тільки всередині функції.

Отже, передача параметрів в ЛІСПі здійснюється в основному за значенням. Параметри в ЛІСПі використовуються переважно лише для передачі даних у функцію, а результати повертаються як значення функції, а не як значення параметрів, що передаються за посиланням.

Наприклад, визначимо функцію *f*, що змінює значення *x*:

```
_( defun f ( x ) ( setq x ' new ) )
f
```

Присвоїмо *x* значення *old* за допомогою функції SETQ:

```
_( setq x ' old )
old
```

Перевіримо, яке значення має змінна *x*:

```
_x
old
```

Під час обчислення функції змінна *x* тимчасово прийме значення *new*.

```
_( f x )
new
```

Ще один приклад:

```
_( defun double (num) (* num 2))
double
```

```
_( setq num 5 )
5
_( double 2 )
4
_num
5
```

Тобто, під час обчислення функції було здійснено тимчасовий зв'язок символу *num* із значенням 2, але по закінченні обчислень цей зв'язок було ліквідовано і символу *num* повернено його попередній зв'язок із значенням 5.

Але якщо в тілі функції є змінні, що не входять до числа її формальних параметрів (так звані вільні змінні), то їхнє значення залишається в силі і після її обчислення.

Наприклад:

```
_( defun f1 ( y ) (setq x 3 ) )
f1
_( f1 5 )
3
_x
3
```

Розглянемо приклад написання корисної програми на ЛІСПі. Нехай треба написати програму розрахунку опору електричного ланцюга, наведеного на рисунку 3.3:

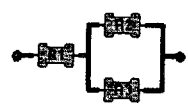


Рисунок 3.3 - Приклад електричного ланцюга для розрахунку опору

Визначимо функцію SERIAL для розрахунку опору послідовно з'єднаних резисторів R=R1+R2:

(defun SERIAL (R1 R2) (+ R1 R2)).

Функція PARALLEL для паралельного з'єднання резисторів R=(R1*R2)/(R1+R2) може мати такий вигляд:

(defun PARALLEL (R1 R2) (/(* R1 R2) (+ R1 R2)).

Виконаємо за допомогою розроблених функцій розрахунок ланцюга, де R1=R2=R3-10:

```
_(SERIAL 10 (PARALLEL 10 10))
15
```

Розглянемо більш складний ланцюг з параметрами R1=R2=R3=R4=10, який наведено на рисунку 3.4:

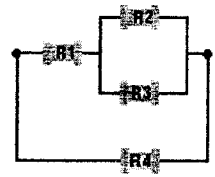


Рисунок 3.4 - Приклад складнішого електричного ланцюга для розрахунку опору

Отримаємо:

_(PARALLEL 10 (SERIAL 10 (PARALLEL 10 10)))
6

Аналогічні розрахунки можна провести для електричного ланцюга будь-якої конфігурації.

3.3 Контрольні питання

1. Що таке символ?
2. Чим відрізняються функції SET, SETQ, SETF?
3. Які особливості властивостей символів Ви знаєте?
4. Що таке лямбда-вираз?
5. Для чого використовується функція DEFUN?
6. Складіть список студентів своєї групи:
   (ПІБ ПІБ ... ПІБ)

6.1 Для кожного студента:

а) з використанням функції LIST складіть такі списки:
для самого студента - (дата народження), (адреса), (середній бал з лекційних занять), (середній бал з практичних занять), (середній бал з лабораторних робіт). Для батьків - (ПІБ), (дата народження), (адреса), (місце роботи).

б) з використанням функцій CONS і SETQ об'єднайте отримані списки і присвойте їх у вигляді значень символам, що означають ПІБ кожного студента:
ПІБ ст. - (((дата народження ст.) (адреса ст.)((середній бал(до десятих) з лекційних занять) (середній бал з практичних занять) (середній бал з лабораторних робіт))) (((ПІБ батька) (дата народження батька) (адреса) (місце роботи батька)) ((ПІБ матері) (дата народження матері) (адреса) (місце роботи матері)))).

6.2 Для довільно вибраних студентів порівняйте за допомогою базових функцій:

а) рік народження;

б) успішність (з урахуванням того, що число, що характеризує середній бал, може бути як цілим, так і ні);

в) з'ясуйте, чи не є вони родичами;

г) з'ясуйте, чи живуть вони з батьками.

7. Визначте за допомогою лямбда-виразу функцію, що обчислює:
   a) +y-x*y;
   b) x*x+y*y;
   c) x*y/(x+y)-5*y;
8. Визначте з використанням базових функцій такі функції: (NULL x), (CADDR x) і (LIST x1 x2 x3). Використайте імена NULL1, CADDR1 і LIST1, щоб не перевизначати одноіменні вбудовані функції системи.
9. З використанням композиції напишіть функції, які здійснюють обернення списку з 2, 3, ... , n елементів.
11. З використанням композиції предикатів, що описані вище, і логічних зв'язок, побудуйте функцію, яка перевіряє, чи є її аргумент:
   a) списком з 2, 3, ... n елементів;
   b) списком з 2, 3, ... n атомів.
12. Напишіть функцію:
   a) таку, що P(n) для будь-якого цілого n є списком з трьох елементів, а саме: квадрата, куба і четвертої степені числа n;
   b) для двох аргументів значенням якої є список з двох елементів (різниці і залишку від ділення);
   c) таку, що A(n) є списком (The answer is n). Тобто, значенням (A 12) буде (The answer is 12);
   d) семи аргументів, значенням якої служить сума всіх семи аргументів.
13. Для кожної з наступних умов визначте функцію одного аргументу L, яка має значення T, якщо умова виконується, і NIL у випадку:
   a) n-ий елемент L є 12;
   b) n-ий елемент L є атомом;
   c) L має не більше як n елементів (атомів або підсписків).
14. Напишіть функцію, яка вводить фразу природною мовою і перетворює її на список.
15. Напишіть функцію, яка запитує у користувача ПІБ студента з групи (список групи складений раніше) і видає такі дані про нього:
   a) рік народження;
   b) середній бал;
   c) батьків;
   d) списки властивостей, що були присвоєні йому раніше.
16. Напишіть функцію:
   a) від одного аргументу (ПІБ будь-якого студента), що заміщує в списку з даними про нього (написаному раніше) підсписки з середніми балами на списки властивостей.

4 КЕРУЮЧІ СТРУКТУРИ

4.1 Введення і виведення інформації

Розглядаючи приклади функцій і програм ми здійснювали введення і виведення даних тільки у діалозі з інтерпретатором. Якщо не використовувати спеціальні команди введення, то дані ЛІСП-функції можна передавати тільки через параметри і вільні змінні. Відповідно, без використання спеціальних функцій виведення, результат можна отримати лише через кінцеве значення виразу. Разом з тим, на практиці часто виникає необхідність вводити початкові дані і видавати повідомлення, і тим самим управляти і отримувати проміжні результати під час обчислень, як це робиться у інших мовах програмування.

ЛІСП-функція читання READ відрізняється від операторів введення -виведення інших мов програмування тим, що обробляє при введені не окремі елементи даних, а весь вираз, що вводиться, як єдине ціле.

Виклик функції здійснюється у вигляді (трохи спрощено):

(READ)

Як тільки інтерпретатор зустріне речення READ, він призупиняє обчислення доти, поки користувач не введе будь-який символ або речення:

```
_(READ)
(вираз, що вводиться)          - значення, що вводиться
(ВИРАЗ, ЩО ВВОДИТЬСЯ)          - значення, що повертається
```

READ ніяк не вказує на очікування інформації. Програміст повинен сам повідомити про це за допомогою функцій виводу, які ми розглянемо далі. READ є псевдофункцією, побічний ефект якої полягає у введенні інформації.

Якщо зчитаний вираз необхідний для подальшого користування, READ слід зробити аргументом певної форми, яка зв'яже отримане значення:

```
_(setq x '(read))
(+ 1 2)                        - вираз, що вводиться
(+ 1 2)                        - значення
_x
(+ 1 2)
_(eval x)
3
```

Ще один приклад:

```
(defun tr (arg) (list (+ arg (read))(read)))
_(tr 8)
```

```
14
cat
(22 cat)
```

Для виводу виразів в ЛІСПі використовують декілька функцій: PRINT, PRIN1, PRINC.

Функція PRINT є функцією з одним аргументом, що спочатку обчислює значення аргументу, а потім виводить це значення. Функція PRINT перед виведенням аргументу переходить на новий рядок, а після нього виводить пропуск. Таким чином, значення виводиться завжди у новий рядок.

```
_(PRINT (+ 1 2))
3                                   ; виведення
3                                   ; значення
```

PRINT є псевдофункцією, у якої є як побічний ефект, так і значення. Значенням функції є значення її аргументу, а побічним ефектом - друк цього значення.

Функції PRIN1 і PRINC працюють так само, як PRINT, але не переходять на новий рядок і не виводять пропуск:

```
(PRIN1 5) ⇨ 55
(PRINC 4) ⇨ 44
```

Обома функціями можна виводити крім атомів і списків і інші типи даних які ми розглянемо пізніше:

```
_(PRIN1 «CHG») ⇨ «CHG»«CHG»
_(PRINC «tfd») ⇨ tfd«tfd»          ; виведення без лапок,
                                    ; результат - значення аргументу.
```

За допомогою функції PRINC можна одержати більш приємний вигляд. Вона виводить ліспівські об'єкти в тому ж вигляді, що і PRIN1, але перетворює деякі типи даних у більш просту форму.

4.2 Умовні речення

У традиційних мовах програмування існують засоби управління обчислювальним процесом, які забезпечують можливості організації розгалуження і циклів. У ЛИСПІ для цього використовуються спеціальні керуючі структури – речення (clause). Зовні речення записуються як виклики функцій: перший елемент – ім'я, інші – аргументи. Наслідком обчислень є значення. Відмінність від виклику функцій полягає тут у використанні аргументів.

Структури управління розподіляються на групи. Однією з таких груп є структури для розгалуження обчислень, до яких належать умовні речення COND, IF, WHEN, UNLESS.

У більшості діалектів ЛІСПу використовується конструкція IF…THEN…ELSE. Але найбільш поширеною є функція COND, яка відповідає цій конструкції, але має невизначену кількість аргументів, кожен з яких має бути списком з двох елементів:

(COND (предикат1 вираз1) (предикат2 вираз2)…(предикатn виразn) (T zz)),

або

(COND (x1 y1) (x2 y2)…(xn yn) (T zz)),

що відповідає виразу

IF x1 THEN y1 ELSE IF x2 THEN y2 … ELSE zz.

Значення COND визначається за такими правилами:

1) предикати < предикат-і > обчислюються послідовно, зліва направо, доки не зустрінеться вираз, значенням якого не є **NIL**;
2) обчислюється вираз-і, що відповідає даному предикату-і. Отримане значення повертається в якості значення всього речення **COND;**
3) якщо істинне значення відсутнє, то значенням **COND** буде **NIL.**

Отже, умовне речення COND завжди повертає значення, що відповідає предикату із значенням відмінним від NIL. Але якщо істинний предикат в COND відсутній, то значенням COND буде NIL. Тому рекомендується в якості останнього предикату використовувати символ T і тоді відповідний йому вислідний вираз завжди буде обчислюватись у випадках, коли ніяка інша умова не виконується. Наприклад, визначимо за допомогою COND такий вираз:

(DEFUN TEST (l) (COND ((NULL l) 'пусто) ((ATOM l) 'атом) (t 'список)).

Перевіримо його роботу:

_(test '(a b c)) ⇒ список
_(test '(atom '(a t o m))) ⇒ пусто.

У COND може бути відсутня вислідна дія <вираз - і>, або замість однієї дії може виконуватися їх певна послідовність:

(COND (предикат1) (предикат2 вираз2)
(предикат3 вираз31 вираз 32 вираз 33 ….) (T zz)).

Якщо будь-якому предикату (предикат 1), не ставиться у відповідність жодна дія, то в якості результату виконання речення COND, при істинності предикату, повертається само значення предикату. Якщо ж одному предикату (предикат 3) відповідає не одна дія, а послідовність дій, то при істинності предикату всі ці дії обчислюються послідовно зліва направо і результатом речення COND буде значення останнього виразу послідовності.

Речення COND є найбільш загальною умовною структурою. Використовуються в ЛІСПі і інші, знайомі з Паскаля умовні конструкції:

(IF умова, то - форма, інакше - інша форма) ⇔
(COND (умова то - форма) (T інакше – інша форма)

Наприклад,

_(IF (atom t) 'атом 'список)
атом

Умовні речення WHEN і UNLESS є також окремими випадками COND (як і IF):

(WHEN умова вираз1 вираз2 ...) ⇔
(UNLESS (NOT умова) вираз1 вираз2 ...) ⇔
(COND (умова вираз1 вираз2 ...)).

Можна використовувати і речення CASE, подібне тому, що використовується в мові програмування ПАСКАЛЬ:

(CASE ключ (список ключів1 m11 m12 ...)
(список ключів2 mi21 mi22 ...)
...).

В CASE спочатку обчислюється значення ключа, після чого це значення порівнюється з елементами списків ключів СПИСОК КЛЮЧІВ і, при знаходженні в списку значення ключової форми, починається обчислення відповідних форм m1, m2, ..., значення останньої з яких повертається в якості значення всього речення CASE.

Відмітимо, що в ЛІСП функція трактується як правило, що дозволяє ставити у відповідність аргументам результат. Але для умовного виразу цього не достатньо. Хоча він і має три аргументи, але реально обчислюється значення лише двох з них: спочатку першого, а потім, в залежності від умови – другого або третього. Тому можна записати таке речення:

(IF (ATOM x) THEN x ELSE (CAR x)).

Дійсно, якщо x є атомом, то вираз (CAR x) повинен був би дати помилку. Але цього не станеться якщо x дійсно є атомом, тому що функція (CAR x) буде обчислюватися тільки в тому випадку, якщо x не є атомом, а в цьому випадку функція (CAR x) завжди визначена.

До іншої групи керуючих структур, а саме до групи об'єднання послідовних обчислень, відносяться речення PROG1, PROG2, PROGN. Вони дозволяють працювати з декількома обчислювальними формами:

(PROG1 форма1 форма2 ... формаN)
(PROG2 форма1 форма2 ... формаN)
(PROGN форма1 форма2 ... формаN)

Ці спеціальні форми послідовно обчислюють свої аргументи й в якості значення повертають значення першого (PROG1), другого (PROG2) або останнього (PROGN) аргументу.

_(PROG1 (SETQ x 1) (SETQ y 5)) ⇨ 1
_(PROGN (SETQ j 8) (SETQ z (+x j))) ⇨ 9

В ЛІСПі часто використовується і так званий неявний PROGN, тобто обчислюється послідовність форм, а в якості значення береться значення останньої форми:

(defun < ім'я функції > < список параметрів >
< форма1 форма2 .... формаN >)

При цьому тіло функції складається з послідовності форм, що відображають послідовність дій, а в якості значення функції приймається значення останньої форми.

Наприклад, визначимо функцію, яка друкує список, вводить два числа і друкує їх суму:

```
( defun print-sum ( )                    ; функція без аргументів
( print '( type two number ) )
( print ( + ( read ) ( read ) ) ) )
_( print-sum )
( type two number ) 3 4
7
7
```

В ЛІСПі також широко використовуються логічні зв'язки OR і AND, які є функціями з невизначеним числом аргументів:

$$(\text{OR } x_1\ x_2 \ldots x_n)=\text{NIL},$$

якщо всі x_i дорівнюють NIL, інакше x_l, де x_l – перший по порядку аргумент, значення якого відмінне від NIL;

$$(\text{AND } x_1\ x_2 \ldots x_n)=\text{NIL},$$

якщо хоча б один аргумент дорівнює NIL, інакше – x_n.

4.3 Локальне присвоювання змінних

У випадках, коли використовуються обчислення послідовності форм, буває зручно вводити локальні змінні, які зберігаються до закінчення обчислень. Це робиться за допомогою речення LET.

В загальному випадку LET записується в такому вигляді:

(LET ((var1 знач1) (var2 знач2)...) форма1 форма2 ... формаN)

LET обчислюється за такими правилами:

1) локальні змінні var1, var2, ...varM зв'язуються одночасно зі значеннями знач1, знач2, ..., значM;
2) послідовно обчислюються аргументи форма1, форма2, формаN;
3) в якості значення речення береться значення останнього аргументу (неявний PROGN);

4) після виходу з речення зв'язки змінних var1, var2, ...varM знищуються.

Речення LET зручно використовувати коли потрібно тимчасово зберігати проміжні значення.

Розглянемо функцію RECTANGLE, що має один аргумент - список з двох елементів, які задають довжину і ширину прямокутника. Функція визначає і друкує значення площі і периметра прямокутника:

```
(defun rectangle (dim) (let ((len (car dim)) (wid (cadr dim)))
(print (list 'area (* len wid))) (print (list 'perimeter (* (+ len wid) 2))))))
                    _(rectangle '(4 5))
                    (area 20)
                    (perimetr 18)
                    (perimetr 18)
```

Можна спочатку визначити площу:

```
(defun rectangle (dim) (let ((len (car dim)) (wid (cadr dim))
(area (* len wid)) (print ( 'area area)) (print (list 'perimeter (* (+ len wid)))))))
```

Але звернення

```
_(rectangle '(4 5))
```

поверне помилку, оскільки значення area не визначене.

Необхідно використати речення LET* , в якому значення змінних задається послідовно:

```
(defun rectangle (dim)
(let* ((len (car dim)) (wid (cadr dim)) (area (* len wid)))
(print (list 'area area))
(print (list 'perimeter (* (+ len wid) 2))))) .
```

Іноді буває необхідно вийти з тіла функції, що є послідовністю форм, не дійшовши до останньої форми. Це можна зробити з використанням речень PROG і RETURN, які використовуються разом.

Нехай, наприклад, необхідно написати функцію, яка вводить два значення. Якщо це числа, функція друкує їхні суму та різницю. Якщо хоча б один з аргументів не є числом, програма друкує nil:

```
(defun s-d () (prog (x y)                          ; локальні змінні
(print '(type number))
(setq x (read)) (and (not (numberp x)) (return nil))
(print '(type number)) (setq y (read))
(and (not (numberp y)) (return nil))
(print (+ x y)) (print (- x y))))

_(s-d)
(type number) 8
      (type number) (1)
      nil
```

Return повертає результат для Prog. Якщо return не зустрівся результатом prog буде nil .

```
_(s-d)
(type number)8
(type number)1
9
7
nil
```

В разі відсутності локальних змінних пишуть (prog ()...).

Розглянемо додаткові функції друку. PRINT друкує значення аргументу без пропуску і переводу на інший рядок:

```
_(progn (print 1) (print 2) (print 3))
123 7
```

Послідовність знаків поміщена у дужки називається у ЛІСПі рядком "string".

Рядок є спеціальним типом даних в ЛІСПі. Це атом, який не може бути змінною. Як у числа значенням рядка є сам рядок.

```
_"(+ 1 2)"
"(+ 1 2)"
```

Рядки зручно використовувати для виведення за допомогою оператора PRINC.

PRINC друкує рядки без дужок "", а аргумент без пропуску і переводу рядка.

Наприклад,

```
_(progn (setq x 4) (princ "x =") (prin1 x) (princ "m"))
x=4m,
```

де " m " - значення останнього аргументу.

PRINC забезпечує гнучке виведення.

Функція TERPRI здійснює переведення рядка. В неї немає аргументів і як значення вона повертає nil.

```
_(progn (setq x 4) (princ "xxx ") (terpri) (princ "xox "))
xxx
xox
" xox"
```

4.4 Циклічні речення

Циклічні речення в ЛІСПі виконуються або за допомогою ітераційних (циклічних) речень або рекурсивно. Розглянемо спочатку циклічні речення [7].

Речення LOOP реалізує нескінченний цикл:

(LOOP форма1 форма2 .....)

в якому форми обчислюються доти, доки не зустрінеться явний оператор завершення RETURN.

Визначимо функцію add-integer, яка бере один аргумент, що є додатним цілим і повертає суму всіх чисел від 1 до цього числа:

1+2+3+4+ ... +N

```
_(add-integers 4)
10                                   ;  1+2+3+4=10

(defun add-integers (last) (let (( count 1) (total 1))
(loop
(cond (( equal count last) (return total)))
(setq count (+ 1 count)) (setq total (+ total count)))))
```

В якості ще одного прикладу числової ітераційної функції визначимо функцію, що виконує множення двох цілих чисел через додавання. Тобто множення *x* на *y* виконується додаванням *x* самого до себе *y* разів. Наприклад, 3 x 4 можна записати як: 3 + 3 + 3 + 3 = 12. Отже:

```
_(int-multiply 3 4)
12

(defun int-multiply (x y)
(let ((result 0)( count 0))
(loop
(cond (( equal count y) (return result)))
(setq count (+ 1 count)) (setq result (+ result x)))))
```

З наведених прикладів отримуємо загальну форму для числової ітерації:

```
(defun < ім'я-функції > < список-параметрів >
(let (< ініціалізація змінної індексу >
< ініціалізація змінної наслідку >)
(loop
(cond < перевірка індексу на вихід > (return наслідок))
< зміна змінної лічильника >
< інші дії в циклі, в тому числі зміна змінної наслідку >)))
```

Визначимо в якості ще одного прикладу функцію factorial:

```
_(factorial 5)
120                ;          1 x 2 x 3 x 4 x 5 = 120

(defun factorial ( num ) (let ((counter 0)( product 1))
(loop
(cond (( equal counter num) (return product)))
```

```
(setq counter (+ 1 counter)) (setq product (* counter product )))))
```

Приклад,

```
(progn (setq x 0)
(loop (if ( = 3 x) (return 't) (print x))
(setq x (+ 1 x))))
0
1
2
t
```

Визначимо функцію, що використовує друкування і введення. Функція без аргументів зчитує послідовність чисел і повертає суму цих чисел коли користувач вводить не число. Функція має друкувати: "Enter the next number:" перед введенням кожного числа:

```
_( read-sum)
Enter the next number: 15
Enter the next number: 30
Enter the next number: 45
Enter the next number: stop
90
```

```
(defun read-sum () (let ((input) (sum 0))
(loop
(princ "Enter the next number:")
(setq input (read))

(cond (( not (numberp input)) (return sum)))
(setq sum (+ input sum)))))
```

Розглянемо приклад застосування LOOP для ітераційної обробки списків. Припустимо, що нам необхідна функція double-list, яка приймає список чисел і повертає новий список в якому кожне число подвоєне.

```
_(double-list '(5 15 10 20))
(10 30 20 40)
```

Визначення функції може мати такий вигляд:

```
(defun double-list (lis)
(let ((newlist nil))
(loop
(cond ((null lis) (return newlist)))
(setq newlist (append newlist (list (* 2 (car lis)))))
(setq lis (cdr lis )))))
```

Порядок виконання обчислень ілюструється в таблиці 4.1.

Таблиця 4.1 - Порядок виконання обчислень функції double-list

	List	Newlist
Початковий стан	(5 15 10 20)	()
Ітерація 1	(15 10 20)	(10)
Ітерація 2	(10 20)	(10 30)
Ітерація 3	(20)	(10 30 20)
Ітерація 4	()	(10 30 20 40)
Результат		(10 30 20 40)

Найбільш загальним реченням для реалізації циклічних обчислень у ЛІСПі є речення DO:

(DO ((var1 знач1 крок1) (var2 знач2 крок2)....)
(умова-закінчення форма11 форма12...)
форма21 форма22 ...)

Функція виконує обчислення за такими правилами:
1) спочатку локальним змінним var1 ..varn присвоюються початкові значення знач1..значn. Змінним, для яких не задані початкові значення присвоюється nil;
2) перевіряється умова закінчення. Якщо вона виконується, обчислюється форма11, форма12... В якості значення береться значення останньої форми;
3) якщо умова не виконується, то обчислюються форма21, форма22...;
4) на наступному циклі змінним vari присвоюються одночасно нові значення, що визначаються формами кроків і все повторюється.

Наприклад:

```
_(do ((x 1 ( + 1 x)))
(( > x 10) ( print 'end))
( print x))
```

буде друкувати послідовність чисел, яка закінчиться словом end.

Можна порівняти ітераціонні обчислення з використанням LOOP і DO.

Напишемо за допомогою LOOP функцію list-abs, яка бере список чисел і повертає список абсолютних величин цих чисел:

```
(defun list-abs (lis)
(let ((newlist nil))
(loop
(cond (( null lis ) (return (reverse newlist))))
(setq newlist (cons (abs (car lis)) newlist))
(setq lis (cdr lis )))))
_(list-abs '(-1 2 -4 5))
```

Те ж саме, тільки з використанням DO:

```
(defun list-abs (lis)
(do ((oldlist lis (cdr oldlist))
(newlist nil (cons (abs (car oldlist)) newlist)))
((null oldlist) (reverse newlist)))))
```

Бачимо, що одночасно можуть змінюватися значення кількох змінних:

```
_( do (( x 1 (+ 1 x))
( y 1 (+ 2 y)) ( z 3))                ;   значення не
змінюється
(( > x 10) ( print 'end))
(princ " x=") ( prin1 x)
(princ " y=") ( prin1 y)
(princ " z=") ( prin1 z) (terpri))
```

Є можливість і реалізації вкладених циклів:

```
_( do (( x 1 (+ 1 x)))
(( > x 10))
( do (( y 1 (+ 2 y)))
(( > y 4))
( princ " x= ") ( prin1 x) ( princ " y= ") ( prin1 y)
(terpri) ))
```

За допомогою речення DO зручно здійснювати обробку списків. Напишемо, наприклад, функцію, яка буде читати елементи з клавіатури і об'єднувати їх у список. Введення буде закінчено, коли буде введений останній елемент end:

```
(defun appen-read ()
(do (( x ( list ( read)) ( append x (list (read)))))
((equal (last x) '(end))); ???? '(end)
(print x)))
(appen-read)
```

В разі необхідності можна використовувати DO* аналогічно LET*.

З метою повторення обчислень задану кількість разів замість DO зручно використовувати функцію DOTIMES:

(DOTIMES (var num форма-return) (форма-тіло))

де,

var - змінна циклу,

num - форма, що визначає кількість циклів,

форма-return - результат, який повинен бути повернений.

При обчисленні функції перш за все обчислюється num-форма, внаслідок чого отримують ціле число - count. Потім var змінюється від 0 до count (за винятком count) і, відповідно, кожен раз обчислюється форма-

тіло. Останнім обчислюється форма-return. Якщо форма-return відсутня, повертається nil.

Наприклад,

```
_(dotimes ( x 3 )
( print x))
0 - o
1
2
t
_(let ((x nil))
(dotimes (n 5 x)
(setq x (cons n x))))
( 4 3 2 1 0)
```

4.5 Контрольні питання

1. Чим відзрізняються основні функції виведення?
2. Що повертає в якості значення функція READ?
3. Особливості функцій, що змінюють структуру?
4. Для чого використовується речення LET?
5. В чому полягає різниця між реченнями LET і LET*?
6. В чому полягає різниця між функціями COND і IF?
7. В чому полягає різниця між функціями PROG1 і PROGN?
8. Чому використання операторів передачі управління не бажане? Чим їх можна замінити?
9. Запишіть такі лямбда-виклики з використанням форми LET і обчисліть їх на комп'ютері:

```
a) ((LAMBDA (x y) (LIST x y)
      '(+ 1 2) 'c);
b) ((LAMBDA (x y) ((LAMBDA (z) (LIST x y z)) 'c)
      'a 'b);
c) ((LAMBDA (x y) (LIST x y))
      ((LAMBDA (z) z) 'a)
      'b).
```

10. Напишіть за допомогою композиції умовних виразів функції від чотирьох аргументів AND4(x1 x2 x3 x4) і OR4(x1 x2 x3 x4), які збігаються з функціями AND і OR від чотирьох аргументів.
11. Нехай L1 і L2 - списки. Напишіть функцію, яка повертає T, якщо два елементи цих списків відповідно дорівнюють один одному, а в іншому випадку - NIL.

12. Напишіть умовний вираз (з використанням COND), який:
    а) повертає NIL, якщо L є атомом, в іншому випадку - T;
    б) повертає для списку L, що складається з трьох елементів, перший з цих трьох елементів, що є атомом, або список, якщо в списку відсутні атоми.
14. З використанням речень COND або CASE визначте функцію, яка повертає в якості значення столицю заданої аргументом країни.
15. Напишіть з використанням умовного речення функцію, яка повертає з трьох числових аргументів значення більшого та меншого за величиною числа.
16. Запрограмуйте за допомогою речення DO функцію факторіал.
17. Запишіть за допомогою речення PROG функцію (аналог вбудованої функції LENGTH), яка повертає в якості значення довжину списку (кількість елементів на верхньому рівні).
18. З використанням функції COND, напишіть функцію, яка запитує у користувача ПІБ двох студентів з групи (список групи складено раніше) для яких:
    а) порівнює рік народження і видає результат (хто є старшим або що вони однолітки);
    б) порівнює середній бал і видає повідомлення за результатами порівняння;
    с) перевіряє родинні зв'язки (якщо студенти мають тих самих батьків, то вони є родичами) і видає повідомлення про це.
19. Напишіть подібні функції, але вже з використанням функції IF.

Для двох останніх завдань виведення інформації здійснювати за допомогою функцій PRINT, PRIN1, PRINC.

5 РЕКУРСІЯ

5.1 Числова рекурсія

Під рекурсією розуміють прийом, коли програма викликає сама себе чи то безпосередньо, чи через підпрограми, які вона викликає. Тобто, функція є рекурсивною, якщо у її визначенні міститься виклик самої себе. Рекурсія є основним і найбільш ефективним способом організації обчислень у функціональному програмуванні і в ЛІСПі.

Наприклад, напишемо рекурсивну програму для обчислення факторіалу. Відмітимо два факти:

1. Якщо N=0, то факторіал 0 дорівнює 1;
2. Якщо N >0, то факторіал N (добуток чисел від 0 до N) дорівнює N помножити на факторіал (N-1)! (добуток чисел від 0 до N-1).

Схема таких обчислень приведена на рисунку 5.1.

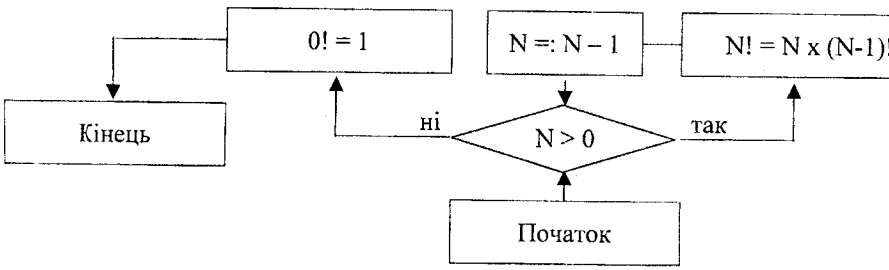


Рисунок 5.1 - Схема рекурсивного обчислення факторіалу N

Ці два факти можна безпосередньо перевести у визначення функції:

__(defan factorial (N) (cond ((= N 0) 1) (t (* N (factorial (- N 1)))))).

Тут виконується перевірка, чи дорівнює N нулю. Якщо так – функція повертає 1. Інакше функція викликає сама себе для обчислення факторіалу (N-1)! і перемножує N на цей факторіал.

Подивимося як працює функція, для чого простежимо кілька викликів.

Як працює (factorial 0) очевидно. Функція повертає 1. Ця гілка називається термінальною, оскільки функція повертає значення без рекурсивного виклику.

При виклику (factorial 1) обчислення спрямовується по другій гілці, яка називається рекурсивною, оскільки викликає сама себе. В цьому випадку виконуємо (* 1 (factorial 0)) і отримуємо результат 1. При виклику (factorial 2) по рекурсивній гілці виконується (* 2 (factorial 1)) і повертається 2. Якщо подивитися далі, побачимо таку послідовність дій:

(factorial 3) викликає (factorial 2),
(factorial 2) викликає (factorial 1),
(factorial 1) викликає (factorial 0).
Після того як виклик (factorial 0) поверне 1,
(factorial 1) поверне 1,
(factorial 2) поверне 2,
(factorial 3), який є викликом верхнього рівня, поверне остаточний результат 6.

Розглянемо процедуру трасування розробленої функції:

```
> (trace factorial)    -    виклик трасування
(FACTORIAL F)    - режим трасування ввімкненний
> (factorial 3)      -    виклик функції
  Entering: FACTORIAL, Argument list: (3)
    Entering: FACTORIAL, Argument list: (2)
      Entering: FACTORIAL, Argument list: (1)
      Existing: FACTORIAL, Value: 1
    Existing: FACTORIAL, Value: 2
  Existing: FACTORIAL, Value: 6
6
>
```

5.2 Правила побудови рекурсивних функцій

Розглянутий простий приклад ілюструє кілька правил запису рекурсивної функції [2]:

1. Термінальна гілка необхідна для закінчення виклику. Без неї рекурсивний виклик був би нескінченним. Термінальна гілка повертає результат, який є основою для обчислення результатів рекурсивних викликів.
2. Після кожного виклику функцією самої себе, обчислення повинні наближатися до термінальної гілки. У нашому випадку кожний виклик зменшує значення N, отже ми мали гарантію, що на певному кроці буде здійснено виклик (factorial 0). Завжди повинна бути впевненість, що рекурсивні виклики ведуть до термінальної гілки.
3. Треба уявляти, що простежити обчислення у рекурсії надзвичайно складно. Дуже важко уявити собі дію рекурсивних функцій. Це практично неможливо для складних функцій.

Отже, треба навчитися писати рекурсивні функції без того, щоб чітко уявляти порядок їх обчислення. Зробимо це для функції factorial.

При написанні рекурсивної функції необхідно уміти планувати термінальні і рекурсивні гілки. Розглянемо приклад такого планування для рекурсивної функції factorial.

- Крок 1. Закінчення (термінальна гілка)
  1N= 0 - аргумент
  (factorial 0) = 0 – значення
- Крок 2. Рекурсивна гілка
  рекурсивні відношення між (factorial N) і (factorial (- N 1))
  2а. Приклади рекурсії
  (factorial N) (factorial (- N 1))
  (factorial 5) = 120 (factorial 4) = 24
  (factorial 1) = 1 (factorial 0) = 1.
  2б. Характеристичне рекурсивне відношення (factorial N) може бути отримане з (factorial (- N 1)) множенням на N.
  - Планування термінальної гілки.
  При написанні рекурсивної функції необхідно вирішити, коли функція може повернути значення без рекурсивного виклику.
  - Планування рекурсивної гілки.
  В даному випадку ми викликаємо функцію рекурсивно із спрощеним аргументом і використовуємо результат для розрахунку значення при поточному аргументі.

Для цього необхідно вирішити:

1. Як спрощувати аргумент, щоб спостерігати його крок за кроком до кінцевого значення.
2. Як побудувати форму, що має назву рекурсивного відношення, яка зв'язує правильне значення поточного виклику із значенням рекурсивного виклику. В нашому випадку це (factorial N) і (factorial (- N 1)). Іноді таке відношення знайти дуже просто. Якщо ж це не так, то треба виконати таку послідовність кроків:
   а) визначити значення деякого простого виклику функції і відповідного ій рекурсивного виклику;
   б) визначити співвідношення між парою цих функцій.

Наприклад, визначимо функцію (power), яка бере два числових аргументи M і N і обчислює значення M у степені N.

Складемо спочатку рекурсивну таблицю.

- Крок 1. Термінальна гілка.
  N = 0 – аргумент
  (power 2 0) = 1 – значення
- Крок 2. Рекурсивна гілка
  Рекурсивні відношення між (power M N) і (power M (- N1))
  2а. Приклади рекурсії
  (power 5 3) = 125 (power 5 2) = 25
  (power 3 1) = 3 (power 3 0) = 1
  2б. Характеристичне рекурсивне відношення
  (power M N) може бути отримане з (power M (- N 1)) множенням на M.

Отже, відповідно функція буде мати такий вигляд:

(defun power (M N) (cond (= N 0) 1) (t (* M (power M (- N 1))))).

В розглянутих прикладах визначення функцій здається дуже простим, але вже для чисел Фібоначі рекурсивне визначення має явно більш "прозорий" вигляд за будь-яке інше подання:

$$F1=1,\ F2=1,\ F_n=F_{n-1}+F_{n-2},\ \text{при } n>2.$$

Саме використання рекурсії дозволяє дуже легко (практично дослівно) запрограмувати обчислення за рекурсивними формулами і зберегти при цьому високу прозорість програм.

Для чисел Фібоначчі отримаємо таке визначення функції:

(defun fibb (N) (cond ((eq1 N) 1) ((eq2 N) 1)
(t (+ (fibb(- N 1) (fibb(- N 2)))))).

Але рекурсія має і недолики. Один з них чітко видно на прикладі програми fibb, яка робить багато зайвих викликів, по кілька разів обчислюючи одне й те ж саме число (рисунок 5.2):

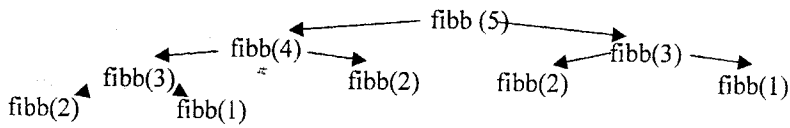


Рисунок 5.2 - Неефективність рекурсивної функції fibb

В той же час можна написати програму, яка б обмежувалася лише n викликами.

5.3 CDR - рекурсія

Ми розглянули приклади рекурсивної обробки чисел. Але коли інформація подана у вигляді списку, то з'являється необхідність рекурсивної обробки списків. Основною рекурсією над списками є CDR-рекурсія. Логіка і структура М-рекурсії схожі з числовою рекурсією.

Напишемо, наприклад функцію length1, яка бере один аргумент – список, і повертає довжину списку. Аргументом, що послідовно спрощується, в даному випадку буде список. Спрощення списку можна здійснити якщо на кожному кроці брати в якості аргументу його хвіст (cdr lis). Останнім значенням аргументу буде nil.

Складемо рекурсивну таблицю для (length1 lis).

- Крок 1. Термінальна гілка
  (length1 nil) = q - значення
- Крок 2. Рекурсивна гілка
  Рекурсивні відношення між
  (length1 lis) і (length1 (cdr lis))
  2а. Приклади рекурсії

(length1 '(a b c d)) = 4 (length1 '(b c d)) = 3
(length1 '(d)) = 1 (length1 '(nil)) = 0

2б. Характеристичне рекурсивне відношення (length1 lis) може бути отримане з (length1 (cdr lis)) додаванням 1 (якщо будемо вважати, що довжина голови, як одного елемента, дорівнює 1, то довжина списку буде дорівнювати 1+довжина хвоста).

Визначення функції:

```
_(defun length (lis) (cond ( (null lis) 0) (t (+ 1 (length ( CDR lis)))))).
```

Розглянемо приклад трасування функції:

```
> (trace length )    -    виклик трасування
(LENGTH  F)    - режим трасування ввімкнено
> (length (á b c)      -    виклик функції
  Entering: LENGTH, Argument list: ((a b c))
    Entering: LENGTH, Argument list: ((b c))
      Entering: LENGTH, Argument list: ((c))
        Entering: LENGTH, Argument list: (NIL)
        Existing: LENGTH, Value: 0
      Existing: LENGTH, Value: 1
    Existing: LENGTH, Value: 2
  Existing: LENGTH, Value: 3
3
>
```

Ясно, що дана програма підраховує тільки кількість елементів списку верхнього рівня.

5.4 Рекурсія з кількома рекурсивними гілками

Ми розглянули випадки рекурсії з однією термінальною і однією рекурсивною гілками. Але у визначені рекурсивної функції може бути і кілька термінальних гілок (рисунок 5.3). Наприклад, дві термінальні гілки можуть з'явитися в тому випадку, коли ведеться пошук цілі у послідовності значень і ми бажаємо отримати результат, як тільки знайдемо ціль:

- Гілка 1. Ціль знайдено і необхідно повернути відповідь.
- Гілка 2. Ціль не знайдено і елементів більше немає.

Наприклад, визначимо функцію greatenum, яка має два аргументи – список і число. Функція має повертати перше число зі списку, яке перевищує задане. Якщо таке число відсутнє, повертається число, що було задане в якості другого аргументу.

```
(defun greaternum (lis num) (cond ((null lis) num) ((> (car lis) num) (car lis))
                                  (t (greaternum (cdr lis) num)))).
```

Зауважимо, що порядок запису гілок у рекурсивному визначенні є істотним.

Рисунок 5.3 - Рекурсія з кількома рекурсивними та термінальними гілками

Кілька рекурсивних гілок може знадобитися, наприклад, якщо функція обробляє всі елементи в структурі, але використовує деякі елементи не так, як інші. В такому випадку складаються два рекурсивних відношення.

Визначимо функцію negsums, яка отримує список чисел і повертає список, який містить тільки від'ємні числа (0 є додатним числом). Наприклад (negsums '(-1 5 -6 0 2)) повертає (-1 -6):

- Крок 1. Термінальна гілка
  (negsums nil) = nil
- Крок 2. Рекурсивна гілка
  Рекурсивні відношення між (negsums l) і (negsums (cdr l))
  1. (car l) < 0
     - Приклади рекурсії
     (negsums '(-5 2)) = (-5)
     (negsums '(2)) = nil
     (negsums '(-5 3 –6 0)) = (-5 -6)
     (negsums '(3 -6 0)) = (-6)
     - Характеристичне рекурсивне відношення
     (negsums l) може бути отримане з (negsums (cdr l)) за допомогою виклику (cons (car l) (negsums (cdr l))
  2. (car l) >= 0
     - Приклади рекурсії
     (negsums '(1 -5 3 -6 0)) =(-5 -6)
     (negsums) '(- 5 3 -6 0)) = (-5 -6)
     - Характеристичне рекурсивне відношення
     (negsums l) можна отримати з (negsums (cdr l)) за допомогою виклику (negsums l) = (negsums (cdr l))

Отже, визначимо необхідну функцію:

```
(defun negsums (l) (cond ((null l) nil)
                         ((< (car l) 0) (cons (car l) (negsums (cdr l)))) (t (negsums (cdr l))))))
```

А тепер розглянемо рекурсивну функцію для визначення загальної довжини списку, тобто і основного списку і всіх його підсписків:

```
(defun fulllength (U) (cond( (atom U) 1)
                            (t (+ (fulllength (CAR U))  (fulllength ( CDR U))))))
```

Помітимо, що остання функція не враховує особливу роль атома NILL і тому видає довжину списку на 1 більшу, ніж кількість явно заданих в ньому елементів.

5.5 Діаграми трасування рекурсивних функцій

Визначимо функцію об'єднання списків APPEND, яка отримує на вході два списки і повертає один список, в якому спочатку послідовно розташовані елементи першого списку, а потім елементи другого списку.

Оскільки процедура має два аргументи, треба розглянути всі можливі випадки, в яких один з аргументів або обидва рівні нулю, а також коли жоден з аргументів нулю не рівний:

Випадок 1 u=nill

Підвипадок 1.1. v=nill (append u v)=nill

Підвипадок 1.2. v=/=nill (append u v)=v

Випадок 2 u=/=nill

Підвипадок 2.1. v=nill (append u v)=u

Підвипадок 2.2. v=/=nill нехай (append (cdr u) v)=z, тоді (append u,v)=(cons(car u),z).

Це можна записати на ЛІСПі в такому вигляді:

```
(defun fullappend (u v) (cond ((null u)  (cond ((null v) nil) (t v))) ((null v) u)
                              (t (cons (car u) (fullappend (cdr u) v))))).
```

Помітимо, що в усіх випадках окрім 2.2 можна одразу записати результат і лише у випадку 2.2 використовується рекурсія. Якщо відомо, що деякий список не дорівнює nill, то необхідно подивитися, чи не можна написати неявну відповідь у термінах cdr від цього списку. У випадку 2.2. є три кандидата на обчислення значень рекурсивним викликом:

(append (cdr u) v), (append u (cdr v)), (append (cdr u) (cdr v)).

В нашому випадку ми обрали перший з них.

Якщо ж помітити, що при u=nill функція (append u v)=v незалежно від того чи рівний v nill, чи ні, то можемо записати більш коротку версію функції append:

```
(defun shortappend (u v) (cond ((null u)  v) ((null v) u)
```

(t (cons (car u) (shortappend (cdr u) v))))).

Якщо ж врахувати, що при u=/=nill

(append u v) = (cons (car u) (append (cdr u) v))

незалежно від того, чи виконується y=nill чи ні, отримуємо найбільш компактний результат:

_(defun append (u v) (cond ((null u) v) (t (cons (car u) (append (cdr u) v))))).

Помітимо, що якщо перевагою компактного запису є його мала довжина, то перевагою визначення fullappend є явне перерахування всіх випадків, а перевагою shortappend - більш висока ефективність порівняння з append, оскільки тут не виконуються зайві обчислення у випадку u=/=nill, v=nill.

Для визначення деяких функцій часто буває зручно вводити допоміжні функції, які б зробили процес розв'язання задачі простішим і ефективнішим. У випадку append такою допоміжною функцією могла б стати функція app, яка б з'єднувала списки u і v у припущенні, що v=/=nill. При цьому ми б отримали таке визначення:

_(defun superappend (u v) (cond ((null v) u) (t (app u v)))).
_(defun app (u v) (cond ((null u) v) (t (cons (car u) (app (cdr u) v))))).

При даному визначенні поєднуються переваги короткого запису і високої ефективності функції.

Приклад трасування для функції append ('(a b) '(c d)) наведений нижче:

```
__(trace append)
(append)
(append ('(ab) '(c d))
append:
   x=(a b)
   y=(c d)
      append:
         x=(b)
         y=(c d)
            append:
               x=nill
               y=(c d)
            append=(c d)
      append=(b c d)
append=(a b c d)
(a b c d)
```

Для кращого розуміння роботи функції можна також будувати діаграму трасування рекурсивного виклику. Для функції append така діаграма наведена на рисунку 5.4:

_(defun append (u v) (cond ((null u) v) (t (cons (car u) (append (cdr u) v)))))

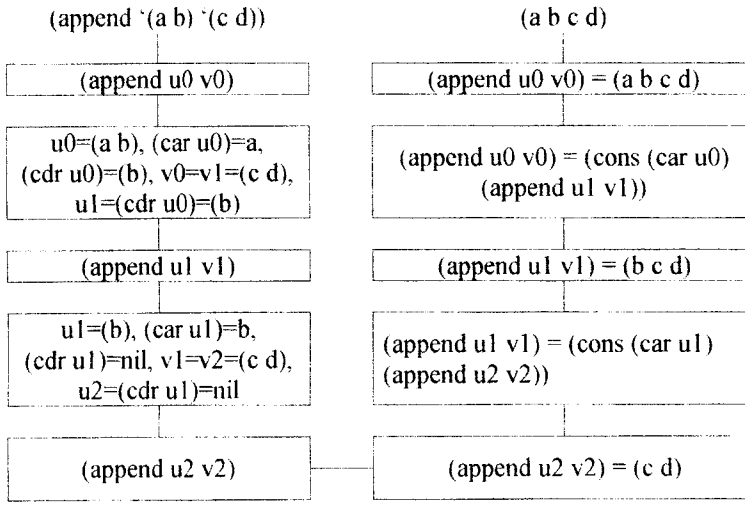


Рисунок 5.4 - Діаграма трасування рекурсивної функції APPEND

Розглянемо також приклад побудови функції (reverse u), що повертає список в якому елементи розташовані в порядку, зворотному до того, що був у вхідному спискові. Якщо елементами списку є підсписки, то елементи цих підсписків не обертаються, тобто наша функція буде працювати лише з елементами верхнього рівня вхідного списку.

Проаналізуємо можливі випадки. Якщо u=nill, то отримаємо (reverse u)= nill. Якщо ж u=/=nill, то можна використати (reverse(cdr u)). Крім того треба мати функцію, яка б розташувала результат (car u) у кінці списку (reverse (cdr u)). З цієї точки зору корисною може бути функція (add u x), яка отримує список u і елемент x і робить x останнім елементом списку u. При цьому необхідна функція буде мати такий вигляд:

_(defun revers (u) (cond ((null u) nill) (t (add (reverse (cdr u)) (car u))))).

Тепер побудуємо функцію add. Ця функція має два аргументи, перший з яких є списком, а другий – S-виразом. Тому аналіз випадків стосується в даному випадку лише першого аргументу. Можемо записати:

Випадок 1. Якщо u=nill, то (add u x)=(cons x nill)
Випадок 2. Якщо u=/=nill, і (add (cdr u) x)=z, то (add u x)=(cons (car u) z)

Таким чином отримуємо:

_(defun add(u x) (cond ((null u) cons(x nill)) (t (cons (car u) (add (cdr u) x))))).

Функцію (add u x) можна було б визначити і через функцію append: (add u x)=append(u cons(x, nil)). При цьому отримаємо:

_(defun reverse1(u) (cond ((null u) nil)
(t (append (reverse1 (cdr u)) (cons (car u) nil))))).

5.6 Використання параметрів накопичування

Спробуємо по іншому побудувати функцію reverse, визначивши при цьому допоміжну функцію із зайвим параметром, який буде використовуватися для накопичення необхідного результату. Такий метод досить часто використовується у функціональному програмуванні [6,7].

Визначимо функцію (rev u v) з тією властивістю, що u є списком який має бути обернений, а v – параметр, який накопичує обернений список. Маємо визначення функції:

_(defun rev (u v) (cond ((null u) v) (t (rev (cdr u) (cons (car u) v))))),

через яку можна визначити функцію reverse:

_(defun reverse (u) (rev u nil)).

Розглянемо як працює така функція. Коли викликається функція (rev u v), список v вже накопичив в зворотному порядку всі розглянуті на цей момент елементи початкового списку u. Отже, коли u є nill, то v вже накопичив весь результат в цілому, а якщо u=/=nill, то можна накопичити в v car від u і викликати рекурсивно функцію rev для обробки cdr від u.

Нижче наведені значення u і v, які будуть отримані при послідовних викликах функції rev, якщо спочатку функція reverse застосовується до списку (a b c d):

U	V
(a b c d)	nill
(b c d)	(a)
(c d)	(b a)
(d)	(c b a)
nil	(d c b a)

У функції (rev u v) другий параметр є накопичувальним. Проблема побудови таких функцій полягає у підборі відповідної функції з накопичивальним параметром, який обчислює необхідний результат. Отримана функція є значно економнішою за попереднє визначення, оскільки викликає функцію cons лише n разів, тоді як попереднє визначення функції reverse робить n(n+1)/2 викликів функції cons.

Розглянемо застосування методу накопичення параметрів в більш типовому випадку, коли треба визначити функцію, яка накопичує більш ніж один результат. Розглянемо функцію сума добутків (summult u), яка

видає в якості результату двочлен, перший елемент якого є сумою, а другий – добутком елементів. Тобто визначимо функцію так:

_(defun summult (u) (double (sum u) (mult u))).

Введемо допоміжну функцію (cn u s p), з двома додатковими параметрами, які будуть накопичувати суму і добуток відповідно. Тобто спробуємо визначити функцію cn у такому вигляді:

_(defun (cn u s p) (double (+ s (sum u)) (* p (mult u))))

і тоді зможемо визначити

_(defun summult (u) (cn u 0 1)).

Функцію cn(u,s,p) побудуємо шляхом аналізу випадків по u:

Випадок 1. Якщо u=nill то (cn u s p)=(double s p)
Випадок 2. Якщо u=/=nill то,

нехай

(cn (cdr u) s` p`)=(double(+ s` (sum (cdr x))) (* p` (mult (cdr u)))),
тоді
(cn u s p)=(double (+ s (sum u)) (* p (mult u))) =
= (double (+ s (car u) (sum (cdr u))) (* p (car u) (mult (cdr u))))=
= (cn (cdr u) (+ s (car u)) (* p (car u))).

Знов прийшлося робити досить довге виведення. При цьому знадобилося не тільки обрати підфункції, але і використати прості рівності при u=/=nill:

(sum u)=(+ (car u) (sum (cdr u)))
(mult u)=(* (car u) (mult (cdr u)))

Кінцевим результатом стало визначення:

```
_(defun cn(u s p) (cond (null u) (double s p))
                       (t (cn (cdr u) (+ s (car u)) (* p (car u))))))).
_(defun double(x y) (cons x (cons y nill))).
```

Ми отримали дуже цікаву і просту функцію. При цьому запобігається подвійний перебір списку u, а також не використовується маса зайвих розкладань і об'єднань двочленів, які були б необхідні, якщо б спробувати запрограмувати цю функцію без накопичування параметрів. В цьому випадку визначення могло б мати, наприклад, такий вигляд:

```
_(defun summu(u) (cond ((null u) (double 0 1))
                       (t  (reach (car u) (summa (cdr u))))))).
_(defun reach (n z) (double (+ n (car z)) (* n car (cdr z)))).
```

В той час як версія із накопичуванням параметрів будує лише один двочлен, остання версія будує n+1 двочленів, де n –довжина початкового списку u.

5.7 Приклад реалізації алгоритму пошуку на ЛІСПі

Розглянемо реалізацію на основі функціонального підходу відомої задачі про фермера, вовка, козу і капусту.

Фермер (Farmer), вовк (Wolf), козел (Goat) і капуста (Cabbidge) знаходяться на одному березі річки. Треба перебратися човном на інший берег. При цьому повинні виконуватися такі умови:

- човен може водночас перевозити не більше двох пасажирів;
- неможна залишати на одному березі сам на сам козу і капусту та козу і вовка.

Головна проблема в формуванні алгоритму полягає в тому, як знайти найбільш ефективне подання інформації про задачу структурою даних ЛІСПу [2].

Процес перевезення може бути поданий послідовністю станів. Стан подається списком з чотирьох елементів, кожний з яких відображає розміщення об'єктів **F**, **W**, **G**, **C**. Наприклад, стан:

(e w e w)

означає, що – **F** і **G** знаходяться на східному березі (e - east);
W і **C** знаходяться на західному березі (w - west).

Визначимо дві функції:

конструктор - make-state,

який бере за аргументи розміщення **F**, **W**, **G**, **C** і повертає стан:

(defun make-state (f w g c) (list f w g c)),

і чотири функції доступу, кожна з яких отримує стан і повертає розміщення:

- (defun farmer-side (state) (nth 0 state))
- (defun wolf-side (state) (nth 1 state))
- (defun goat-side (state) (nth 2 state))
- (defun cabbage-side (state) (nth 3 state))

Вся подальша програма базується на цих функціях доступу і конструкторах. Зокрема, вони використовуються для реалізації чотирьох можливих дій фермера: перевезення через ріку самого себе або **W**, **G**, **C**.

Ці функції використовують чотири функції доступу для розбиття стану на його компоненти.

Функція opposite (її буде визначено пізніше) визначає нове розміщення об'єктів, що переїхали ріку, а make-state збирає їх у новий стан.

Наприклад, функція farmer-takes-self може бути визначена таким чином:

```
defun farmer-take-self (state)
(safe (make-state (opposite (farmer-side state))
(wolf-side state)
(goat-side state)
(cabbage-side state))))
```

Відмітимо, що ця функція повертає новий стан незалежно від того, чи є він небезпечним чи ні. Але стани можуть бути і небезпечними, наприклад коли **W** і **G** або **G** і **C** знаходяться на одному березі.

Програма ж повинна використовувати як рішення тільки безпечні стани. Перевірка на небезпечність станів має проводитися у різних місцях програми. В нашому випадку це можна зробити у функціях руху:

Реалізуємо таку перевірку з використанням функції safe, яка має таку поведінку, в разі якщо стан є безпечним, (safe '(w w w w)) повертає його без змін (w w w w).

Safe використовується в кожній функції переміщення з метою фільтрування небезпечних станів. Отже, будь-яке переміщення, що веде до небезпечного стану, буде повертати nil замість стану. Алгоритм формуванн шляху в просторі станів може перевіряти цей nil і використовувати його для заборони даного стану.

З використанням safe, можемо записати нове визначення функції:

```
(defun farmer-take-wolf (state)
(cond ((equal (farmer-side state) (wolf-side state))
(safe (make-state (opposite (farmer-side state))
(opposite(wolf-side state))
(goat-side state)
(cabbage-side state))))
(t nil)))
```

Інші функції руху визначаються аналогічно, але містять в собі умовний тест для визначення, чи знаходяться фермер і можливий пасажир на тому ж самому боці ріки. Якщо ні, то переміщення неможливе, тобто якщо фермер і пасажир не знаходяться на одному боці ріки, ця функція буде повертати nil:

```
(defun farmer-take-goat (state)
(cond ((equal (farmer-side state) (goat-side state))
(safe (make-state (opposite (farmer-side state))
(wolf-side state)
(opposite(goat-side state))
(cabbage-side state))))
(t nil)))
```

```
(defun farmer-take-cabbage (state)
(cond ((equal (farmer-side state) (cabbage-side state))
(safe (make-state (opposite (farmer-side state))
(wolf-side state)
(goat-side state)
(opposite(cabbage-side state)))))
(t nil)))
```

Тепер можна визначити функцію opposite, яка відповідає за повернення на іншу сторону:

```
(defun opposite (side)
(cond ((equal side 'e) 'w)
((equal side 'w) 'e)))
```

Функція safe визначена з використанням cond для перевірки двох небезпечних станів:

F знаходиться на протилежному березі від **W** і **G** та від **G** і **C**.

Якщо стан є безпечним, то він буде повертатися без змін:

```
(defun safe (state)
(cond ((and (equal (goat-side state) (wolf-side state))
(not (equal (farmer-side state) (wolf-side state)))) nil)
((and (equal (goat-side state) (cabbage-side state))
(not (equal (farmer-side state) (goat-side state)))) nil)
(t state)))

(defun path (state goal)
(cond ((equal state goal))
(t (or (path (farmer-takes-self state) goal)))
(path (farmer-take-wolf state) goal)
(path (farmer-take-goat state) goal)
(path (farmer-take-cabbage state) goal))
```

Ця версія функції path є простим перекладом і містить кілька помилок, які треба виправити. Зокрема, відмітимо використання форми OR для управління виконанням її аргументів.

Повторимо, що OR виконує свої аргументи доти, поки один з них не поверне не-nil величину. Коли це відбудеться, OR закінчує роботу без виконання інших аргументів і повертає це не-nil значення як результат.

Таким чином, OR використовується не тільки як логічний оператор, але також забезпечує спосіб управління пошуком шляху. Логічна функція OR використовується тут замість cond, оскільки величина що тестується і величина, що повертається в разі не-nil випадку, є однією і тією ж.

Одна проблема з цим визначенням полягає в тому, що функція переміщення може повернути значення nil, якщо переміщення не може бути здійснене, коли воно веде не до небезпечного стану, щоб запобігти функцію path від спроби генерувати дочірні стани від стану nil. Функція path має спочатку перевіряти, чи дорівнює поточний стан nil, і якщо це так, то path має повертати nil.

Інша проблема, яка виникає у реалізації функції path, полягає в можливості виникнення петель у просторі станів. Якщо дану реалізацію path запустити на виконання, фермер буде їздити взад-вперед між двома берегами нескінченно, тобто алгоритм призведе до нескінчених переходів між двома однаковими станами.

Для запобігання цього до path треба ввести третій параметр, been-list - список всіх станів, які вже були досягненні. Кожного разу, коли path викликається рекурсивно з новим дочірнім станом, батьківський стан має бути доданим до been-list. Додавши до path предикат member можна перевіряти, чи поточний стан не є елементом been-list, тобто, чи пошук ще не використовував цей стан. Це виконується шляхом перевірки поточного стану на його наявність у been-list перед генерацією його нащадків.

Тепер path можна визначити таким чином:

```
(defun path (state goal been-list)
(cond ((null state) nil)
((equal state goal) (reverse (cons state been-list)))
((not (member-lis state been-list ))
(or (path (farmer-take-self state) goal (cons state been-list))
(path (farmer-take-wolf state) goal (cons state been-list))
(path (farmer-take-goat state) goal (cons state been-list))
(path (farmer-take-cabbage state) goal (cons state been-list))))))
```

Функція member використовується трохи інакше за звичайну, оскільки здійснюється перевірка належності елемента списку списків member-equal:

```
(defun member-lis (x lis)
(cond ((null lis) nil)
((equal x (car lis)) t)
(t (member-lis x (cdr lis)))))
```

Замість того, щоб повертати t, можна повертати список станів, які вже були пройдені до досягнення мети. Оскільки мета не міститься у списку, вона може бути вставлена як останній елемент.

Перед тим, як повернути список, його треба перевернути з використанням функції reverse. Остаточно, для приховання параметру been-list від користувача, можна написати додаткову функцію виклику, яка має два аргументи – початковий і кінцевий стани і викликає функцію path з пустим списком been-list = nil.

```
(defun solve-fwgc (state goal) (path state goal nil))
```

Для розв'язання задачі можна використати такий виклик:

```
(solve-fwgc '(w w w w) '(e e e e)).
```

Відповіддю для нашого прикладу буде така послідовність станів:

1) w w w w	5) w e w w
2) e w e w	6) e e w e
3) w w e w	7) w e w e
4) e e e w	8) e e e e

5.8 Контрольні питання

1. Напишіть рекурсивну функцію, яка визначає скільки разів функція FIB викликає сама себе. Зрозуміло, що FIB(1) і FIB(2) не викликають функцію FIB.
2. Напишіть функцію для обчислення поліномів Лежандра (P0(x)=1, P1(x)=x, Pn+1(x)= ((2*n+1)*x*Pn(x)-n*Pn-1(x))/(n+1) при n>1).
3. Напишіть функцію, яка:
   a) обчислює кількість атомів на верхньому рівні списку (для списку (a в ((a) c) e) вона дорівнює трьом.);
   b) визначає кількість підсписків на верхньому рівні списку;
   c) обчислює повну кількість підсписків, що входять до даного списку на будь-якому рівні.
4. Напишіть функцію, яка:
   a) має два аргументи X і N, і створює список з N раз повторених елементів X;
   b) вилучає повторне входження елементів до списку;
   c) з даного списку створює список його елементів, наприклад, (a b) ⇨ ((a) (b));
   d) обчислює максимальний рівень вкладення підсписків у списку;
   e) має один аргумент, що є списком списків, і об'єднує всі ці списки в один;
   f) залежить від трьох аргументів X, N і V, і додає X на N-е місце в список V.
5. Напишіть функцію:
   a) аналогічну функції SUBST, але в якій третій аргумент W обов'язково має бути списком;
   b) яка має здійснювати заміни X на Y тільки на верхньому рівні W;
   c) яка замінює Y на число рівне глибині вкладення Y в W, наприкладY=A, W=((A B) A (C (A (A D)))) ⇨ ((2 B) 1 (C (3 (4 D))));
   d) аналогічну функції SUBST, але яка здійснює взаємну заміну X на Y, тобто X ⇨ Y, Y ⇨ X.

6 ФУНКЦІОНАЛИ ТА ВЛАСТИВОСТІ СИМВОЛІВ

6.1 Функціонали відображення

Всі функції, що були розглянуті раніше, мали в якості аргументів дані. Але як аргумент функції можна вказувати і функцію.

Аргумент, значенням якого є функція, називається функціональним аргументом, а функція, що має функціональний аргумент – називається функціоналом [8].

Розбіжність між поняттями "дані" і "функція" визначаються не на основі їх структури, а в залежності від використання. Якщо аргумент використовується в функції як об'єкт, що бере участь в обчисленнях, то це дані. Якщо аргумент використовується як засіб, що визначає обчислення, то це функція.

Важливим класом функціоналів у ЛІСПі є функціонали відображення (MAP-функціонали). MAP-функціонали є функціями, які певним чином відображають (map) вихідний список у новий список.

Одним з основних функціоналів відображення є MAPCAR-функціонал. Він має два аргументи, перший з яких є функцією, а інший – списком:

(MAPCAR f '(x1 x2 x3 ... xN))

Під час виконання MAPCAR функція, що визначена першим аргументом, застосовується до кожного елемента списку, що визначається другим аргументом, і результат відображає в новий список.

Наприклад, виклик функції:

_(MAPCAR 'add1 '(1 2 3)),

де add1 є функцією яка збільшує значення аргументу на 1, та поверне в якості результату список (2 3 4).

Взагалі визначення (MAPCAR f '(x1 x2 x3 ... xN)), еквівалентне визначенню (list (f 'x1) (f 'x2) ... (f 'xN)).

MAPCAR можна використовувати в функціях:

```
(defun list-add1 (lis) (mapcar 'add1 lis))
_(list-add1 '(1 2 3))
(2 3 4)
```

В якості аргументу для MAPCAR можна використовувати і значення символів:

```
_(setq x '(a b (d)))
_(setq y 'atom)
_ (mapcar y x)
(t t nil)
```

Якщо у функціональному аргументі MAPCAR присутні кілька аргу-

ментів, він може обробляти кілька списків.

Наприклад:

```
_(defun addlist (l1 l2) (mapcar '+ l1 l2))
_(addlist '( 1 2 3) '(2 3 4))
(3 5 7).
```

Тобто, дія еквівалентна дії функції є такою:

(list (+ 1 2) (+ 2 3) (+ 3 4))

Якщо списки мають різну довжину, то довжина результату буде дорівнювати довжині найменшого з списків.

Замість визначення спеціальних функцій, які потрібні тільки для визначення MAPCAR, доцільно використовувати лямбда-функції.

Наприклад, нехай потрібно визначити функцію над елементами списку, яка буде замінювати елемент x на x^2:

```
(defun f1 (x) (+ 1 (* x x)))
_(mapcar 'f1 '(1 2 3))
```

Більш ефективним у цьому випадку буде використання лямбда-виразу:

(mapcar '(lambda (x) (+ 1 (* x x))) '(1 2 3))

6.2 Властивості символів

ЛІСП дозволяє зв'язувати з символом не тільки значення але і додаткову інформацію, яка називається списком властивостей (property list).

Наприклад, розглянемо інформацію про Mary:

Властивість	Значення
aqe	28
occupation	lawyer
salary	90
children	Bill Alice Susan

Список властивостей в даному випадку має такий вигляд:

(aqe 28 occupation lawyer salary 90 children (Bill Alice Susan))

Для зчитування властивостей атома служить функція:

(GET <символ> <властивість>),

яка повертає значення конкретної властивості:

```
_( get 'Mary 'age)
28
_( get 'Mary 'children)
```

```
( Bill Alice Susan))
_( get 'Mary 'hobby)
nil
```

Задати властивості можна за допомогою узагальненої функції присвоєння setf :

```
( setf ( get <символ> <властивість>) <значення>)
        _( setf ( get 'Mary 'salary) 90)
        90
```

Ясна річ, що спочатку властивість має бути задана, а потім вже зчитана.

Щоб задати властивість, можна скористатися і спеціальною функцією putprop:

```
( putprop <символ> <значення> <властивість>)
```

<властивість> - нечисловий атом;
<значення> - будь-який вираз.

Функцію putprop можна визначити таким чином:

```
(defun putprop ( atom value property)
(setf (get atom property) value))
```

Атом може мати багато властивостей, але кожна властивість може мати лише одне значення. При внесенні нової властивості, вона додається на початок списку властивостей:

```
_(putprop 'Mary 'cinema 'hobby)
( hobby cinema .....)
```

Заміна значення властивості здійснюється повторним присвоюванням. Наприклад,

```
_(putprop 'mary 29 'age)
_(get 'mary 'age)
29
```

В разі заміни поточного значення на нове, яке враховує поточне, можна зробити це, наприклад, таким чином:

```
(putprop 'mary (+ 1 (get 'mary 'age)) 'age)
```

Вилучення властивості і її значень здійснюється за допомогою функції remprop:

```
(remprop <символ> <властивість>).
_(remprop 'Mary 'age)
T
```

Інформацію про список властивостей можна отримати за допомогою функції SYMBOL-PLIST:

_(SYMBOL-PLIST 'Mary)
(aqe 28 occupation lawyer salary 90 children (Bill Alice Susan))

6.3 Функції для обробки списків

Звичайні функції виконують операції над списками без внесення змін у покажчики спискових комірок. У тих випадках, коли виникає така необхідність, створюється нова копія спискової комірки з новим вмістом полів. Наприклад, розглянемо послідовність дій для функції APPEND:

```
_ (setq first '(a b))
(a b)
_ (setq second '(c d))
(c d)
_ (setq both (append first second))
(a b c d)
```

Проілюструємо ці дії діаграмою, що показана на рисунку 6.1:

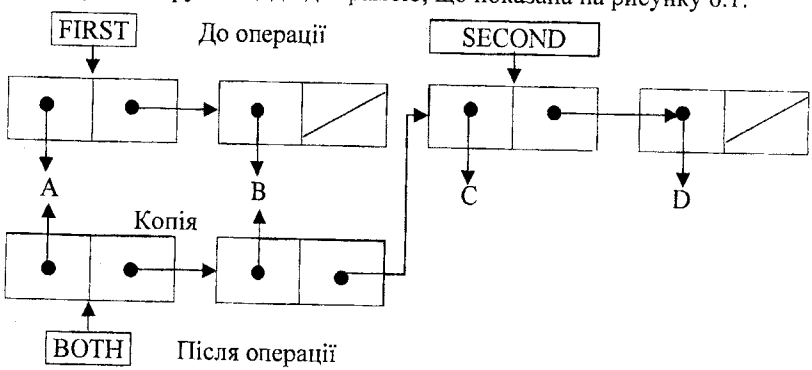


Рисунок 6.1 - Створення нових копій спискової комірки з використанням функції APPEND

APPEND створює копії всіх спискових комірок для кожного елемента кожного з аргументів, крім останнього аргументу. В той же час, наприклад, CONS створює тільки одну спискову комірку. Якщо об'єднуються два списки в 1000 і 1 елемент, то буде створено 1000 копій спискових комірок, замість того щоб виправити один покажчик.

Замість того, щоб створювати нові спискові комірки ЛІСП дозволяє змінювати вміст покажчиків. Це забезпечується використанням руйнівних функцій, які отримали таку назву тому, що замінюють покажчик, внаслідок чого початкова структура руйнується.

Поєднати два списки шляхом руйнування покажчика дозволяє функція NCONC:

(setq new (nconc first second))

Результат застосування функції NCONC показаний на рисунку 6.2.

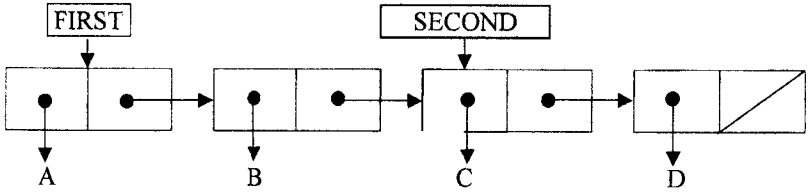


Рисунок 6.2 - Результат застосування функції NCONC до спискових комірок

В даному випадку список не копіюється. Замість цього nil в останній списковій комірці замінюється покажчиком на першу спискову комірку другого списку. При цьому побічним ефектом буде руйнування списку first:

```
_both
(a b c d)
_first
(a b c d)
```

Дві інші функції змінюють структуру своїх аргументів:

- RPLACA - "replace the car" має два аргументи , причому перший повинен бути списком. Функція замінює голову першого аргументу на другий аргумент. Тобто покажчик CAR першої спискової комірки, що вказував на голову списку, замінюється на покажчик до другого аргументу.
- Аналогічно, функція REPLACD - "replace the cdr", замінює покажчик CDR.

Розглянемо таку послідовність дій (рисунок 6.3):

```
_ (setq lis1 '(a b c))
_ (setq lis2 '(a b c))
```

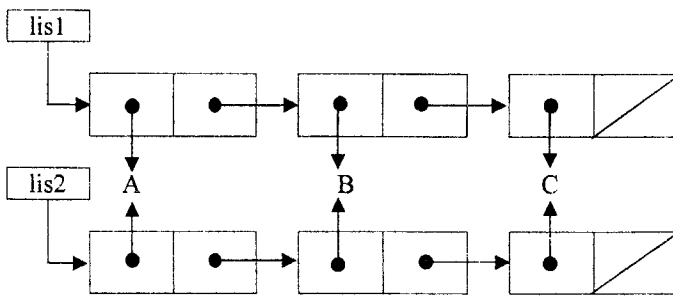


Рисунок 6.3 - Подання списків lis1 і lis2 списковими комірками

Тепер виконаємо такі дії (рисунки 6.4, 6.5):

```
_ (rplaca lis1 'd)
(d b c )
```

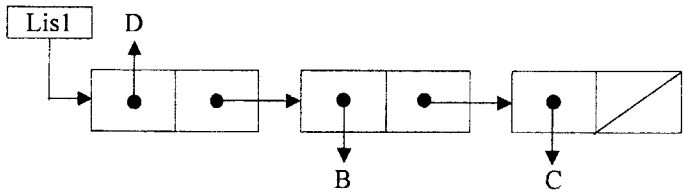


Рисунок 6.4 - Зміна структури списку з використанням функції rplaca

```
_ (rplacd lis2 '(e f))
(a e f)
```

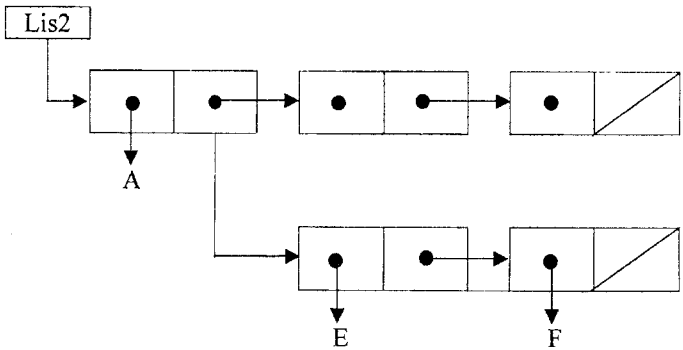


Рисунок 6.5 - Зміна структури списку з використанням функції rplacd

Функції rplaca і rplacd можна подати через setf:

(rplaca x y) <=> (setf (car x) y)
(rplacd x y) <=> (setf (cdr x) y)

Їх можна також використовувати для заміни елементів. Наприклад, розглянемо функцію replace-item, яка має три елементи. При цьому перший елемент повинен бути списком. Функція заміщує з руйнуванням перший елемент списку, що відповідає другому аргументу, на третій аргумент:

(defun replace-item (lis old new)
(rplaca (member old lis) new)).

Функції з руйнуванням необхідно використовувати при роботі з великими списками, аби не збільшувати витрати пам'яті. Наприклад, використовувати NCONC замість APPEND. Але використовувати руйнівні функції треба дуже обережно, оскільки їх використання викликає побічні

ефекти. Наприклад, можна несподівано отримати нескінчені цикли (рисунок 6.6.)

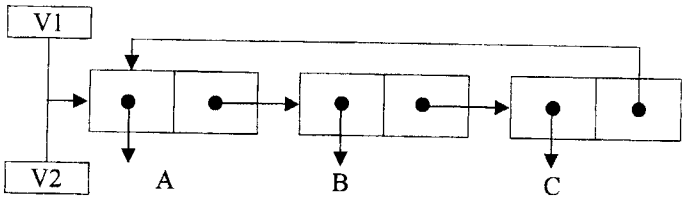


Рисунок 6.6 - Приклад виникнення нескінченного циклу

```
_ (setq v1 '(a b c))
(a b c)
_ (setq v2 v1)
(a b c)
_ (setq v2 (nconc v1 v2))
(a b c a b c....)
```

6.4 Функціонали застосування

Група функціоналів, які застосовують функціональний аргумент до його параметрів, називається у ЛІСПі функціоналами застосування. Оскільки функціонали застосування обчислюють значення функції, вони певною мірою подібні до функції EVAL, яка обчислює значення виразу. Розглянемо функцію APPLY. Вона має два аргументи: ім'я функції і список, та застосовує названу функцію до елементів списку, як до аргументів функції:

(APPLY f '(x1 x2 x3 ... xn)) = (f 'x1 'x2 'x3 ... 'xn)

Наприклад, ми можемо об'єднати до одного списку кілька вкладених списків, тобто з ((a b c) (d e f) (k l)) отримати (a b c d e f k l):

(APPLY 'append '((A)(B)(C)))

Визначимо функцію, яка обчислює середнє значення чисел списку:

```
(defun list-mean (lis)
(/ (apply '+ '(lis))(length lis)))
_ (list-mean '(1 2 3 4))
2.5
```

Часто APPLY використовують разом із MAPCAR. Наприклад, за допомогою цього прийому можна знайти загальне число елементів списку:

```
_ (countall '((a b c) (d e f) (k l)))
8
(defun countall (lis)
```

```
(apply '+ (mapcar 'length lis)))
```

Можна визначити і більш складну функцію countatom, яка обчислить загальну кількість елементів у будь-якому списку:

```
_ (countatom '(a (a (b) c) (d) e (f g)))
8
(defun countatom (lis)
(cond ((null lis) 0)
((atom lis) 1)
(t (apply '+ (mapcar 'countatom lis)))))
```

LISP надає багато різноманітних можливостей щодо реалізації певних функцій, кожна з яких має свої особливості. Наприклад, нехай необхідно побудувати функцію list-last, яка створює новий список з хвостів вихідних списків.

```
_ (list-last '((a b) (b c) (c d)))
(b c d)
```

Ми можемо визначити таку функцію за допомогою APPEND

```
(defun list-last (lis)
(apply 'append (mapcar 'last lis)))
```

Але APPEND працює повільно і залишає багато сміття. Можна визначити потрібну функцію з використанням NCONC:

```
(defun list-last (lis)
(apply 'nconc (mapcar 'last lis)))
```

А можна використати і функціонал відображення MAPCAN, який існує в ЛІСПі:

(mapcan fn x1 x2 ... xN) <=> (apply 'nconc (mapcar fn x1 x2 ... xN))

Тобто, MAPCAN об'єднує результати до одного списку з використанням функції NCONC:

```
(defun list-last (lis)
(mapcan 'last lis))
```

В ЛІСПі існує також подібний до APPLY функціонал застосування FUNCALL, який відрізняється лише тим, що приймає окремі аргументи, а не список, як APPLY:

(funcall fn x1 x2 ... xN) <=> (fn x1 x2 ... xN),

де fn - функція з n аргументами.

Розглянемо приклади використання FUNCALL:

```
_ (funcall '+ 1 2) <=> _ (+ 1 2)
3
```

```
_ (funcall (car '(+ - / *)) 1 2)
3
```

Побудуємо з використанням FUNCALL функцію MAP2, яка діє подібно до MAPCAR, але бере в якості аргументів два елементи зі списку, а не один:

```
(defun map2 (f2 lst)
(if (null lst)
nil
(cons (funcall f2 (car lst) (cadr lst))
(map2 f2 (cddr lst)))))
_ (map2 'list '(A Christie V Nabokov K Vonnegut))
((A Christie) (V Nabokov) (K Vonnegut))
```

6.5 Контрольні питання

1. Визначте поняття властивості символу.
2. Які переваги надає використання властивостей символів?
3. Що таке функціонал?
4. Надайте стислу характеристику функціоналів відображення.
5. Які функціонали застосування Вам відомі?
6. Призначення, переваги та недоліки руйнівних функцій для роботи зі списками.
7. Для кожного студента зі списку, що був складений при виконанні вправ 6 –6.2 до розділу 2 (стор.34), складіть такі списки властивостей:
   а) оцінки за лекції;
   б) оцінки з практичних занять;
   в) оцінки з лабораторних робіт.
8. Для довільно вибраних студентів порівняйте їх властивості.
9. Напишіть функцію, яка запитує у користувача ПІБ студента з групи (список групи складений раніше) і видає список його властивостей.
10. Напишіть функцію від одного аргументу (ПІБ будь-якого студента), що заміщує в списку з даними про нього (написаному раніше) підсписки з середніми балами на списки властивостей.
11. Припустимо, що ім'я міста має властивості *x* і *y*, які містять координати місця розташування міста відносно початку координат. Напишіть функцію (ВІДСТАНЬ a b), яка обчислює відстань між містами *a* і *b*, якщо значенням функції (SQRT x) є квадратний корінь числа *x*.
12. Обчисліть значення таких викликів:
    a) (APPLY 'LIST '(a b));
    b) (FUNCALL 'LIST '(a b));
    c) (FUNCALL 'APPLY 'LIST '(a b));
    d) (FUNCALL 'LIST 'APPLY '(a b);

13. Визначте функціонал (APPLY f x), який застосовує кожну функцію f_i списку f = (f_1 f_2 ... f_n) до відповідного елементу x_i списку x = (x_1 x_2 ... x_n) і повертає список, сформований з результатів.
14. Визначте функціональний предикат (КОЖНИЙ *пред список*), який є істинним тільки в тому випадку, коли істинним для всіх елементів списку *список* є предикат *пред*, що є функціональним аргументом.
15. Визначте функціональний предикат (ДЕЯКИЙ *пред список*), який є істинним, коли предикат *пред* істинний хоча б для одного елемента списку *список*.
16. Визначте FUNCALL з використанням функціоналу APPLY.
17. Визначте фільтри: (ВИЛУЧИТИ-ЯКЩО *пред список*), (ВИЛУЧИТИ-ЯКЩО-НЕ *пред список*) що вилучають зі списку *список* всі елементи які, відповідно, мають або не мають властивості, наявність яких перевіряє предикат *пред*.
18. Визначте функціонал (MAPLIST fn *список*) для одного спискового аргументу.

7 МАСИВИ І МАКРОСИ

7.1 Робота з масивами

В багатьох застосуваннях масиви є найбільш зручними структурами даних. За допомогою масивів можна зібрати і впорядкувати дані в цілому, причому до окремих елементів даних можна звертатися за допомогою механізму індексування і змінювати їх незалежно від вмісту даних.

Для визначення масиву заданої розмірності використується функція MAKE-ARRAY (make-array <розмірність>), яка підтримує тільки одномірні масиви-вектори.

Наприклад, визначимо масив з десяти елементів:

```
(setq data (make-array 10))
_(0 0 0 0 0 0 0 0 0 0)
```

де,

data – ім'я масиву, 0 - початкове наповнення

Доступ до комірок масиву здійснюється за допомогою функції AREF, яка має два аргументи – ім'я масиву і індекс, і повертає значення комірки:

```
(aref <ім'я> <індекс>).
_ (aref data 8)
0,
```

(оскільки там записаний 0).

При роботі з функцією слід пам'ятати, що перший аргумент функції не блокується, а перша комірка має номер 0.

Помістити дані до масиву можна за допомогою функції SETF:

```
_ (setf (aref data 2) 'dog),
```

де aref - викликає значення комірки, а функція setf – розміщує значення.

Розглянемо масив testdata:

```
_(setq testdata (make-array 4))
_(setf (aref testdata 1) 'dog)
_(setf (aref testdata 0) 18 )
_(setf (aref testdata 2) '(a b) )
_(setf (aref testdata 3) 4 )
```

Можна використати і такий виклик функції:

```
(setq testdata ( vector 18 'dog '(a b) 0)) (aref d 1)
```

В результаті отримаємо:

```
testdata 0 1 2 3
18 dog (a b) 0
```

Тепер можна працювати з даними, що занесені до масиву:

```
(cons (aref testdata 1) (list (aref testdata 3) (aref testdata 2)))
(dog 0 ( a b))
(aref testdata (aref testdata 3))
18
```

Оскільки доступ до елементів масиву здійснюється за номерами, то зручно використовувати числові ітерації та рекурсію. Розглянемо функцію, яка бере два аргументи: ім'я масиву та його довжину і повертає всі значення, які розміщені у списку:

```
(defun array-list (arnam len)
(do (( i 0 ( + 1 i))
( result nil (append result (list (aref arnam i)))))
(( equal i len) result)))
(array-list testdata 4)
( 18 dog (a b) 0)
```

Довжину масиву повертає функція (ARRAY-LENGTH <ім'я>). Наприклад, виклик (array-length testdata) поверне в якості довжини масиву – 4.

7.2 Зворотне блокування

Звичайне блокування забороняє обчислення виразів:

```
_ '(a b c)
(a b c)
```

Але іноді буває необхідно обчислити частину виразу. Наприклад, якщо маємо:

```
(setq b '(x y z))
```

і запишемо:

```
_ `( a ,b c),
```

то b - буде обчислено і функція поверне такий результат:

```
(a (x y z) c).
```

Зворотна верхня кома ` визначає блокування, яке може частково зніматися комою.

Зворотне блокування може використовуватися і при друкуванні:

```
(setq n 'john)
(setq m 'Robert)
(print `(boys ,n and ,m))
(boys john and roberts)
```

7.3 Макроси

Часто буває корисним не записувати вираз для обчислення вручну, а сформувати його автоматично, за допомогою програми. Така ідея динамічного програмування особливо зручно реалізується в ЛІСПі, оскільки програма і дані в цій мові подаються за допомогою однакових списків. При цьому обчислення отриманого виразу або його частини завжди можна при необхідності заблокувати за допомогою QUOTE, наприклад, з метою перетворення виразу. В той же час, для обчислення сформованого виразу завжди можна викликати інтерпретатор EVAL.

Найбільш зручним засобом в ЛІСПі для програмного формування виразів є спеціальні макроси. Макрос - це спеціальний засіб, що дозволяє формувати вираз, що обчислюється безпосередньо в процесі виконання програми.

Розглянемо, наприклад, функцію BLANCS, яка виконує n переводів каретки (пропускає n рядків):

```
(defun blancs (n)
(do ((count n ( - count 1)))
((zerop count) nil)
(terpri)))
```

(blancs 4) - пропустить чотири рядки.

Напишемо програму, яка дозволить виконати певну дію n разів, наприклад, пропустити чотири рядки:

```
(do-times '(terpri) 4),
```

або тричі надрукувати рядок hello:

```
(do-times '(print 'hello) 3).
```

Таку функцію можна визначити з використанням EVAL:

```
(defun do-times (operation n)
(do ((count n ( - count 1)))
(( zerop count) nil)
(eval operation)))
```

Але це можна зробити також за допомогою спеціальної форми DEFMACRO

```
(defmacro do-times (operation n)
` (do ((count ,n ( - count 1)))
(( zerop count) nil)
,operation))
```

Як можна побачити з наведеного прикладу, форма макросу схожа на визначення функції, але є й певні розбіжності:

1. Аргументи макросу не обчислюються перед їх використанням. При цьому звернення записується як:

(do-times (print 'hello) 3)

2. При виклику макросу ЛІСП спочатку обчислює його макро, а потім виконує вираз, що утворився. Наприклад, після звернення:

(do-times (print 'hello) 3)

отримаємо

```
(do ((count 3 ( - count 1)))
(( zerop count) nil)
(print 'hello))
```

Після цього даний список буде обчислено.

Отже, при виклику макросу спочатку обчислюється тіло (цей етап називається розширенням) і формується вираз. На другому етапі обчислюється отриманий вираз і отримане значення повертається як результат.

Виклик macro з різними аргументами повертає різні результати. При виклику:

(do-times (print count) 10)

після обчислення тіла отримаємо:

```
(do ((count 10 ( - count 1)))
(( zerop count) nil)
(print count))
```

При цьому будуть друкуватися числа від 10 до 1.

Можна визначити і функцію зворотного друку чисел:

```
(defun print-number (n)
(do-times (print count) n))
( print-number 5)
```

При розробці макросів необхідно виконати три кроки:

1) написати приклад функції, яку повинен формувати макрос;
2) виділити постійні і змінні частини для кількох функцій. Змінні частини позначити змінними, виділити комами і винести до аргументів. Постійні частини записати напряму;
3) визначити макрос, який реалізує потрібний виклик.

В якості прикладу визначимо макрос TERM-SEARCH, який буде проглядати список і виділяти перший елемент, що задовольняє задану умову.

Крок1. Сформулюємо приклад. Запишемо тіло для пошуку чисел:

(setq l '(s d 3 d)) (setq item 5)

```
(do (( tail  l (cdr tail)))
((null tail) nil)
( cond ((numberp (car tail)) (return (car tail)))))
```

Крок2. Виділимо загальні частини: список lis - l і предикат predicate - number.

Крок3.Формирулюємо макрос:

```
(defmacro term-search ( predicate lis)
` (do (( tail , lis (cdr tail)))
((null tail) nil)
(cond ((,predicate (car tail)) (return (car tail))))))
```

7.4 Приклад розробки програми диференціювання алгебраїчних виразів

Напишемо програму диференціювання алгебраїчних виразів на ЛІСПі. Для наочності обмежимося алгебраїчними виразами в такій формі:

(+ x y) (* x y)

Додавання і множення можуть вільно комбінуватися. Наприклад, (*(+ a (* a b)) c).

Безпосередньо програмуючи отримаємо:

```
(defun diff0 ( l x)
(cond (( atom l)
       (if (eq l x) 1                ;l=1
           0))                       ;l=константа
   (( eq (first l) '+)
     (list '+
          (diff0  (second l) x)
          (diff0  (third  l) x)))
   (( eq (first l) '*)
     (list '+
       (list '*
          (diff0  (second l) x)
          (third  l))
       (list '*
          (diff0  (third l) x)
          (second  l))))
   (t l)))
```

Використаємо отриману програму:

```
(diff0 '(+ x (* 3 x)) 'x)
```

```
( + 1 (+ (* 0 x) (* 1 3))) = 4
(diff0 '(- x (* 3 x)) 'x) return
  (- x (* 3 x))  Why?
(diff0 '(* x ( + x 1)) 'x)
  (+ (* 1 ( x 1))(*(1 0) x))
```

В даній програмі вирази отримані шляхом обчислень не спрощуються, але це не важко зробити.

Отримана програма є не дуже зручною, оскільки вона не є модульною. Її важко розширювати, необхідно групувати всі умови в одному COND.

Більш зручне рішення можна отримати якщо для кожної дії визначити свою функцію диференціювання і завдяки властивості DIFF зв'язати цю функцію з символом, який позначає дію.

Спростимо запис самої функції диференціювання:

```
(defun dif1 (l x)
      (cond (( atom l)
            (if (eq l x) 1 0))
            ( t (funcall (get (first  l) 'diff) (cdr l) x))))
```

Функції диференціювання стають значеннями властивості 'DIFF:

```
(setf (get '+ 'diff) 'dif+) (setf (get '* 'diff) 'dif*)
```

Таким чином з даних отримується дія. Самі функції записуються у вигляді:

```
(defun dif* (l x)
      (list '+ (list '* (dif1 (first l) x)
                        (second l))
               (list '* (dif1 (second l) x)
                        (first l))))
(defun dif+ (l x)
      (list '+ (dif1 (first l) x) (dif1 (second l) x)))
```

Завдяки модульності тепер можна легко доповнити програму операцією диференціювання для віднімання:

```
(defun dif- (l x)
      (list '- (dif1 (first l) x) (dif1 second l) x)))
```

Таким чином, початковий варіант управління обчислювальним процесом, зв'язаний із структурою програми, ми перетворили на динамічне управління, основане на даних.

Можна використати в даній програмі і макроси. Визначимо макрос DEFDIF, за допомогою якого визначаються функції диференціювання для нових дій:

```
(defmacro defdif (act args body)
`(setf (get ',act 'diff)
'(lambda,args,body)))
```

Тоді функції диференціювання можна задати таким чином:

```
(defdif + (l x)
      (list '+ (difl (first l) x) (difl (second l) x)))
(defdif * (l x)
      (list '+ (list '* (difl (first l) x)
                        (second l))
               (list '* (difl (second l) x)
                (first l))))
(difl '(+ x x) 'x)
(defdif - (l x)
      (list  '- (difl (first l) x)
               (difl (second l) x)))
 (difl '(+ x (* 3 x)) 'x)
(difl '(- x (* 3 x)) 'x)
(difl '(* x ( - x 1)) 'x)
```

Доповнимо програму кількома функціями для забезпечення введення інформації.

Зчитування списку, який необхідно продиференціювати:

```
(defun read-list () (princ " diff-list ? ") (setq l (read)))
```

За ознаку припинення обчислень оберемо введення символу d:.

```
(defun d ()  (princ "enter command : d -diff;q - quit") (terpri)
   (if (eq  (read) 'd)   (progn (read-list)
                  (print  (difl l 'x ))
                  (terpri) (d))
                  'end))
```

Виклик програми тепер здійснюється у формі: (d).

З використанням функції LOAD можна одразу завантажувати отриману програму і починати її виконання:

(load <файл>)

Запис здійснюється звичайним чином, але для того, щоб одразу розпочалося обчислення слід використовувати подвійні лапки: (load "difl")

7.5 Контрольні питання

1. Назвіть відомі Вам функції для роботи з масивами.
2. Наведіть поняття зворотного блокування.
3. Приклади застосування зворотного блокування.
4. Що таке макрос?
5. Відмінності у визначеннях функції і макросу.
6. Поясніть основні дії, які треба виконати при розробці макросу.
7. Що робить такий макрос:
   (def macro аргумент (форма) (list '(lambda (x) x) (car форма))) ?
8. Визначте макрос, який повертає свій виклик.
9. Визначте ліспівську форму (IF умова p q) у вигляді макросу.
10. Визначте у вигляді макросу форму (REPEAT e UNTIL p) паскалевського типу.

8 ЗЧИТУВАННЯ ТА ЗАПИС ІНФОРМАЦІЇ У ФАЙЛИ

8.1 Параметри визначення функцій

При визначенні функції можна використовувати механізм ключових слів, який дозволяє по різному трактувати аргументи при виклику функцій.

За допомогою ключових слів в лямбда-списку можна виділити:

1) необов'язкові аргументи (optional)
2) параметр, що зв'язується з хвостом списку аргументів змінної довжини (rest)
3) ключові параметри (key)

Ключові слова починаються з символу & і їх записують перед відповідними параметрами в лямбда-списку. Дія ключового слова розповсюджується до наступного ключового слова. Параметри перелічені в лямбда-списку до першого ключового слова є обов'язковими. Необов'язкові аргументи функції визначаються дуже просто. Будь-який аргумент після символу &optional є необов'язковим:

```
_ ( defun bar ( x &optional y) ( if y x 0 ))
bar
_ ( defun baaz ( &optional ( x 3 ) ( z 10 )) ( + x z) )
BAAZ
_ ( bar 5 )
0
_ ( bar 5 t)
5
_ ( Baaz 5 )
15
_ ( Baaz 5 6 )
11
_ (Baaz)
13
```

Можна викликати функцію bar з одним або з двома аргументами. Якщо вона викликана з одним аргументом, x буде зв'язане із значенням цього аргументу і незаданий аргумент y буде зв'язано із nil. Якщо вона викликана з двома аргументами, x і y будуть зв'язані із значеннями першого і другого аргументу, відповідно.

Функція baaz має два необов'язкових аргументи. Крім того вона визначає відсутнє значення для кожного з них. Якщо користувач визначить тільки один аргумент, z буде зв'язане з 10 замість nil, і якщо користувач не визначить ні яких аргументів, x буде зв'язане з 3, а z з 10. Таке визначення значень називається визначенням за замовчуванням.

У ЛІСПі можна задати функцію так, щоб вона приймала будь-яку кількість аргументів. Для цього треба закінчити список аргументів &rest

параметром. ЛІСП буде збирати всі аргументи, що не потрапили в обов'язкові параметри, до списку і зв'язувати &rest параметр із цим списком.

Наприклад:

```
_ ( defun foo ( x &rest y) y)
FOO
_ ( Foo 3 )
NIL
_ ( Foo 4 5 6 )
(5 6)
_ ( defun fn ( x &optional y &rest z))
    (list x y z))
fn
_ (fn 'a)
(A NIL NIL)
_ ( a b (c d))
```

Існує ще один вид необов'язкових аргументів, які називаються аргументами ключового слова. Користувач може задавати надалі ці аргументи в будь-якому порядку, оскільки вони відмічені ключовими словами.

Символи t і nil називаються константами-символами, оскільки вони при виконанні повертають самі себе.

Існує цілий клас таких символів, які називаються ключовими словами. Будь-який символ, чиє ім'я починається з двокрапки є ключовим словом.

Розглянемо кілька прикладів використання ключових слів::

```
_ :this-is-a-keyword
:THIS-IS-A-KEYWORD
_ :so-is-this
:SO-IS-THIS
_ :me-too
:ME-TOO
 ( Defun foo ( &key x y) ( cons x y) )
FOO
_ ( Foo :x 5 :y 3 )
(5 . 3)
_ ( Foo :y 3 :x 5 )
(5 . 3)
_ ( Foo :y 3 )
( NIL. 3 )
_ (Foo)
(NIL)
```

Параметр &key може мати значення за замовчуванням:

```
_ ( Defun foo ( &key ( x 5 )) x )
FOO
_ ( Foo :x 7 )
7
_ (Foo)
5
(defun test ( x &optional (y 3) (z 4) &rest  a)
(cons z ( list x a y)))
(test 1 2 3)
(test 1)
(test 3 4 5)
(test 3 2 1 1 2 3)
```

8.2 Вхідні і вихідні потоки

При введенні і виведенні інформації в ЛІСПі використовується поняття потоків – stream. Для потоку визначені ІМ'Я, операції відкриття OPEN, операції закриття CLOUSE і напрямки OUTPUT і INPUT.

Щоб відкрити файл для запису задається його ім'я, виконується операція open і вказується напрямок output:

```
(setq our-output-stream (open "sesame" :direction :output))
```

Задамо:

```
(setq s 'e)
```

Можна вивести це значення в файл:

```
(princ s our-output-stream) ;
```

Можна занести список:

```
(print '(a b c d) our-output-stream)
```

Щоб правильно закрити потік необхідно в його кінець помістити:

```
(terpri our-output-stream)
```

Потім файл закривається:

```
(close our-output-stream)
```

Можна проглянути інформацію в файлі. Для цього відкриємо файл для читання:

```
(setq our-input-stream (open "sesame" :direction :input))
```

Прочитаємо інформацію:

```
(read our-input-stream)
```

Закриємо файл:

```
(close our-input-stream)
```

8.3 Зчитування символів з файлу.

Припустимо, що в файлі зберігається символьна інформація, призначена для обробки. Причому нас цікавить кожен символ у файлі. Досі ми мали можливість вводити тільки атоми, числа і списки.

Формуємо файл. Нехай

```
(setq s "---+++")
(setq p "+++---")
```

Визначимо потік виведення:

```
(setq our-output-stream (open "picture.spl" :direction :output))
(princ s our-output-stream) ; записуємо перший рядок
(terpri our-output-stream) ; закінчуємо його
(princ p our-output-stream) ; записуємо другий рядок
(terpri our-output-stream) ; закінчуємо файл
```

Тепер файл закривається:

```
(close our-output-stream)
```

Отже, файл містить такі рядки:

```
---+++
+++---
```

Для зчитування символів з файлу будемо використовувати функцію:

```
(READ-CHAR <вхідний потік>)
```

Ця функція дозволяє читати друковані символи (CHAR) з файлу. В якості значення отримується десяткове представлення коду символу. Використаємо цю функцію для посимвольного введення інформації з файлу для її подальшого аналізу:

Визначимо:

```
(setq our-input-stream (open "picture.spl" :direction :input))
```

Для зчитування символу використаємо:

```
(read-char our-input-stream)
```

Будемо отримувати послідовність значень:

```
43
43
43
45
```

45
45
10 і т.ін.

Для відбудови вмісту файлу застосовується перекодування:

```
(setq x (read-char our-input-stream)
```

Отже можна показати вміст x:

```
(cond (( = x 43) (prin1'+))
      (( = x 45) (prin1 '-))
          (( = x 10 ) (terpri)))
```

Можна представити інформацію без перекручувань, якщо використовувати цикл:

```
(loop (progn   (setq x  (read-char our-input-stream) )
  (cond (( = x 43) (prin1'+))
        (( = x 45) (prin1 '-))
        (( = x 10 ) (terpri))))))
```

Після виведення маємо:

```
---+++
+++---
```

Закриємо вхідний потік:

```
(close our-input-stream)
```

Для пошуку кінця файлу можна аналізувати помилку зчитування, але краще знати довжину файлу до початку роботи з ним.

9 ЛАБОРАТОРНИЙ ПРАКТИКУМ

Згідно з часом, який виділено на виконання лабораторних робіт в межах дисципліни «Функціональне і логічне програмування», для вивчення ЛІСПу пропонується виконати чотири лабораторних роботи, тематика та зміст яких наведені нижче.

9.1 Лабораторна робота №1

Тема.

Знайомство з середовищем XLISP. Типи даних і засоби роботи з ними. Базові функції ЛІСПу. Визначення функцій. Функції введення-виведення.

Мета.

Знайомство з середовищем XLISP, базовими функціями ЛІСПу, функціями введення-виведення. Отримання навичок у створенні функцій, що визначаються користувачем.

Зміст роботи:

1. Основні положення програмування на ЛІСПі.
2. Точкова нотація.
3. Структуровані типи даних.
4. Робота з середовищем XLISP.
5. Базові функції ЛІСПу.
6. Функції введення.
7. Функції виведення.
8. Створення функцій, що визначаються користувачем.

9.2 Лабораторна робота №2

Тема.

Організація обчислень у ЛІСПі.

Мета.

Вивчення основних функцій та їх особливостей для організації обчислень у ЛІСПі.

Зміст роботи:

1. Речення LET і LET*.
2. Послідовні обчислення.
3. Ітераційні обчислення.
4. Розгалуження обчислень.
5. Циклічні обчислення.
6. Передача управління.

9.3 Лабораторна робота №3

Тема.

Рекурсія у ЛІСПі. Властивості символів.

Мета.

Вивчення основ програмування з використанням рекурсії.

Зміст роботи:

1. Побудова простих рекурсивних функцій.
2. Трасирування рекурсивних функцій.
3. Рекурсія з кількома рекурсивними гілками.
4. Використання параметрів накопичування.
5. Символи. Властивості символів.

9.4 Лабораторна робота №4

Тема.

Функціонали. Масиви і макроси. Робота з файлами.

Мета.

Вивчення основ роботи з функціоналами, макросами, масивами та файлами.

Зміст роботи:

1. Функціонали відображення.
2. Функціонали застосування.
3. Функції для обробки списків.
4. Використання параметрів накопичування.
5. Робота з масивами.
6. Використання зворотного блокування.
7. Макроси.
8. Параметри визначення функцій.
9. Вхідні і вихідні потоки.

ЛІТЕРАТУРА

1. Лорьєр Ж.-Л. Системы искусственного интеллекта: Пер. с франц. – М.:Мир, 1991. - 568 с.
2. Морозов М.Н. Функциональное программирование. Курс лекций. - http://www.marstu.mari.ru:8101/mmlab/home/lisp/title.htm, 1999.
3. Программирование на языке R-Лисп / А.П.Крюков, А.Я.Родионов, А.Ю.Таранов, Е.М.Шаблыгин. – М.: Радио и связь, 1991. – 192 с.
4. Уинстон П. Искусственный интеллект: Пер. с англ. /Под ред. Ю.В.Матиясевича. –М.:Мир, 1980. – 513 с.
5. А.Филд, П.Харрисон. Функциональное программирование.-М.: Мир, 1993.
6. Хендерсон П. Функциональное программирование. Применение иреализация: Пер. с англ./Под ред. А.П.Ершова. – М.:Мир, 1983. – 349 с.
7. Хювенен Э., Сеппянен Й. Мир Лиспа. В 2-х т. Т.1: Введение в язык Лисп и функциональное программирование. Пер. с финск. – М.:Мир, 1990. – 447 с.
8. Хювенен Э., Сеппянен Й. Мир Лиспа. В 2-х т. Т.2: Методы и системы программирования. Пер. с финск. – М.:Мир, 1990. – 319 с.

Навчальне видання

Месюра В.І.

ФУНКЦІОНАЛЬНЕ ТА ЛОГІЧНЕ ПРОГРАМУВАННЯ

Навчальний посібник

Оригінал-макет підготовлено автором

Редактор С.А. Малішевська

Підписано до друку 5.04.2001р
Формат 29,7х42 1/4 Гарнітура Times New Roman
Друк різографічний Ум. друк. арк. 5,21
Тираж 75 прим.
Зам. № 2001 - 111

Відруковано в комп'ютерному інформаційно-видавничому центрі Вінницького державного технічного університету
21021, м. Вінниця, Хмельницьке шосе, 95, ВДТУ, ГНК, 9-й поверх
Тел. **(0432) 44-01-59**