НАЦІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ

ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК

КАФЕДРА ІНТЕЛЕКТУАЛЬНИХ СИСТЕМ

МЕТОДИЧНІ ВКАЗІВКИ

щодо виконання лабораторних робіт

з курсу “Технологія програмування та створення програмних продуктів”

для студентів 4-го курсу

спеціальності “Інтелектуальні системи прийняття рішень”

Частина 2. Функціональне програмування

КИЇВ — 2009

Методичні вказівки щодо виконання лабораторних робіт з курсу “Технологія програмування та створення програмних продуктів”. Частина 2. Функціональне програмування / Укл. Баклан І.В., Степанкова Г.А. - К.: НАУ, 2009. - 40 с.

ЗМІСТ

ЛАБОРАТОРНА РОБОТА 1: Знайомство з інтерпретатором Ліспп

Ціль роботи:

У задачу лабораторної входить освоєння роботи з інтерпретатором, освоєння вбудованого отладчика й режиму покрокового виконання. Вивчення функціонального базису Ліспу, поняття S-виразів й різних способів їхнього подання. Створення першої програми на Ліспі. Освоєння редактора й різних способів завантаження Лісп програм для виконання.

Запуск інтерпретатора

Common Lisp реалізований на дуже великій кількості платформ. У рамках навчального посібника всі приклади випробувані на PC на базі Debian Linux. Як реалізація обрана продукт: GNU Common Lisp (http://clisp.sourceforge.net/).

Запуск інтерпретатора виконується за допомогою команди clisp.

Після запуску інтерпретатора з'являється рядок запрошення:

```
[1]>
```

Інтерпретатор готовий до виконання команд.

Вправи

Необхідно виконати деякі з запропонованих вправ, щоб освоїтися з інтерпретатором. Якщо виникають помилки - спробуйте нажати сполучення клавіш: ctrl-d і повторити набір. У версії для windows це сполучення може не працювати, тому можна спробувати ввести Abort. У кожному разі можна продовжувати виконання вправ.

Вправа	**Результат**	**Увага**
(+ 3 4)	7	Наявність пробілів після + та між 3 та 4 істотна. Пробіл у Ліспі є роздільником між іменамифункцій та аргументами.
(* (+ 7 8) 9)	135	У Ліспі прийнятий так званий префіксний спосіб запису виразів. На початку йде запис деякого функціонального символу, символу операції, потім через пробіл – аргументи.
(! 9)	362880	
(princ “hello, world!”)	hello, world! "hello, world!"	Функцію princ можна використати для друку строкових виразів
(atom ’a)	T	Atom - предикат. Якщо аргумент є атомом (як у цьому випадку), то предикат повертає істину (T)
(atom ’(a b c))	NIL	Та повертає неправду в противному випадку (NIL)
(listp ’(a b c))	T	Listp – предикат списку. Якщо аргумент є

		списком – предикат повертає істину T, у противному випадку NIL – неправда.
(defun f (x) (+ x x))	F	Ми визначаємо функцію F. У результаті визначення інтерпретатор повертає F – ім'я функції, тим самим підтверджуючи коректність визначення (синтаксичну коректність)
(f ’1)	2	Використаємо знову певну функцію

Помилка

У процесі програмування неминучі помилки. Вони можуть з'явитися при аналізі задачі, побудові рішення й навіть просто в процесі безпосереднього програмування, кодування, коли рішення повинне бути написане на деякій алгоритмічній мові.

Помилок, на відміну від правильних рішень, дуже багато, тому приведемо лише найпоширеніші з них.

Приклад помилки при роботі в інтерпретаторі Common Lisp

```
[1]> (a 1 2)
*** - EVAL: the function A is undefined
1. Break [2]>
```

В Common Lisp немає визначеної функції з ім'ям “a”, тому інтерпретатор видає діагностику про те, що функція A не визначена. За замовчуванням, інтерпретатор не розрізняє заголовні й прописні букви, тому ім'я “a” й “A” означають ту саму функцію.

Щоб вийти з відладчика (debugger) необхідно нажати комбінацію клавіш: Ctrl-D.

```
[1]> q
*** - EVAL: variable Q has no value
1. Break [2]>
```

У цьому випадку помилка відбулася через те, що в контексті виконання форми q змінна q не зв'язана ні з яким значенням.

Ще однією класичною помилкою є неправильний запис Лісп виразів.

Не можна забувати, що якщо Ви хочете запустити функцію F з аргументами 5, атомом A, і результатом виразу (list 'a 'b), те робіть це в такий спосіб:

(F 5 'A (LIST 'A 'B))

Істотна наявність пробілів між F й 5 й 'A. А також те, що аргументи до функції F ідуть на одному рівні й не вкладені у додаткові дужки, як це робиться в Паскалі або C.

Функціональний базис

Після того, як ми зробили перші кроки в роботі з інтерпретатором, розглянемо common lisp як мову програмування. Що таке мова програмування?

Мова програмування, формальна знакова система, використовувана для зв'язку людини з комп'ютером; призначена для опису даних (інформації) і алгоритмів (програм) їхньої обробки на комп'ютері.

S-вираз

Відповідно до визначення, ключовими поняттями для визначення мови програмування є поняття даних і програми.

Для подання даних у Ліспі використаються так звані S-вирази.

Атом - ідентифікатор довільної довжини. Серед атомів є один виділений: NIL - цей атом означає неправду, а також порожній S-вираз. У деякому змісті NIL є аналогом порожньої множини в теорії множин.

S-вираз (символічний вираз):

1) атом є S-вираз

2) (S-вираз-1 . S-вираз-2) - S-вираз

3) інших S-виразів немає

S-вираз1 називається **головою** (head) S-виразу, а S-вираз2 - його **хвостом** (tail).

Якщо в другому пункті виразу S-виразу по обидві сторони від крапки - атоми, то таку конструкцію називають крапковою парою (dot pair).

Спосіб запису S-виразів, у якому використовуються тільки крапкові пари називається дотом-нотацією (dot-notation).

Іноді він занадто громіздкий, тому використовуються й інші способи запису S - виразів, які спрощують сприйняття.

Розглядаються наступні нотації:

- крапкова нотація (dot-notation) Крапкова нотація походить із індуктивного визначення S-виразу.
- облікова нотація (list-notation) це спрощення крапкової нотації, полягає в наступному: (a . (b . nil)) = (a b) - це називається **списком**.
- змішана нотація це щось середнє між dot й list нотаціями (коли хочемо, тоді позбуваємося від крапки).

Облікова нотація

В основі облікової нотації лежить поняття списку.

Список:

NIL - список

якщо x - список, а y - атом або список, то (y. x) - список.

Зображується список з елементів A,B,C у такий спосіб (A B C). У крапковій нотації це виглядає в такий спосіб: (A . (B . (C . Nil))).

Таким чином, правило переходу від крапкової нотації до облікової можна виразити в еквівалентному виді ".(" заміняється на " " - пробіл, “. Nil” опускається.

Змішана нотація

Є компромісом між крапковою й обліковою нотаціями. Компроміс ґрунтується на міркуванні наочності.

Крапкова нотація	**Змішана нотація**
сонцеб	сонце
NIL	NIL
пекуче	пекуче
(пекуче . сонце)	(пекуче . сонце)
(сонце . NIL)	(сонце)

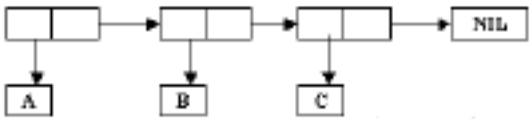
(пекуче . (сонце . NIL))	(пекуче сонце)
(пекуче . (сонце . :-Ь))	(пекуче сонце . :-Ь)

Розглядають також графічну нотацію.

Графічна нотація

Графічна нотація зв'язана безпосередньо з поданням списків у пам'яті комп'ютера.

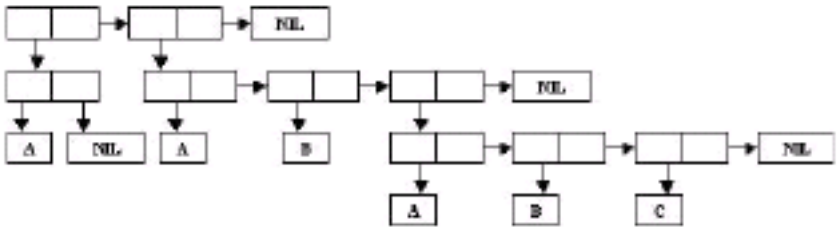
Розглянемо подання списку (A B C) у графічній нотації:



Розглянемо подання списку ((A) (A B (A B C))) у графічній нотації.

Зауваження

Атоми в системі Ліспі унікальні, тобто якщо необхідно кілька разів послатися на атом A, те щораз це та сама адреса. Крім того, атом NIL, як правило, є заглушкою, тобто не відбувається посилання на деяку адресу. Можливі різні варіації графічної нотації. Вона дозволяє більш наочно представити процеси, що відбуваються при обробці списків.



Ліспі – мова програмування, у якої дані й програма мають однакове подання. Таким чином, будь-який S-вираз можна намагатися інтерпретувати як програму й, навпаки, будь-яку програму можна розглядати як дані. Серед всіх можливих S-виразів виділяють ті вирази, які можуть бути обчислені інтерпретатором Ліспу. Ці S-вирази називають формами.

Функції базису

Перейдемо безпосередньо до функцій, що становлять функціональний базис Ліспу. У першу чергу це функції роботи зі списками. Логічно функціональний базис можна розділити на наступні категорії

Робота зі списками

Предикати

Розгалуження

Визначення функції й лямбда-вираження

Перетворення даних і програм

Для розширення кругозору будуть перераховані деякі арифметичні функції.

Робота зі списками

У цю категорію входять функції: CAR, CDR, CONS.

(CAR L)

Функція повертає голову списку L.

```
(CDR L)
```

Функція повертає список L без першого елемента (хвіст списку L).

```
(CONS E L)
```

Функція додає до L елемент E (скласти список, голова якого E, хвіст — L).

Розглянемо *приклади*:

```
> (car '(a b c))
>A
> (car '(ПЕКУЧЕ СОНЦЕ))
>ПЕКУЧЕ
```

Значення апострофа перед списком буде пояснено пізніше, коли ми розглянемо функції, що ставляться до категорії "перетворення дані - програма". Коротко: апостроф показує інтерпретаторові, що те, що йде слідом за ним - суть дані, які немає необхідності намагатися обчислити.

```
> (cdr '(a b c))
> (B C)
> (cons 'a 'b)
> (A . B)
```

Нехай x - деякий список. У загальному випадку вірно:

```
> (cons (car 'x) (cdr 'x))
>x
```

Предикати

У цю категорію входять наступні функції: ATOM, EQ, NULL.

Предикат суть функція, результат якої - логічне значення. У Ліспі є дві константи: NIL й T. Під істиною розуміється будь-яке значення відмінне від NIL.

```
(ATOM E)
```

E - атом, тоді значення функції T, у противному випадку NIL

Приклади:

```
>(atom 'a)
>T
>(atom '(a))
>NIL
(EQ A B)
```

Значення функції є T, якщо A й B - рівні атоми, NIL у противному випадку.

Приклади:

```
> (eq (car '(ПЕКУЧЕ СОНЦЕ)) 'пекуче)
```

```
T
  > (eq 'сонце 'пекуче)
NIL
```

Коментар до прикладів:

Як ми бачили (car '(ПЕКУЧЕ СОНЦЕ)) дає в результаті атом ПЕКУЧЕ, що, мабуть, дорівнює атому 'пекуче

```
(NULL E)
```

Предикат повертає T, якщо E - порожній список й NIL у противному випадку.

Приклади:

```
> (null (cdr '(b)))
T
> (null '(b))
NIL
```

Розгалуження

Розгалуження у функціональному базисі реалізується за допомогою COND.

```
(COND (L1 E1)(L2 E2)...(LN EN)(T ENN))
```

Якщо L1 не NIL, то E1 й обчислення припиняються, як значення форми повертається значення E1. Якщо L2 не NIL, то E2. обчислення припиняються, як значення форми повертається значення E2. … Інакше обчислюємо ENN.

```
>(cond
 ((= 4 5) (princ "4 = 5"))
 ((/= 4 5) (princ "4 <> 5"))
 (t (princ "hello world"))
)
4 <> 5
"4 <> 5"
```

У цьому випадку L1 = (= 4 5), L2 =(/= 4 5)

```
> (= 4 5)
NIL
> (/= 4 5)
T
```

Визначення функції й лямбда-виразу

Визначення функції й лямбда-виразу виконується за допомогою DEFUN й LAMBDA.

Визначення нової функції:

```
(DEFUN ім'я_функції (параметри)
вираз1
```

```
вираз2
...
вираз)
```

Під "виразом" розуміється будь-яка форма. Наприклад, виклик функції - це форма, значення її - значення функції на вхідних даних.

Приклад: визначення функції факторіал.

Аналіз:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & \text{если } n \neq 0 \end{cases}$$

Визначення:

```
> (defun N! (n)
  (cond
   ((eq n 0) 1)
   (t (* n (N! (- n 1))))
  )
 )
N!
```

Тепер визначення можна використати:

```
> (N! 45)
119622220865480194561963161495657715064383733760000000000
```

У Ліспі існує можливість написання безіменних функцій - лямбда-виразів.

Синтаксис лямбда-виразів аналогічний синтаксису DEFUN, за винятком того, що замість DEFUN використається LAMBDA й ім'я функції опускається.

```
> (lambda (x) (+ x x) )
#<CLOSURE :LAMBDA (X) (+ X X)>
```

Для того, щоб використати лямбда-вираз організують лямбда-виклик, що має наступний синтаксис:

```
(«лямбда-вираз» . «список фактичних параметрів»)
```

Приклад лямбда-виклику

```
> ((lambda (x) (+ x x) )10)
20
> ((lambda (x y) (+ (* x x) (* y y))) (+ 1 2) ( + 2 3 ))
34
```

Останній лямбда-виклик обчислює суму квадратів 3 й 5.

Перетворення: дані - програма

До цього розділу відносяться функції: EVAL, QUOTE.

```
(EVAL E)
```

Функція обчислює S-вираз E

```
(QUOTE E)
```

блокування обчислення (QUOTE E) рівносильне 'E

Спробуємо зрозуміти роботу функцій на прикладі:

```
> (car '( b c))
B
> '(car '( b c))
(CAR '(B C))
> (eval '(car '( b c)))
B
```

У такий спосіб функція QUOTE блокує обчислення. Іншими словами функція повертає S-вираз, що збігається з її аргументом. Функція EVAL - інтерпретує свій аргумент точно також як інтерпретатор обробляє те, що ми вводимо при роботі з Ліспом. EVAL - серце реалізації Ліспу. Функція має дуже велику потужність.

Розберемо роботу функції EVAL на прикладі:

```
> (defun ex_eval () (eval '(defun g (x)(+ x 2)))
(g 4)
)
> (ex_eval)
6
```

Ми ввели в систему функцію й зробили це програмно й можемо відразу її використати.

Арифметичні функції

```
(+ x1 x2 ... x)
```

повертає суму своїх аргументів. Аналогічно для "*", "/" й "-"

```
(EXPT x y)
```

x у ступені y

```
(MOD x y)
```

остача від x/y

```
(SQRT x)
```

квадратний корінь з x.

Застосування арифметичних функцій досить прозоро.

Розглянемо тепер більш докладно режими роботи інтерпретатора Ліспу.

Режими роботи інтерпретатора Common Lisp

У системі інтерпретатора розглядають три режими :

l. Режим інтерпретації

2. Режим налагодження

3. Режим покрокового виконання

Після запуску інтерпретатора командою clisp користувач попадає в режим інтерпретатора:

```
[1]>
```

При виникненні помилки або примусово при виклику деяких функцій (наприклад, break) користувач попадає в режим налагодження. Вхід у режим налагодження у випадку помилки обчислення:

```
[1]> q
*** - EVAL: variable Q has no value
1. Break [2]>
```

Вхід у режим налагодження примусово:

```
[1]> (break)
** - Continuable Error
Break
If you continue (by typing 'continue'): Return from BREAK loop
1.Break [2]>
```

У режим покрокового виконання можна потрапити тільки при виконанні макросу step:

```
[1]> (step (N! 6))
step 1 -i> (N! 6)
Step 1 [2]>
```

У цьому прикладі користувач попадає в режим покрокового виконання при покроковому обчисленні факторіалу від 6.

Відладчик

Якщо програміст помиляється, то інтерпретатор Ліспу переходить у режим налагодження. Також відладчик в Common Lisp можна задіяти за допомогою виклику деяких функцій, таких як BREAK й ін.

Крім відладчика, Common Lisp містить у собі можливість покрокового обчислення форми, що реалізується за допомогою макросу: STEP .

Найбільш істотні команди для режиму налагодження й покрокового виконання:

Команда	**Скорочення**	**Призначення**
Help	:h	Друк всіх доступних команд у поточному режимі
Abort	:a	Вихід з режиму налагодження / покрокового виконання
Where	:w	Показати поточний фрейм, у якому відбулася помилка
Error	:e	Надрукувати останнє повідомлення про помилку.
Up	:u	Розглянути фрейм, що породив поточний
Down	:d	Розглянути фрейм, що буде обчислюватися наступним

Top	:t	Перейти до розгляду фрейму, що є ініціюючим для даної серії обчислень
Bottom	:b	Перейти до розгляду фрейму, що викликав перехід у режим налагодження або покрокового обчислення
Redo	:rd	Заново обчислити відповідну форму або виклик функції. Ця команда може бути використана для продовження обчислень без їхнього аварійного припинення.

Відладчики по своїй природі тісно зв'язані із середовищами, у допомогу яким вони створені. Робота з відладчиком в Common Lisp відбувається в термінах внутрішньої організації обчислень в інтерпретаторі. Для організації обчислень рекурсивних функцій використається стек. Елементами стека є так звані фрейми, а також інші елементи. Як правило, кожному обчисленню форми відповідає деякий фрейм у стеці обчислень.

У режимі налагодження обчислення форм відбувається в статичному контексті поточного фрейму й динамічному контексті режиму налагодження. Тобто у режимі налагодження програміст має доступ до змінних, може їх змінити й потім продовжити обчислення вже з новими значеннями

Приклад роботи з відладчиком

Нагадаємо, що команда :e скорочення від error і команда друкує останнє повідомлення про помилку. Where і скорочення :w показує поточний фрейм. Redo і скорочення :rd приводить до повторної спроби обчислити поточний фрейм.

```
[1]> q
*** - EVAL: variable Q has no value
1. Break [2]> :e
Recent Error is: >> EVAL: variable Q has no value <<
1. Break [2]> where
EVAL frame for form Q
1. Break [2]> redo
*** - EVAL: variable Q has no value
1. Break [3]> (setq q 10)
10
1. Break [3]> :rd
10
[4]>
```

Як бачимо, обчислення форми проводиться в лексичному контексті форми й динамічному контексті відладчика — у відладчику ми додали значення змінної q, шляхом обчислення форми (setq q 10), тепер змінна q зв'язане й ще одне повторне обчислення форми приводить до успішного й природного завершення обчислень

Вправа

Спробуйте використати команди покрокового виконання для дослідження виклику

(N! 5)

Використання редактора

Запуск редактора виконується за допомогою функції ed. Редактор, що буде запускатися, визначається змінної оточення EDITOR.

Функцію можна запускати в такий спосіб: (ed x)

Якщо x - ім'я файлу, то ми переходимо до редагування файлу з ім'ям x. Якщо його ні, то він створюється.

Приклад використання:

```
(ed “test.lsp”)
```

Коротка довідка по використанню редактора Vi

Якщо Ви працюєте під Debian Linux, то швидше за все, за замовчуванням у Вас редактор Vi. Приведемо коротку довідку по використанню редактора.

Розглянемо систему команд 'vi'. Більшість команд - це одиночні клавіші або комбінації клавіш, які виконують прості функції редагування. 'vi' працює у двох основних режимах - у режимі "уведення тексту" й у режимі "команд".

Після запуску 'vi' виявляється в режимі "команд". Для переходу в режим "уведення тексту" необхідно нажати на клавішу 'a' або 'i' (обертаємо ваша увага на регістр клавіш). Після цього можна набирати текст.

Для переходу в командний режим натисніть на клавішу 'Esc'. Ця ж клавіша використається для скасування не до кінця набраної команди. Якщо ви неправильно ввели команду, редактор повідомить вам про цьому одиночним звуковим сигналом.

Щоб одержати список команд потрібно ввести :viusage. Після цього можна одержати довідку до команди, уводячи :viu <command>.

Наприклад:

Після уведення

```
:viu i
Key: i insert before cursor
Usage: [count]i
```

Збереження й вихід (виконуються в режимі команд).

:w	Зберегти текст без виходу з редактора
:w	Ім'я_файлу Зберегти текст у зазначеному файлі
:wq или :x	Зберегти текст і вийти з редактора
:q	Вийти з редактора. Якщо файл був модифікований, вам буде запропоновано для виходу без збереження використати команду :q!
:q!	Вийти з редактора без збереження тексту.

Вправа

Напишіть невеликий текст довільного змісту, у процесі редагування спробуйте освоїти 5 команд редактора VI, використовуючи убудовану довідкову систему.

Після того, як Ви створили файл у редакторі, його можна завантажити за допомогою команди

```
load
```

```
(load “test.lsp”)
```

Крім команди load за допомогою командного рядка можливо завантажити файл у систему:

```
clisp -i test.lsp
```

ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

Операції над списками

1) Функція COPY - створює в обліковій пам'яті другий екземпляр довільного S-вираження.

2) Функція LENGTH - повертає як значення довжину списку [*].

3) Функція APPEND - з'єднує два списки в один новий список.

4) Функція REMOVE - видаляє зі списку всі співпадаючі з даним атомом елементи й повертає як значення список із всіх елементів, що залишилися, [*].

5) Функція REMOVEF - видаляє зі списку перші входження даного елемента [*].

6) Функція REMOVEL - видаляє зі списку останній елемент [*].

7) Функція SUBSTITUE - заміняє всі входження даного елемента в списку на новий елемент [*].

8) Функція REVERSE - змінює порядок елементів у списку на зворотний [*].

9) Функція FIRST-ATOM - результатом функції є перший атом списку (в облік приймаються списки всіх рівнів).

10) Функція LAST - повертає останній елемент списку.

11) Функція ADDIFNONE - перевіряє, чи втримується заданий елемент (значення першого аргументу) у заданому списку (значення другого аргументу), і якщо ні, те додає цей елемент до списку [*].

12) Функція COLLECT - перегрупує елементи заданого списку так, щоб однакові елементи, якщо вони є в списку, стояли всі підряд [*].

13) Функція FLATTEN - усуває в довільному S-вираженні всі внутрішні дужки, а в крапкових вираженнях - і крапки, перетворюючи його в список атомів. Кількість і відносний порядок атомів у вираженні зберігаються.

14) Функція REVL - обертає список і розбиває його на рівні. Приклад: вихідний список - (a b c), що результирует список - (((c) b) a).

15) Функція DEVLEV1 - розбиває список на рівні. Приклад: вихідний список - (a b c), що результирует список - (a (b (c))).

16) Функція DEVLEV2 - розбиває список на рівні. Приклад: вихідний список - (a b c), що результирует список - (((a) b) c).

17) Функція DESTLEV1 - забирає рівні в списку. Приклад: вихідний список - (a (b (c))), що результирует список - (a b c).

18) Функція DESTLEV2 - забирає рівні в списку. Приклад: вихідний список - (((a) b) c), що результирует список - (a b c).

19) Функція REMSEC - видаляє зі списку кожен другий елемент [*].

20) Функція DEVPAIR - розбиває список на пари. Приклад: вихідний список - (a b c d ...), що результирует список - ((a b)(c d)...).

21) Функція MIX - чергує елементи двох списків-аргументів й утворить новий список. Приклад: вихідні списки - (a b ...) і (1 2 ...), що результирует список - (a 1 b 2 ...).

22) Функція DEPTH - обчислює глибину списку (найглибшої галузі).

Предикати

Предикати FORALL, FORSOME, FORODD - у цих предикатів по двох аргументу. Значенням першого аргументу повинен бути деякий список L, а другого (функціонального) аргументу - найменування або визначальне вираження функції P.

1) Предикат FORALL - приймає значення T лише в тому випадку, якщо функція P приймає значення "істина" (тобто не NIL) на всіх елементах списку L.

2) Предикат FORSOME - приймає значення T, якщо функція P приймає значення "істина" хоча б на одному елементі списку L.

3) Предикат FORODD - приймає значення T, якщо число елементів списку L, на яких функція P приймає значення "істина", нечетно.

4) Предикат ATOMLIST - перевіряє, чи є його аргумент списком (можливо, порожнім), складеним лише з атомів.

5) Предикат LISTP - приймає значення NIL, якщо задане вираження є атомом, відмінним від NIL, або вираженням, що може бути записано тільки в крапкових позначеннях. У противному випадку задане вираження є списком, якому можна записати, не прибігаючи до крапкових позначень на жодному з рівнів, і предикат приймає значення T.

6) Предикат ONELEVEL - перевіряє, чи є аргумент одноуровневым списком.

Порядок й упорядкування списків

1) Предикат ORDER - перевіряє, у якому порядку два заданих елементи зустрічаються в даному списку. Список (упорядочивающая послідовність) задається як значення третього аргументу функції ORDER. Якщо при перегляді елементів цього списку ліворуч

праворуч зустрічається елемент, що збігається зі значенням першого аргументу, а жоден з раніше переглянутих елементів не збігся зі значенням другого аргументу, то значення предиката дорівнює T, у всіх інших випадках воно дорівнює NIL.

2) Предикат ORDER1 - обчислюється так само, як й ORDER, однак, якщо жоден із заданих елементів не втримується в даному списку, то як значення предиката видається атом ORDERUNDEF.

3) Предикат LEXORDER - порівнює - елемент за елементом - два списки, задані як значення першого й другого аргументів. Якщо в якій-небудь позиції виявляються різні елементи, то вони рівняються між собою за допомогою предиката ORDER1, причому як третій аргумент (упорядочивающей послідовності) указується третій аргумент звертання до LEXORDER. Результат порівняння (T,NIL,ORDERUNDEF) видається як значення предиката LEXORDER. Якщо раніше, ніж зустрінуться різні елементи, вичерпається перший список, то видається результат T, якщо першим вичерпається другий список, то як результат видається NIL.

4) Предикат LEXORDER1 - відрізняється від LEXORDER тим, що значенням третього

аргументу для нього повинен бути список, кожна позиція якого у свою чергу є списком, що використається якщо буде потреба для порівняння відповідних позицій списків, заданих як значення перших двох аргументів.

5) Функція FIRST - серед елементів списку, заданого як значення першого аргументу, вибирає той, котрий раніше зустрічається в списку, заданому як значення другого аргументу. Якщо жоден з елементів першого списку не втримується в другому, то вибирається перший елемент першого списку.

6) Функція RANK - упорядковує список, заданий у якості її першого аргументу, переставляючи його елементи в тій послідовності, у якій вони зустрічаються в списку, що є значенням другого аргументу.

Пошук у списках

У цьому розділі зібрані функції, що вибирають зі списку елемент або елементи, що володіють заданою властивістю. Список задається у вигляді значення аргументу, що відповідає зв'язаної змінної L, властивість характеризується предикатом, найменування або визначальне вираження якого дане як значення аргументу, що відповідає зв'язаної (функціональної) змінної P.

1) Функція POSSESSING - утворить список із всіх елементів даного списку, що володіють заданою властивістю.

2) Функція SUCHTHAT - вибирає із заданого списку перший елемент, що володіє заданою властивістю. Якщо такого елемента ні, то виробляється значення NIL.

3) Функція SUCHTHAT1 - перевіряє, чи втримується в даному списку хоча б один елемент із заданою властивістю. Якщо так, то в момент виявлення такого елемента як результат приймається значення четвертого аргументу функції SUCHTHAT1. Якщо ні, то результатом є значення третього аргументу.

4) Функція SUCHTHAT2 - перевіряє, чи втримується в даному списку хоча б один елемент із заданою властивістю. Якщо так, то до хвоста заданого списку, починаючи зі знайденого елемента, застосовується функція, найменування або визначальне вираження якої задане як значення третього аргументу (функціонального). Якщо ні, то виробляється значення NIL.

5) Функція FIRST-COIN - повертає перший елемент, що входить в обидва списки X й Y, у противному випадку - NIL.

Висновок

У лабораторній роботі були розглянуті питання запуску інтерпретатора Лісп й роботи з ним. Освоєно базові навички створення програм за допомогою редактора і їхнє налагодження за допомогою вбудованого відладчика й режиму покрокового обчислення. Вивчено ідеологічну основу мови Лісп, а саме, поняття S-виразів, різні нотації для їхнього подання. Вивчено функціональний базис Ліспу, що по своїй обчислювальній потужності дорівнює будь-яким його розширенням.

ЛАБОРАТОРНА РОБОТА 2: Техніки функціонального програмування

Ціль роботи:

Задачею є знайомство з техніками функціонального програмування. Вивчення поняття рекурсії. Оволодіння навичками використання нагромаджуючого параметра. Знайомство з поняттям чистої й тотальної функції. Дослідження взаємини цих понять і властивості аддитивного налагодження системи функцій. Знайомство з універсальним програмуванням і ледачими обчисленнями. Вирішення задач.

Техніки функціонального програмування

Рекурсія

Функція є рекурсивною, якщо в її визначенні утримується виклик цієї функції.

На рекурсії заснована декомпозиція задачі на підзадачі, вирішення яких, наскільки це можливо, намагаються звести до вже вирішеного або до розв'язуваного в даний момент задачі.

Рекурсія близька до апарата математичної індукції, що визначає зручність реалізації індуктивних визначень.

Для рекурсивних функцій характерна наявність термінальної галузі. Як правило, рекурсивна функція реалізує розбір випадків, деякі з них тягнуть продовження процесу обчислення через виклик рекурсивної функції, інші не прибігають до рекурсії, вони називаються термінальними галузями.

Розглянемо задачу обчислення факторіала. Допустимо, що нам необхідно написати функцію, що повертає значення відповідно до наступного визначення:

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

Якщо аргумент функції n дорівнює 0, то як значення функції необхідно повернути 1, у противному випадку продовжити обчислення, шляхом виклику композиції функції множення й функції обчислення факториала.

```
(defun Факторіал (n)
(cond
 ((= n 0) 1) ; <- термінальна галузь
 (t (* n (Факторіал (- n 1))))))
```

Вправа

Реалізуйте функцію, протестуйте в інтерпретаторі Ліспа.

Класифікація рекурсії

Феномен рекурсії досить складний. Розглянемо види рекурсивних функцій.

Посилання функції на саму себе відбувається в деякому виразі в тілі функції. Якщо цей вираз визначає результат функції, то говорять про рекурсії за значенням. Якщо ж як результат функції повертається значення деякої іншої функції й рекурсивний виклик бере участь в

обчисленні аргументів цієї функції, то говорять про рекурсії по аргументах.

Розглядають наступні способи організації рекурсії:

l. Проста рекурсія

2. Паралельна рекурсія

3. Взаємна рекурсія

4. Рекурсія високих порядків

Говорять про просту рекурсію, якщо виклик функції зустрічається в деякій галузі лише один раз. Проста рекурсія часто еквівалентна циклу.

Загальний вид простої рекурсії:

```
(defun f (...)
(cond ((...) (...)) ((...)( ...(f ...) ...)) ((...)(...)) )
)
```

Визначення функції Факторіал є простою рекурсією по аргументах.

Загальний вид паралельної рекурсії:

```
(defun f ... (g ... (f ...) ... (f ...) ...) ...)
```

Таким чином, при паралельній рекурсії тіло визначення функції f містить виклик деякої функції g, кілька аргументів, який є рекурсивними викликами функції f. Говорять про взаємну рекурсію, якщо визначення функції f містить виклик деякої іншої функції g, що у свою чергу містить виклик функції f.

Загальний вид взаємної рекурсії:

```
(defun f ... (g ...)...)
(defun g ... (f ...)...)
```

Ясно, що в загальному випадку, кількість функцій, залучених в організацію взаємної рекурсії, може бути більше, ніж дві.

Рекурсією більше високих порядків називають спосіб організації рекурсії з наступним загальним видом функції:

```
(defun f ... (f ...(f ...)...) ...)
```

Використання параметрів функції

У функціональному програмуванні уникають використання глобальних змінних. Замість цього намагаються використати параметри функцій. Виявляється, що використання параметрів замість глобальних змінних приводить до більше надійного й продуктивного методу програмування.

Накопичувальний параметр

Використання накопичувального параметра рекурсивної функції дозволяє створити контекст до серії обчислень, породжуваної рекурсивним викликом. Такий підхід є більше коректним у порівнянні з використанням деякої глобальної змінної.

Накопичувальний параметр у випадку рекурсії за значенням являє собою додатковий параметр, що використовується замість глобальної допоміжної змінної. Обчислення виробляються шляхом формування цього додаткового аргументу, тому такий підхід називається формування аргументу. У випадку рекурсії по аргументах рекурсивний виклик бере участь в обчисленні деякої функції FUNC. Рішення формується в аргументі цієї функції

і являє собою деяку композицію формул, що використовується в процесі обчислення рекурсивного виклику, тому такий підхід називається композиція формул.

Розглянемо приклад використання накопичувального параметра.

Композиція формул

```
(defun N!-1 (x)
 (cond
  ((= 0 x) 1)
  (t (* x (N!-1 (- x 1))))))
```

Формування аргументу

```
(defun h-N! (x help)
 (cond ((= 0 x) help)
   (t (h-N! (- x 1) (* x help) ))))
```

Визначення необхідно загорнути:

```
(defun N!-2 (x) (h-N! x 1))
```

Вправа

Функція time повертає час й об'єм пам'яті, витрачений на обчислення expression. Синтаксис:

```
(time <expression>)
```

Проведіть порівняльний аналіз двох підходів: композиція формул і формування аргументу. Спробуйте пояснити отримані результати.

Перетворення системи функцій

Любий виклик системи рекурсивних функцій еквівалентний лямбда-виклику лямбда-вираз з функціональними аргументами. Розглянемо систему взаємнорекурсивних функцій:

```
(defun f (x)
 ( cond ((< x 0) x)
    (t (g (- x 2))) ))
(defun g (x)
 ( cond ((< x 0) x)
    (t (f (- x 5))) ))
```

Перетворимо виклик (f 6) в еквівалентний лямбда-виклик.

Вправа

З'ясуєте чому дорівнює виклик (f 6) визначите область значень функції f. Чи збігається вона з областю значення функції g?

Параметризуємо функції f й g. Це робиться шляхом введення двох додаткових параметрів:

```
(defun f (x func func-self)
 ( cond ((< x 0) x)
    (t (funcall func (- x 2) func-self func )) ))
```

```
(defun g (x func func-self)
 ( cond ((< x 0) x)
    (t (funcall func (- x 5) func-self func )) ))
```

Тепер можна позбутися від імен і вже перетворювати виклик (f 6). Робиться це в такий спосіб: у кажому визначенні defun <ім'я^-функції> заміняється на lambda. При організації виклику необхідно правильно зв'язати функціональні аргументи з лямбда-виразами.

```
>(funcall
   #'(lambda (x func func-self)
       ( cond ((< x 0) x)
          (t (funcall func (- x 2) func-self func )) ))
   6
   #'(lambda (x func func-self)
       ( cond ((< x 0) x)
        (t (funcall func (- x 5) func-self func )) ))
   #'(lambda (x func func-self)
       ( cond ((< x 0) x)
         (t (funcall func (- x 2) func-self func )) ))
)
-1
```

Вправа

Перетворіть виклик (N! 6) до лямбда-виклику деякого виразу. Результати протестуйте на інтерпретаторі.

```
 (defun N! (n)
  (cond
  ((eq n 0) 1)
  (t (* n (N! (- n 1))))))
```

Чисті й тотальні функції

Чиста функція - функція, що не залежить від контексту й не змінює його в процесі обчислення.

Прикладом функції, що залежить від контексту й змінює його є наступна функція:

```
 > (defun f (x) (setq q (+ q x)) q)
 F
 > (setq q 10)
 10
 > (f 5)
 15
 > (f 5)
 20
```

Який з двох виразів у тілі функції f змінює зовнішній контекст?

Чиста функція по своїй природі дуже схожа на функції, які зустрічаються в математиці. У будь-який момент обчислення функції при однакових вхідних даних результати обчислення теж однакові. Це дозволяє один раз перевіривши правильність функції бути впевненим, що вона буде завжди правильно працювати. Описана властивість невірно для функцій, відмінних від чистих, що демонструє розглянутий приклад.

Чисті функції мають властивість аддитивного налагодження. Ця властивість полягає в тому, що складність налагодження системи чистих функцій при додаванні ще однієї чистої функції складається зі складності тестування системи й складності тестування функції. Для систем не чистих функцій це твердження не вірно.

Рекомендується програмувати в чистих функціях, тому що вони легше піддаються тестуванню й налагодженню. Навіть у випадку, коли інформаційній системі потрібні які-небудь змінні стани, які, як правило, є динамічними змінними, варто оформляти їх у вигляді параметрів, що характеризують стан системи. Якщо змінна стану приймає кілька значень під час обчислення функції, то це може говорити про невдалу декомпозицію функцій.

Вправа

Уявіть собі в системі функцій усе є чистими крім однієї. Чи володіє ця система властивістю аддитивного налагодження?

Для того, щоб писати функції, які не змінюють зовнішній контекст, досить користуватися статичним контекстом обчислення, тобто для конкретної функції використати вираз, який містить лише імена параметрів і функцій.

Для того, щоб писати функції, які не залежать від зовнішнього контексту часто прибігають до методу побудови лексичного замикання. Лексичне замикання не залежить від контексту в момент його використання.

Лексичне замикання - функція + контекст у момент визначення функції. Замикання можна використати як функціональний аргумент. Лексичне замикання може бути побудоване з використанням функції function.

Усюди певні функції називаються тотальними функціями. Це означає, що будь-який параметр буде коректно оброблений. Природно, що це підвищує надійність програмування в цілому, однак, не завжди написання таких функцій можливо. Тому при написанні функцій варто приділяти особливу увагу типу функції - специфікації області визначення й значення цієї функції.

Універсальне програмування

Метод програмування, у якому зовнішні до програми дані використаються з метою керування роботою програми або самі інтерпретуються як програма, називається програмуванням, керованим даними.

За допомогою програмування, керованого даними, можна створити динамічні програми, дії яких визначаються в момент обчислення залежно від вступників на обробку даних. Такі динамічні програми більше універсальні й найчастіше представляють абстракцію того або іншого процесу.

Розглянемо приклад:

```
(defun tester ( input output test-function eq-p)
 (cond (( funcall eq-p (apply test-function input) output )
   (print “тест пройдений”))
  (t (print “тест провалений”))
```

)

)

Приклад демонструє «абстрактний» тестувальник - програму, що використовує деякі, заздалегідь певні вхідні дані (input) і відповідні їм вихідні дані (output). Ця програма застосовує функцію, що тестується (test-function), до вхідних даних і деяким способом (eq-p) порівнює отриманий результат з еталонними вихідними даними (output).

Вправа

Опишіть ситуацію, у якій можна використати описану функцію. Реалізуйте її в середовищі Common Lisp.

Функції вищих порядків

Аргумент, значенням якого є функція, називають функціональним аргументом (functional argiment), а функцію, що має функціональний аргумент - функціоналом (functional).

Функція може вироблятися функцією. Такі функції називають функціями з функціональним значенням (function valued).

Функції вищих порядків - потужна техніка, ця техніка лежить в основі парадигми універсального програмування, що дуже добре реалізується в рамках парадигми функціонального програмування.

Розглянемо приклад функції, що як результат повертає іншу функцію:

(defun addnn (n) (lambda (y) ((lambda (x y) (+ x y)) n y)))

Приклад використання:

> (funcall (addnn 6) 7)

13

> (addnn 6)

#<CLOSURE :LAMBDA (Y) ((LAMBDA (X Y) (+ X Y)) N Y)>

Цей вираз є поданням функції в інтерпретаторі. Функція addnn як свій результат повертають унарну функцію, що до свого аргументу додає число, що залежить від аргументу функції. Процес обчислення виразу (addnn 6) є ситуацією, коли інтерпретатор сам робить лексичне замикання, що приводить до того, що n у контексті обчислення лексичного замикання буде пов'язане з 6.

Відображуючі функції

Відображуючі функції - функції, які передаються функціоналам у якості їхніх функціональних аргументів. Техніка використання функціоналів більш докладно описана в лабораторній роботі “Функціонали”. Як і для функцій вищих порядків важливо знати тип відображаючої функції для того, щоб коректно застосувати до деяких даних.

У стандарті Common Lisp-а є ряд вбудованих функцій, що відображають, наприклад: mapcar. Продемонструємо роботу цього функціонала:

> (mapcar #'(lambda (x) (+ 5 x)) '(1 2 3 4 5))

(6 7 8 9 10)

У цьому випадку, роботу відображаючої функції можна розглядати, як роботу простого циклу по елементах списку-аргументу.

У загальному випадку техніка відображаючих функцій має більш широке застосування.

Ледачі обчислення

Тут буде освітлена ідея й поставлені питання, які приводять до необхідності використання цієї техніки.

Коли виникає задача обчислення деякого виклику функції, то виявляється, що обчислювати можна по-різному. В абстрактній формі це питання досліджується в рамках так званого лямбда-обчислення

```
((lambda (x y) (if (> x 0) x y )) (+ 4 5) (/ 6 0) )
```

У виразі декілька підвиразів, які можна обчислювати:

всі цілком (це виклик^-виклик-лямбда-виклик), (+ 4 5), (/ 6 0).

Питання: який з підвиразів обчислювати першим?

Часто розглядають наступні стратегії: нормальний порядок редукцій (ледачі обчислення), апликативний порядок редукцій.

Ідея ледачих обчислень полягає в тім, щоб обчислювати вирази тільки тоді, коли це необхідно. Логічний розвиток цієї ідеї полягає в тім, щоб одержати повний контроль над процесом обчислення виразів. Це значить, що програміст, за своїм розсудом може, як припинити обчислення, так і продовжити обчислення того або іншого виразу.

Ідея апликативного порядку редукцій полягає в тім, щоб обчислювати параметри функцій відразу, не затримуючи обчислень. Апликативний порядок редукцій дозволяє поводити обчислення більш ефективно, однак, у розглянутому прикладі, ця стратегія заведе в тупик:

```
> ((lambda (x y) (if (> x 0) x y )) (+ 4 5) (/ 6 0) )
*** - division by zero
1. Break >
```

Якби ми дотримувалися нормального порядку редукцій, то аргументи виразу-лямбда-вираз не обчислювалися, а вираз передався цілком усередину виразу-лямбда-вираз. Потім при обчисленні умови у форми if було б обчислене вираження (+ 4 5), що приводить до того, що обчислювати другу галузь, а значить і вираз (/ 6 0) не потрібно й обчислення благополучно приведе до результату:

```
> ((lambda (x y) (if (> x 0) x y )) (+ 4 5) '(/ 6 0) )
9
```

Ледачі обчислення дозволяють організувати роботу з нескінченними послідовностями даних. Це робиться шляхом затримки й поновлення обчислення елементів цієї нескінченної послідовності.

Приведемо приклад функції, що породжує ряд натуральних чисел:

```
> (defun numbers (x)(function ( lambda () (setq x (+ x 1)))))
NUMBERS
> (setq next (numbers 0))
#<CLOSURE :LAMBDA NIL (SETQ X (+ X 1))>
> (funcall next)
1
```

> (funcall next)

2

> (funcall next)

3

> (funcall next)

4

Послідовно продовжуючи обчислення ми можемо одержати будь-яке натуральне число.

Вправа

Застосуйте описані техніки функціонального програмування при вирішенні наступних задач:

1. Задача: Довжина списку

Написати функцію LEN , що повертає довжину списку, переданого функції як параметр.

Приклад:

;(LEN '(A B C)) поверне 3

;(LEN NIL) поверне 0

2. Задача: Обіг списку

Написати функцію REV, що бере список як аргумент і повертає звернений список як результат.

Приклад:

;(REV NIL) поверне NIL

;(REV '(A B C)) поверне (C B A)

;(REV '(A B (C) D (T))) поверне ((T) D (C) B A)

3. Задача: В один рівень

Написати функцію IN-ONE-LEVEL , що вибудовує атоми в багаторівневому списку в один рівень.

Приклад:

;(IN-ONE-LEVEL '(A B C)) поверне (A B C)

;(IN-ONE-LEVEL '(A (B (C)))) поверне (A B C)

4. Задача: Установите які значення приймає функція на цілих числах від 1 до 101?

5. Задача: Числа Фібоначі

Написати функцію Fib, що по аргументу n повертає n-е число Фібоначі. Числа Фібоначі визначаються в такий спосіб:

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

Приклад:

;Fib(0) поверне 1

```
;Fib(1) поверне 1
;Fib(4) поверне 5
;Fib(5) поверне 8
```

Класифікуйте вид рекурсії побудованих вами рішень. Спробуйте специфікувати тип цих функцій.

ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

Операції над множинами

Списки розглядаються як множини своїх елементів - порядку елементів у списку не надається значення, а два або більше однакових елементи списку розглядаються як один елемент множини.

1) Функція SETOF - для кожного повторюваного елемента виключає зі списку всі входження, крім одного.

2) Функція MAKESET - робить те ж, що SETOF, але описана через PROG. Порядок елементів у результуючому списку виявляється іншим.

3) Функція DIFLIST - обчислює різниця множин X\Y. Інакше кажучи, вона виключає зі списку, заданого як значення першого аргументу функції всі елементи, що зустрічаються в списку, представленому значенням другого аргументу.

4) Функція SUBSET - обчислює предикат "множина X є підмножиною множини Y". Інакше кажучи, вона виробляє значення T, якщо кожен елемент списку, заданого як перший аргумент функції, утримується в списку, представленому значенням другого аргументу.

5) Функція UNION - обчислює об'єднання двох множин. Значення функції являє собою список всіх виражень, що є елементами хоча б одного із заданих списків. Якщо кожний із заданих списків не містив повторюваних елементів, то в результуючий список кожен елемент увійде лише один раз.

6) Функція LUNION - поєднує множини, задані як елементи списку, що є значенням аргументу функції.

7) Функція INTERSECTION - обчислює перетинання двох множин. Значенням функції є список всіх виражень, що входять елементами в обоє заданих списків. Якщо кожний із заданих списків не містить повторюваних елементів, то в результуючому списку елементи не будуть повторюватися.

8) Предикат EQUALSET - перевіряє, чи рівні множини, представлені двома заданими списками.

9) Функція CART - утворить декартово добуток двох заданих множин. Точніше кажучи, вона формує лексикографічно впорядкований список, елементами якого є всілякі списки, що містять по двох елемента кожний, причому перший елемент береться з першого, а другий - із другого заданого списку.

10) Предикат SETP - перевіряє, чи є список множиною, тобто чи входить кожен елемент у список лише один раз.

11) Функція SIMDIFF - формує множину з елементів, що не входять в обоє множин (симетрична різниця множин).

Висновок

Освітлені базові техніки функціонального програмування. Рекурсія і її класифікація. Поняття чистої й тотальної функції. Отримано навички використання накопичувального параметра. Властивість адитивного налагодження системи функцій порушується при введенні хоча б однієї функції, що залежить від зовнішнього контексту. Освячено ідеї таких технік програмування, як ледачі обчислення, функції вищих порядків та вібоюражаючі функцій.

ЛАБОРАТОРНА РОБОТА 3: Функціонали

Ціль роботи:

Метою є освоєння поняття функції вищого порядку. Вивчення категорій концепцій регулярної обробки даних і процесів. Оволодіння навичками обчислення типу функції й специфікації даних. Вивчення фунарг-проблеми й поняття лексичного замикання. Освоєння композиції функціоналів і методів її аналізу. Вивчення функціоналів, що входять у стандарт common lisp.

Універсальне програмування

Метод програмування, у якому зовнішні до програми дані використовуються з метою керування роботою програми або самі інтерпретуються як програма, називається програмуванням, керованим даними.

Передумовою використання даного методу є наявність у мові програмування програмних засобів для інтерпретації під час виконання програми, зовнішніх стосовно неї, даних як програма. Крім того, у загальному випадку необхідно вміти будувати з даних програми до їхнього виконання.

У традиційних трансльованих мовах така можливість відсутня. За допомогою програмування, керованого даними, можна створити динамічні програми, дії яких визначаються в момент обчислення залежно від вступників на обробку даних. Такі динамічні програми більше універсальні й найчастіше представляють абстракцію того або іншого об'єкта дійсності.

Концепції

Мова програмування - є засобом вираження нашої думки. Тому, мова повинна забезпечувати можливість створювати абстракції. Природно реалізовувати концепції, які перебувають у нас у голові.

Концепції можна розділити на дві категорії: концепції процесів і концепції регулярної обробки інформації. Таке ділення склалося історично. Комп'ютери були створені для обробки інформації. Язикові засоби постійно еволюціонували: концепція простого циклу була розширена циклом із предумовою і постумовою, потім з'явилися ітератори по регулярним, як правило, визначеним структурам і вже потім ітератори по довільних структурах даних. Загальний напрямок розвитку - узагальнення концепції регулярної обробки інформації.

Категорія концепцій регулярної обробки інформації входить у категорію концепцій процесів.

Для реалізації цих категорій у функціональних мовах програмування використовуються функції вищих порядків. Розглянемо докладніше це поняття.

Побудова концепції регулярної обробки списку

Розглянемо концепцію регулярної обробки елементів списку - функцію, що застосовує деяку функцію (fn) до елементів списку (list), а результати застосування конструює в новий список.

Ідею регулярної обробки списку можна записати в такий спосіб:

```
(defun mapcar~ (fn list)
 (cond ((null list ) nil)
```

```
(t (cons (funcall fn (car list)) (mapcar~ fn (cdr list))))))
```

Використовуючи визначення, наведене вище, можна реалізувати кілька процесів обробки списку, наприклад збільшення кожного елемента списку на одиницю

```
[1]> (mapcar~ #'(lambda (x) (+ x 1)) '( 1 2 3 4 5))
(2 3 4 5 6)
```

або подвоєння кожного елемента.

```
[2]> (mapcar~ #'(lambda (x) (* x 2)) '( 1 2 3 4 5))
(2 4 6 8 10)
```

Концепцію регулярної обробки списку можна узагальнювати. Узагальнення можливо в декількох напрямках. Наприклад, процес можна розглянути для списків довільної вкладеності, таким чином, визначити обробку довільного дерева. Іншим напрямком підвищення рівня абстракції є наступна задача.

Багаторівневий список можна трактувати як множину, елементами якої можуть бути інші множини. Наприклад, множині {A, B, {A, B}, C}, що складається з чотирьох елементів - A, B, C і множини, що складається із двох елементів: A й B можна поставити у відповідність наступний список: (A B (A B) C). Розглянемо задачу побудови функції mapr~, яка б по заданій функції fn і предикату pred, а так само списочному представленню множини set генерувала множину всіх підмножин вихідного, потім формувала список множин, істинних на предикаті pred, після чого до елементів отриманого списку застосовувалася б функція fn, а результати формувалися в підсумковий список, який би й повертався як значення обчислення функції.

Легко помітити, що концепція, реалізована функцією mapcar~ утримується в концепції map~, тому що необхідне поводження ми можемо одержати за умови, що pred має такий вигляд:

```
(defun pred-for-mapcar~ (set)
 (cond ((eq (length set) 1) t )
   (t nil)))
```

Цей предикат повертає істину в тому і тільки в тому випадку, якщо set - є одноелементною множиною.

Концепція тестування

Розглянемо концепцію процесу тестування. Розглянемо «абстрактний» тестувальник - програму, що використає деякі, заздалегідь певні вхідні дані (input) і відповідні їм вихідні дані (output). Ця програма застосовує функцію, що тестується (test-function), до вхідних даних і деяким способом (eq-p) порівнює отриманий результат з еталонними вихідними даними (output).

```
(defun tester ( input output test-function eq-p)
(cond (( funcall eq-p (apply test-function input) output )
  (print “тест пройдений”))
  (t (print “тест провалений”))))
```

Розглянемо приклад використання:

Допустимо, що два програмісти одержали одне й теж завдання написати функцію, що підсумує два числа. Розроблювач-1 написав sum1, а розроблювач-2 функцію sum2:

```
(defun sum1 (x y) (+ x y))
```

(defun sum2 (x y) (* x y))

Кожен програмний продукт тестується. Протестуємо результат роботи програмістів:

Тест-1 ми знаємо що 2+2=4, а також те, що порівнювати результат потрібно за допомогою предиката = , що здатний порівнювати два числа на рівність.

Тест-2 ми знаємо що 2+3=5, а також те, що порівнювати результат потрібно за допомогою предиката = , що здатний порівнювати два числа на рівність.

> (tester '(2 2) '4 'sum1 '=) "тест пройдений"	> (tester '(2 2) '4 'sum2 '=) "тест пройдений"
> (tester '(2 3) '5 'sum1 '=) "тест пройдений"	> (tester '(2 3) '5 'sum2 '=) "тест провалений"

Надалі можливо узагальнення концепції тестування. У концепцію можна включити платформу, на якій виробляється тестування, операційну систему, у якій виробляється тестування й т.д.

Вправа

Напишіть функцію, яка б реалізувала концепцію множення й додавання.

Поняття функції вищого порядку

Уточнимо термінологію:

Аргумент, значенням якого є функція, називають функціональним аргументом (functional argiment), а функцію, що має функціональний аргумент - функціоналом (functional).

Функція може повертатися функцією. Такі функції називають функціями з функціональним значенням (function valued).

Функціонали, функції й функціонали з функціональним значенням називаються функціями вищого порядку.

Функціонали

Техніка роботи з функціоналами заснована на понятті функції, що відображає, а це поняття у свою чергу на понятті відображення.

Говорять, що відображення існує, якщо задано пару множин і деяке правило зіставлення елементів тієї й іншої множини.

При визначенні відображень насамперед повинні бути ясні наступні питання:

1) що являє собою правило зіставлення;

2) як організоване дана відображувана множина;

3) яким способом виділяються елементи відображуваної множини.

Правило зіставлення формалізують, як правило, за допомогою функції, її називають відображуваною функцією. Відповіді на наведені питання дозволяють задати порядок перебору множини й метод передачі фактичних аргументів для обчислення відображуваної функції. При переборі множини відображувана функція буде застосована до кожного елемента множини.

У стандарті Common Lisp-а є ряд вбудованих функцій, що відображають, наприклад: mapcar.

Продемонструємо роботу цього функціонала:

```
> (mapcar #'(lambda (x) (+ 5 x)) '(1 2 3 4 5))
(6 7 8 9 10)
```

У цьому випадку, роботу функції, що відображає, можна розглядати, як роботу простого циклу по елементах списку-аргументу.

Функції вищого порядку

Досліджуємо поняття функції вищого порядку.

Функції вищих порядків використають наступний факт. Розходження між поняттями “дані” й “функція” визначається не на основі їхньої структури, а залежно від їхнього використання. Якщо аргумент використається у функції лише як об'єкт, що бере участь в обчисленнях, то ми маємо справу зі звичайним аргументом, що представляє дані. Якщо ж він використається як засіб, що визначає обчислення, тобто грає в обчисленнях роль лямбда-вираження, що застосовується до інших аргументів, то ми маємо справу з функцією. Використовуючи техніку маніпулювання параметрами легко переводити дані у функції й навпаки. Це визначає важливість такого механізму як засобу вираження концепцій.

Способи композиції функцій

Розглянемо можливі способи композиції функцій:

1 Звичайний виклик:

```
(defun f ... (g ...)...)
```

2 Рекурсивний виклик:

```
(defun f ... (f ...)...)
```

3 Вкладений рекурсивний виклик:

```
(defun f ... (f...(f ...)...)...)
```

4 Функціональний аргумент:

```
(defun f (... g ...) ... (funcall g ...)...)
```

5 Рекурсивний функціональний аргумент:

```
(defun f (... f ...) ... (funcall f ... f ...)...)
```

Функціонал, що одержує себе в якості аргументу, називають застосовуваною до самої себе або автоапликативною (self-applicative) функцією. Функцію, що повертає саму себе, називають авторепликативною(self-replicative).

Приклад автоапликативної функції:

```
> (defun N!_self (f n)
 (cond ((eq n 0)1)
    (t (* n (funcall f f (- n 1))))))
N!_SELF
> (N!_self #'!_self '3)
6
```

Приклад авторепликативної функції:

```
> ((lambda (x)(list x x)) '(lambda (x) (list x x)) )
((LAMBDA (X) (LIST X X)) (LAMBDA (X) (LIST X X)))
```

Привидите ще кілька прикладів автоаппликативных й авторепликативных функцій.

Тип функції

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

Функцію можна специфікувати шляхом специфікації множин, що представляють області визначення й значення цієї функції.

Наприклад,

(lambda (x)(x+1)) : N -> N

Функція додає одиницю до свого аргументу, а виходить, відображає множина чисел у множину чисел.

(lambda (x y)(x+y)) : N,N -> N

У цьому випадку пари чисел відображаються в одне, що представляє їхню суму.

Корисно ввести позначення: A - атоми, N - числа, L(X) - NIL або списки з елементів типу X, B - NIL або T, E - будь-який об'єкт.

Специфікуємо деякі функції з функціонального базису.

cons : X,L(X) -> L(X)

car : L(X) -> X

cdr : L(X) -> L(X)

eq : A,A -> B

Розглянемо приклад більше детальної специфікації:

atom : (A -> T) & (L(X) -> NIL)

У цьому випадку, специфікацію варто трактувати в такий спосіб: функція atom повертає відображає будь-який атом в істину, а список в nil. Цю ж функцію можна специфікувати у такий спосіб:

atom : E -> B

Ступінь деталізації специфікації залежить від конкретної задачі. Наприклад, при використанні функціоналів, подібних mapcar. Нас у першу чергу цікавить чи можливо коректне застосування функціонального аргументу до даних, тому від специфікації потрібно лише донести інформацію про те, якого роду об'єкти можуть безболісно подаватися на вхід до функції.

Неважко, використовуючи, наведені позначення специфікувати відображувану множину. Також можна вводити додаткові позначення, зручні для вас.

Фунарг-проблема

З використанням функцій, що відображають, зв'язана так звана фунарг проблема. Вона проявляє себе при використанні вільних змінних у визначеннях функціонального аргументу для відображувальної функції. У випадку використання вільних змінних можливі два типи поводження, які зв'язані насамперед з поняттям області дії й часу життя змінної. Розрізняють статичний контекст і динамічний (синоніми статична область дії й динамічна). Час життя може бути звичайний або нескінченний. У цих термінах зручно визначаються різні типи зв'язування імен і значень. У різних ситуаціях потрібні різні стратегії зв'язування, але в цілому рекомендується використати функції без вільних параметрів і без побічного ефекту.

Для того, щоб обійти фунарг-проблему використовуються лексичні замикання, які зв'язують вільні змінні зі значеннями з контексту моменту визначення лексичного замикання. У такий спосіб:

Лексичне замикання - функція + контекст моменту визначення функції.

Замикання можна використати як функціональний аргумент.

Лексичне замикання не залежить від контексту в момент використання замикання. Для того, щоб створити лексичне замикання в common lisp використовується функція function, крім того, середовище common lisp влаштоване так, що в ситуаціях коли лексичне замикання необхідне - воно виробляється автоматично.

Композиція функціоналів

Виклики функціоналів можна поєднувати в більш складні структури в такий же спосіб, як і виклики функцій, а їхню композицію можна використати у визначеннях нових функцій.

Наприклад, побудова декартова добутку:

```
(defun cartesian_product (x y)
 (apply #'append
  (mapcar
   #'(lambda (x) (mapcar
         #'(lambda (y) (list x y))
          y))
    x
   ))
 )
```

Використання:

```
> (cartesian_product '(a s d) '( e r t))
((A E) (A R) (A T) (S E) (S R) (S T) (D E) (D R) (D T))
```

Розберемося більш докладно в організації функції cartesian_product.

Для цього розглянемо фрагменти визначення:

```
> (apply #'append '((a)(b)(c)))
(A B C)
```

Нагадаємо, що функція apply має наступний синтаксис

apply function arg &rest more-arg

і застосовує функцію function до списку аргументів.

Продовжимо аналіз визначення з розгляду внутрішнього виразу:

```
> #'(lambda (x) (mapcar
         #'(lambda (y) (list x y))
          '(e r t)))
#<CLOSURE :LAMBDA(X)(MAPCAR #'(LAMBDA (Y) (LIST X Y)) '(E R T))>
```

Результатом обчислення є функція від x: (LAMBDA (X) (LIST (LIST X 'E) (LIST X 'R) (LIST X 'T))), що є наслідком семантики функції mapcar. Зрівняєте вираз:

```
> (funcall #'(lambda (x) (mapcar
        #'(lambda (y) (list x y))
         '(e r t))) 'a)
((A E) (A R) (A T))
```

та

```
> (funcall #' (LAMBDA (X) (LIST (LIST X 'E) (LIST X 'R)(LIST X 'T))) 'a)
((A E) (A R) (A T))
```

Зовнішній mapcar забезпечить регулярну обробку списку (a s d), що приведе до результату:

```
>   (mapcar
    #'(lambda (x) (mapcar
          #'(lambda (y) (list x y))
          '(e r t)))
    '(a s d)
   ))
(((A E) (A R) (A T)) ((S E) (S R) (S T)) ((D E) (D R) (D T)))
```

Застосування append приведе до необхідного результату.

Часткове обчислення функції

Можна привести інше визначення декартового добутку. Рішення, засноване на частковому обчисленні функції й використанні лексичного замикання. У відмінності від попереднього визначення, які ми розбирали “зверху вниз”, розглядаючи підвирази, часткові конструкції вже готового. Це визначення побудуємо “знизу нагору”.

```
> (mapcar
 #'(lambda (x) (lambda (y) (list x y)))
 '(a s d))
(#<CLOSURE :LAMBDA (Y) (LIST X Y)> #<CLOSURE :LAMBDA (Y) (LIST X Y)
> #<CLOSURE :LAMBDA (Y) (LIST X Y)>)
```

Як результат ми одержали список їхніх трьох функцій. Здається що це однакові функції, однак, вони є лексичними замиканнями й змінна x зв'язується зі значенням у момент визначення цього замикання, тому вони різні:

```
> (funcall (car(mapcar
 #'(lambda (x) (lambda (y) (list x y)))
 '(a s d))) '(e r t) )
(A (E R T))
> (funcall (cadr(mapcar
 #'(lambda (x) (lambda (y) (list x y)))
 '(a s d))) '(e r t) )
(S (E R T))
```

Всі ці функції є частково обчисленими, тобто тільки один аргумент - x одержав фактичне значення. Розгляд і використання таких функцій часто може спростити опис задачі.

Розглянемо тепер наступний вираз:

```
> (funcall #'(lambda (f)(funcall #'mapcar f '(e r t)))
(car (mapcar
 #'(lambda (x) (lambda (y) (list x y)))
 '(a s d))))
((A E) (A R) (A T))
```

Як видно з результату, для одержання необхідного результату (з точністю до застосування append) необхідно організувати регулярну обробку списку лексичних замикань, що робиться за допомогою mapcar.

Таким чином, що результуючим виразом є:

```
> (apply #'append (mapcar
#'(lambda (f)(funcall #'mapcar f '(e r t)))
(mapcar
 #'(lambda (x) (lambda (y) (list x y)))
 '(a s d))))
((A E) (A R) (A T) (S E) (S R) (S T) (D E) (D R) (D T))
```

Залишилося оформити цей вираз як функцію.

Вбудовані MAP-функціонали

Опишемо докладніше функціонали вбудовані в common lisp.

Функція map

Синтаксис:

```
map result-type function &rest sequences+ => result
```

Функція function викликається на всіх елементах послідовностей з індексом нуль, потім з індексом один і т.д. З результатів викликів формується результуюча послідовність типу result-type

Розглянемо приклади:

```
> (map 'list #'- '(1 2 3 4))
(-1 -2 -3 -4)
```

Результуючий тип - список, тому ми одержуємо список чисел, до кожного елемента якого застосована функція зміни знаку.

Наступний вираз повертає список із двох випадкових чисел від 0 до довжини рядка "01234567890ABCDEF"

```
[81]> (funcall (lambda (x)(list(random x)(random x)))
(length "01234567890ABCDEF"))
(11 3)
```

Використання цього списку у функціоналі map з результуючим типом string:

```
> (map 'string #'(lambda (x) (char "01234567890ABCDEF" x ))
```

```
(funcall (lambda (x)(list(random x)(random x)))
(length "01234567890ABCDEF")))
"27"
```

Таким чином, функціонал map повертає два довільних символи з рядка.

Функція mapcar

Синтаксис:

```
mapcar func &rest list+ => result-list
```

Функція func застосовується до 0-х елементів, потім до 1-х і т.д. Тобто func застосовується до голів списків, результат застосування збирається в результуючий список.

Приклад:

```
> (mapcar #'+ '(1 2 3) '(4 5 6))
(5 7 9)
> (mapcar #'list '(1 2 3)'(4 5 6))
((1 4) (2 5) (3 6))
```

Функція maplist

Синтаксис:

```
maplist func &rest list+ => result-list
```

Функція аналогічна mapcar, але func застосовується до хвостів списків list.

Приклад:

```
> (maplist #'list '(1 2 3)'(4 5 6))
(((1 2 3) (4 5 6)) ((2 3) (5 6)) ((3) (6)))
```

До MAP функціоналів ставляться також наступні функції: mapc, mapl вони аналогічні mapcar й maplist відповідно, за винятком того, що вони повертають перший список.

Приклад:

```
> (mapc #'list '(1 2 3)'(4 5 6))
(1 2 3)
> (mapl #'list '(1 2 3)'(4 5 6))
(1 2 3)
```

mapcan, mapcon аналогічні mapcar й maplist, однак об'єднання результатів відбувається не за допомогою функції list, а за допомогою деструктивної функції nconc.

ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

Функціонали

1) Функція APL-APPLY - застосовує кожну функцію fi списку f=(f1 f2 ... f), що є першим аргументом функції, до відповідного елемента xi списку x=(x1 x2 ... x), що є другим аргументом функції, і повертає список, сформований з результатів.

2) Функція MAPLIST - застосовує функцію, задану в якості її другого аргументу

(функціонального), послідовно до всього списку, заданій як значення першого аргументу, і до всіх списків, по черзі одержуваним відкиданням першого елемента від попереднього списку. Значенням функції MAPLIST є список отриманих результатів.

3) Функція MAPCAR - подібна функції MAPLIST і відрізняється від її тим, що задана функція застосовується до перших елементів тих списків, до яких вона застосовувалася б у випадку звертання до MAPLIST.

4) Функція MAP - має сенс, якщо функція F, задана в якості її другого аргументу, має який-небудь побічний ефект. Ця функція послідовно застосовується до тих же аргументам, що й у випадку функції MAPLIST, але вироблені значення ніяк не використаються й не зберігаються. Значення функції MAP дорівнює NIL (а якщо черговий аргумент, підготовлений для звертанні до функції F, виявиться атомом, те цьому атому).

Списки властивостей

1) Функція PUTPROP - поміщає в список властивостей атома, зазначеного в якості її першого аргументу, властивість, представлена значенням третього аргументу, з індикатором, заданим у вигляді значення другого аргументу. Як результат функція PUTPROP виробляє атом, список властивостей якого піддався зміні. Якщо в списку властивостей цього атома вже була властивість із даним індикатором, то ця властивість заміщається новим, індикатор не дублюється.

2) Функція GETPROP - витягає зі списку властивостей даного атома (значення першого аргументу) властивість із даним індикатором (значенням другого аргументу). Якщо такого індикатора в списку властивостей ні, то виробляється значення NIL.

3) Функція PROP - відрізняється від функції GETPROP тим, що вона видає як результат не властивість, пов'язане з даним індикатором (якщо воно є в списку), а весь хвіст списку властивостей, що випливає за знайденим індикатором. Якщо ж індикатор у списку властивостей не виявляється, то виробляється звертання до функції без аргументів, найменування або визначальне вираження якої повинне бути задане як третій аргумент PROP. Результат цього обігу видається в цьому випадку як результат звертання до PROP.

4) Функція REMPROP - видаляє зі списку властивостей даного атома (значення першого аргументу у звертанні до неї) властивість, постачена даним індикатором (значення другого аргументу), разом із цим індикатором. Якщо такого індикатора не було, то список властивостей не міняється. Значення функції збігається зі значенням першого аргументу.

5) Функція REMPROPS - видаляє всі властивості символу.

6) Предикат HASPROP - перевіряє, чи володіє символ (перший аргумент) даним індикатором (другий аргумент).

Робота з упорядкованими бінарними деревами

Упорядковане бінарне дерево складається з вузлів виду:

(<елемент> <ліве поддерево> <праве поддерево>).

У кожному вузлі виконана наступна умова: всі елементи з вузлів його лівого поддерева в деякому урорядочении (наприклад, по числовій величині або за абеткою) передують елементу з вузла й відповідно елементи з вузлів правого поддерева випливають за ним. Приклад:

(5 (3 (1 NIL NIL)

(4 NIL NIL)

(7 (6 NIL NIL)

(13 (11 NIL NIL)

(15 NIL NIL))))

1) Функція FINDBT - шукає в дереві даний елемент.

2) Функція ADDBT - додає в дерево даний елемент. (Зауваження: копіюйте дерево по шляху пошуку й підправляйте потрібне поддерево).

3) Функція PREDBT - виділяє в окреме (упорядковане) дерево з первісного дерева всі вузли, що передують даному елементу.

4) Функція POSTBT - виділяє в окреме (упорядковане) дерево з первісного дерева всі вузли, що випливають за даним елементом.

5) Функція UNIONBT - поєднує два впорядкованих дерева в одне загальне впорядковане дерево (Зауваження: використайте функції PREDBT й POSTBT).

Висновок

У лабораторній роботі розглянуте поняття функції вищого порядку, авторепликативної й автоапликативної функцій. Вивчено категорії концепцій регулярної обробки даних і процесів. Функції вищих порядків є засобом вираження концепцій. Розглянута фунарг-проблема й поняття лексичного замикання. Розглянуто метод часткового обчислення функцій декількох аргументів. Він може бути корисний при організації складних процесів обчислення. Розглянуті функціонали, що входять у стандарт common lisp.

ДОВІДКОВИЙ МАТЕРІАЛ

Функція є рекурсивною, якщо в її визначенні є виклик цієї функції.

Посилання функції на саму себе відбувається в деякому вираженні в тілі функції. Якщо це вираження визначає результат функції, то говорять про рекурсії за значенням. Якщо ж як результат функції повертається значення деякої іншої функції й рекурсивний виклик бере участь в обчисленні аргументів цієї функції, то говорять про рекурсії по аргументах.

Розглядають наступні способи організації рекурсії:
1 Проста рекурсія
2 Паралельна рекурсія
3 Взаємна рекурсія
4 Рекурсія високих порядків
Чиста функція - функція, що не залежить від контексту й не змінює його в процесі обчислення. Чисті функції мають властивість адитивного налагодження. Ця властивість полягає в тому, що складність налагодження системи чистих функцій при додаванні ще однієї чистої функції складається зі складності тестування системи й складності тестування функції.
Лексичне замикання - функція + контекст у момент визначення функції.

Метод програмування, у якому зовнішні до програми дані використовуються з метою керування роботою програми або самі інтерпретуються як програма, називається програмуванням, керованим даними.

Аргумент, значенням якого є функція, називають функціональним аргументом (functional argiment), а функцію, що має функціональний аргумент - функціоналом (functional).

Функція може повертатися функцією. Такі функції називають функціями з функціональним значенням (function valued). Метод програмування, у якому зовнішні до програми дані використаються з метою керування роботою програми або самі інтерпретуються як програма, називається програмуванням, керованим даними.

Аргумент, значенням якого є функція, називають функціональним аргументом (functional argiment), а функцію, що має функціональний аргумент - функціоналом (functional).

Функція може повертатися функцією. Такі функції називають функціями з функціональним значенням (function valued).

Функціонали, функції й функціонали з функціональним значенням називаються функціями вищого порядку.

При написанні й налагодженні функцій вищих порядків корисно вміти специфікувати тип цієї функції, що представляє області визначення й значення цієї функції.

Корисно ввести позначення:

Atom - атоми,
Number - число,
List (X) - NIL або списки з елементів типу X,
Bool - NIL або T,'
Some - будь-який об'єкт.
Лексичне замикання - функція + контекст у момент визначення функції. Замикання можна використати як функціональний аргумент.