Реферат на тему:

**Основи мови програмування Лісп**

**1. Об’єкти Ліспу**

Будь-яка структура даних є ***об’єктом***. Об’єкти можуть бути двох типів: ***прості*** та ***складені***. Прості об’єкти називаються ***атомами***. До атомів відносяться ***символи*** та ***числа***. Символ не може починатися з цифри. muLisp не розрізняє маленькі та великі літери, а перетворює всі введені літери у великі. Атом є неподільним, тобто його не можна розбити на компоненти. Атом, як і людина, має ***ім’я***. Іменами атомів є рядки символів. DOG, CAT, qw1232df, -32 є типовими іменами атомів. Символи T та NIL мають в Ліспі спеціальне призначення: вони позначають відповідно логічні значення істини та хибності. Ці символи завжди повинні мати одне фіксоване значення. Їх не можна використовувати в якості імен інших об’єктів Ліспу. Числа та логічні значення T та NIL є ***константами***, всі інші символи – ***змінними***.

Складними об’єктами даних є ***списки***. Список містить нуль (тоді говорять про порожній список) або більше об’єктів, кожний з яких може бути як простим, так і складеним. (FACE, LOOK, NOSE) є списком, який складається з трьох атомів. Порожній список позначається NIL = (), який є атомом. Список називається ***лінійним***, якщо його елементи є атомами. Інакше говорять про ***списки з підсписками***, наприклад: (7 (8 9) TR).

Для того щоб введений вираз не обчислювався, перед ним ставиться ***апостроф*** (‘). Якщо вираз вводиться без апострофа, то повертається його значення. При запуску програми muLisp значенням кожного атома вважається він сам. Значенням числа завжди є саме число, тому перед числами апостроф не ставиться. Тобто після старту системи при вводі Q результатом буде його значення – Q, а при вводі ‘Q — буде завжди Q. Апостроф перед виразом – це скорочення форми QUOTE, яка записується в наступній формі: ‘вираз = (QUOTE вираз). QUOTE можна використовувати як спеціальну функцію з одним аргументом, яка нічого з ним не робить, а повертає як результат сам аргумент.

Списки задаються переліком елементів, взятих в дужки, перед якими ставиться апостроф. Наприклад: ‘(ice, hen) або ‘((one 1) (two 2) (three 3)).

**2. Примітивні функції Ліспу**

***Виклик*** доівльної функції у Ліспі має наступний формат:

(*name arg1 arg2* ...), де *name* — ім’я функції, *arg1*,*arg2*, ... — її аргументи.

Мова програмування Lisp має п’ять ***примітивних функцій***.

1. (**CAR** <*list*>) — знаходження голови списку.

2. (**CDR** <*list*>) — знаходження хвосту списку.

3. (**CONS** <*object*> <*list*>) — об’єднання (конкатенація) об’єкта зі

списком.

4. (**EQL** <*atom1*> <*atom2*>) — порівняння двох атомів.

5. (**ATOM** <*object*>) — перевірка, чи є об’єкт <*object*> атомом.

CAR та CDR називаються ***селекторними*** функціями, оскільки вони дають можливість вибирати або знищувати частину об’єкта. Результатом функції (CAR list) завжди є перший елемент списку list, якщо він непорожній і NIL в іншому випадку. Результатом функції (CDR list) є список list без першого елемента, якщо list містить більш одного елемента і NIL в іншому випадку.

$ (CAR ‘(q w e r t y)) $ (CDR ‘(q w e r t y)) $ (CAR ‘((one 1) (two 2)))

q (w e r t y) (one 1)

$ (CAR ‘()) $ (CDR ‘(tree)) $ (CDR ‘((q w)) $ (CDR ‘())

NIL NIL NIL NIL

За допомогою функцій CAR, CDR можна знаходити за даним списком будь-який його підсписок або атом. Дозволяється використовувати функції, які є комбінаціями CAR та CDR. Імена таких функцій починаються на C і закінчуються на R, а між ними знаходиться послідовність літер A та D (але не більше 4 літер у реалізації інтерпретатора muLisp), яка вказує шлях обчислення.

$ (CAR (CDR (CDR ‘(q w e r t y))))

$ (CADDR ‘(q w e r t y))

e

$ (CAR(CDR (CDR ‘((q 1) (w 2) (e 3)))))

$ (CADDR ‘((q 1) (w 2) (e 3)))

(e 3)

$ (CDR (CDR ‘((q 1) (w 2) (e 3)))) $ (CAR (CAR ‘((q w))))

$ (CDDR ‘((q 1) (w 2) (e 3))) $ (CAAR ‘((q w)))

((e 3)) q

Функція ***конструктора*** CONS використовується для додання об’єкту до заданого списку. Об’єкт який додається, стає головою списку. Якщо другий аргумент не задано, то він вважається рівним NIL.

$ (CONS ‘(q w) ‘(r (t y))) $ (CONS apple ‘(q w))

((q w) r (t y)) (apple q w)

$ (CONS ‘(q w) ‘(r t y)) $ (CONS 5)

((q w) r t y) (5)

Зазначимо, що якщо результатом обчислення виразу (CONS *object* *list*) буде *new*, то результатом (CAR *new*) буде *object*, а результатом (CDR *new*) буде *list*, тобто

(CAR (CONS *object* *list*)) = *object*,

(CDR (CONS *object* *list*>)) = *list*.

$ (CAR (CONS ‘(q w) ‘(r (t y)))) $ (CAR (CONS apple NIL))

(q w) apple

Функцією ***порівняння*** є EQL. Вона порівнює значення першого та другого аргумента, які обов’язково повинні бути атомами, та повертає значення істини (Т) або хибності (NIL).

$ (EQL ‘qw ‘qw) $ (EQL (CAR ‘(q w)) q)

T T

$ (EQL (CAR ‘(q, w) NIL) $ (EQL nil ‘(nil))

NIL NIL

При написанні програм на Ліспі часто виникає запитання: чи є даний об’єкт атомом? Це питання вирішує предикат ATOM. Він повертає Т, якщо об’єкт є атомом і NIL в іншому випадку. Порожній список NIL є атомом.

$ (ATOM qwerty) $ (ATOM ‘(q w e)) $ (ATOM ‘())

T NIL T

$ (ATOM ‘(q)) $ (ATOM 3) $ (ATOM ‘(NIL))

F T NIL

**3. Функції призначення**

Функції ***призначення*** застосовуються для надання значень програмним змінним. До них відносяться:

1. (**SET** symbol object)

— заміна символа об’єктом

2. (**SETQ** sym1 form1 sym2 form2 ... )

— спеціальна форма функції SET

3. (**PSETQ** sym1 form1 sym2 form2 ... )

— спеціальна форма функції SET

4. (**POP** symbol)

— повертає вершину стека (списку)

5. (**PUSH** symbol form)

— кладе символ symbol в стек (список) form.

Операція ***заміни*** значення символа здійснюється за допомогою функції SET. Вона присвоює символу symbol значення object, або зв’язує symbol з object. Для скорочення замість SET ‘ пишуть SETQ (SET Quote). Як результат функція присвоєння повертає другий аргумент.

$ (SET ‘fox ‘(a s d)) $ (SETQ vowels ‘(a e i o u)))

$ (SETQ fox ‘(a s d)) $ (SETQ vowels (CONS ‘y vowels))

(a s d) (y a e i o u)

Функція SETQ дозволяє здійснювати заміну значень декільком символам в одній команді: (SETQ a 1 b 2 c 3). При цьому зміни виконуються послідовно зліва направо. Після цього значенням символу a стане 1, b-2,c-3.

Функція PSETQ ідентична до функції SETQ за винятком того, що всі форми оцінюються до того, як будуть здійснені будь-які заміни. Проілюструємо це на прикладі. Значення символа Sym позначатимемо через Val(Sym).

$ (SETQ w 1 e 2) Val(w)=1, Val(e)=2 $ (SETQ w 1 e 2) Val(w)=1, Val(e)=2

$ (SETQ w e e w) Val(w)=2, Val(e)=2 $ (PSETQ w e e w)Val(w)=2, Val(e)=1

При виконанні операції заміни необхідно розрізняти символ та значення. При старті системи mulLsp значенням кожного символа є він сам. Якщо ми введемо DOG, то і результатом буде DOG. Присвоїмо символові DOG значення CAT: (SET ‘DOG ‘CAT). Результатом виразу (SET DOG ‘HEN) буде HEN, але значення HEN ми присвоювали не символу DOG, а значенню символа DOG, тобто символу CAT. Значення символа DOG залишилося без зміни. Розглянемо результат наступних дій:

(SET ‘car ‘road) Val(car) = road Val(road) = road

(SET car flower) Val(car) = road Val(road) = flower Val(flower) = flower

(SET ‘car car) Val(car) = road Val(road) = flower Val(flower) = flower

(SET road car) Val(car) = road Val(road) = flower Val(flower) = road

(SET ‘road 4) Val(car) = road Val(road) = 4 Val(flower) = road

(SET road ‘hen) помилка, 4 не є символом і не може приймати інші значення

POP повертає голову списка (вершину стека) і замінює значення <symbol> на його хвіст. PUSH кладе <symbol> в стек та змінює його значення на збільшений стек.

$ (SETQ a ‘(q w e r t)) Val(a) = (q w e r t)

$ (POP a) Val(a) = (w e r t)

$ (PUSH ‘n a) Val(a) = (n w e r t)

**4. Визначення функцій в Ліспі**

Поряд з примітивними функціями можуть існувати функції, визначені користувачем. Функція викликається набором аргументів і повертає єдине значення. Визначення функції в Ліспі має наступний вигляд:

(DEFUN name (arg1 arg2 ...)

task1

task 2

. . . . . )

де name — ім’я функції, arg1, arg2, ... — аргументи (параметри). Тіло функції містить послідовність задач. Ключове слово DEFUN виникло з DEfine FUNction.

$ (DEFUN FIRST (lst) $ (FIRST ‘(q w e r t y))

 (CAR lst) ) q

$ (DEFUN THIRD (lst) $ (THIRD ‘(q w e r t y))

 (CADDR lst) ) e

Визначимо функцію **NULL**, яка розпізнає порожній список. Вона повертає істину, якщо її аргументом є порожній список і хибність в іншому випадку.

$ (DEFUN NULL (obj) $ (NULL ‘(q w e r)) $ (NULL (CDR ‘(r)))

 (EQL obj NIL) ) F T

Нам вже відомі три функції розпізнання: EQL, ATOM, NULL. Функції, які застосовуються для перевірки певних умов та можуть повертати лише два значення – істини чи хиби, називаються ***предикатами***.

Тіло функції складається з послідовності завдань. Завдання можуть бути двох типів: прості та умовними. Будь-яке завдання береться в круглі дужки і може розглядатися як список виразів, які треба проінтерпретувати.

Якщо завдання є атомом або його перший елемент є атомом, то таке завдання називається ***простим***. Наприклад, (CONS ‘NR LST).

Якщо перший елемент списка, який описує завдання не є атомом, то таке завдання називається ***умовним***. Наприклад, ((ATOM lst) (CONS expr lst)).

В умовному завданні перший елемент списку обов’язково є предикатом. Якщо значення предикату NIL, то значення завдання стає рівним NIL і Лісп переходить до виконання наступного завдання. Якщо предикат повертає не NIL, відбувається виконання хвосту списку завдання, а інші завдання ігноруються. Якщо предикат повертає Т, а хвіст завдання порожній, то результатом всієї функції буде T.

Напишемо предикат, який розпізнає списки. Якщо аргументом є список, то предикат повертає істину, інакше — хибність. Функцію **LISTP** можна проінтерпретувати наступним чином: “Якщо obj є атомом, то повернути NIL, інакше повернути T”.

$ (DEFUN LISTP (obj) $ (DEFUN LISTP (obj)

 ((ATOM obj) NIL) ((NULL obj))

 T ) ((ATOM obj) NIL)

 T )

В другій колонці написано предикат LISTP, який розпізнає додатково порожній список (повертає істину). Перше завдання є умовним, хвіст якого порожній. Його можна проінтерпретувати так: перевірити об’єкт obj на порожній список, і якщо він є таким, передати як результат функції істину. Немає потреби писати: ((NULL obj) T), оскільки це те ж саме, що і ((NULL obj)). Останнім завданням цих предикатів є атом Т. Це означає, що якщо жодне з умовних завдань не виконане (лише за цієї умови керування програмою дійде до останнього завдання), то як результат функції повернути Т. Для другого визначення функції LISTP маємо:

$ (LISTP ‘tree) $ (LISTP ‘()) $ (LISTP ‘(q w e r t y))

NIL T T

Для кращого розуміння роботи тіла функції та простих і умовних завдань розглянемо функцію sm та результати, які вона буде генерувати при певних вхідних значеннях:

$ (DEFUN sm (lst) $ (sm ‘()) $ (sm ‘(q w e))

 ((NULL lst) 10 1) 1 12

 (SETQ b 2)

 ((CDR lst) 12) $ (sm ‘(i)) $ (sm ‘g)

 (SETQ b 3) ) 3 3

Як бачимо, після виконання простого завдання керування завжди передається наступному завданню (якщо таке є). Якщо предикат умовного завдання істинний, то виконується його хвіст і повертається результат останнього виразу хвоста.

Вмонтована функція (**LIST** x1 ... xn) утворює та видає список, елементами якого є x1, ..., xn. Якщо аргументи не задані, результатом буде NIL.

$ (LIST ‘a ‘b ‘c ‘d) $ (LIST ‘a ‘(b c) ‘d) $ (LIST)

(a b c d) (a (b c) d) NIL

Напишемо функцію **MEMBER**, яка має два аргументи: nam - символ та lst - список і яка повинна перевірити чи належить символ списку. Інтуїтивно необхідно порівняти символ з першим елементом списку, потім з другим елементом і так далі. Проблема в такому розв’язку виникає в тому, що ми не знаємо наперед довжини списку. А якщо ми і знаємо цю довжину, і якщо вона велика, то тіло функції буде дуже великим. Така функція буде мати приблизно такий вигляд (перший стовпчик):

$ (DEFUN MEMBER (nam lst) $ (DEFUUN MEMBER (nam lst)

 ((EQL nam (FIRST lst))) ((NULL lst) NIL)

 ((EQL nam (SECOND lst))) ((EQL nam (CAR lst)) T)

 ((EQL nam (THIRD lst))) (MEMBER nam (CDR lst)) )

 ((EQL nam (THIRD (CAR lst))))

 . . . . . . . . . . . . . . .

Змінимо наш підхід до побудови функції. В другому стовпчику побудовано функцію MEMBER, в основі якої лежить *рекурсивний* підхід, який базується на наступних фактах:

1. Якщо список порожній (не має елементів), то nam не належить списку.

2. Якщо nam дорівнює голові списку, то nam належить списку.

3. Якщо nam не дорівнює голові списку, то nam може належити списку тоді і тільки тоді коли nam належить хвосту списку.

Розглянемо дві рекурсивні функції: **REMBER** (REMove memBER), яка має два аргументи — атом obj та список lst і яка видаляє перше зустрічання атома obj в списку lst. REMBER-ALL яка видаляє всі атоми obj в списку lst.

$ (DEFUN REMBER (obj lst) (DEFUN REMBER-ALL (obj lst)

((NULL lst) NIL) ((NULL lst) NIL)

((EQL obj (CAR lst)) (CDR lst)) ((EQL obj (CAR lst))

(CONS (CAR lst) (REMBER-ALL obj (CDR lst))

(REMBER obj (CDR lst))) ) (CONS (CAR lst)

 (REMBER-ALL obj (CDR lst))))

Результат роботи цих функцій проілюструємо на прикладах:

$ (REMBER ‘a ‘(q a w e r t a y)) $ (REMBER-ALL ‘a ‘(q a w e r t a y))

(q w e r t a y) (q w e r t y)

Примітивна функція EQL використовується для порівняння атомів. Часто виникає потреба порівнювати списки. Напишемо функцію **EQLIST**, яка порівнює списки. Її побудуємо на основі наступних фактів:

1. Якщо перший список порожній, то, якщо і другий список порожній, повернути Т, інакше повернути NIL (або просто повернути (NULL другого списку)).

2. Якщо другий список порожній, повернути NIL.

3. Якщо голова першого списку не дорівнює голові другого списку, повернути NIL.

4. Перевірити рівність хвостів першого та другого списків.

$ (DEFUN EQLIST (lst1 lst2) $ (DEFUN NOT (obj)

 ((NULL lst1) (NULL lst2)) (EQL obj NIL) )

 ((NULL lst2) NIL)

 ((NOT (EQL (CAR lst1) (CAR lst2))) NIL)

 (EQLIST (CDR lst1) (CDR lst2)) )

Функція NOT повертає NIL, якщо список не порожній і Т інакше.

Розглянемо задачу об’єднання списків. Роботу функції **APPEND**, аргументами якої є два списки lst1 та lst2, можна описати наступним чином:

1. Якщо lst1 порожній, повернути lst2.

2. З’єднати голову першого списку зі списком, який отримано в результаті об’єднання хвоста першого списку з другим списком.

$ (DEFUN APPEND (lst1 lst2)

 ((NULL lst1) lst2)

 (CONS (CAR lst1) (APPEND (CDR lst1) lst2)) )

Функція (**REVERSE** lst1) обертає список lst1. Якщо вихідний список порожній, то і результатом буде порожній список. Інакше необхідно об’єднати обернений хвіст вихідного списку з його першим елементом. Оскільки на вхід другого аргумента функції APPEND повинен подаватися список, необхідно з першого елемента списку зробити список, який складається лише з нього. Це виконує команда (CONS (CAR lst) NIL).

$ (DEFUN REVERSE (lst)

 ((NULL lst) NIL)

 (APPEND (REVERSE (CDR lst)) (CONS (CAR lst) NIL)) )

Напишемо функцію REVERSE без використання функції APPEND. Для цього побудуємо функцію REVERSE з двома аргументами на принципі обробки стеку. Вихідний список — стек символів. Якщо він порожній, то і результуючий стек буде порожнім. Інакше взяти символ з вершини стеку і покласти його на другий стек. Другий стек при виклику повинен бути NIL: (REVER <list> NIL).

 $ (DEFUN REVER (lst1 lst2) $ (REVER ‘(q w e) NIL)

 ((NULL lst1) lst2) (e w q)

 (REVER (CDR lst1) (CONS (CAR lst1) lst2)) )

**5. Середовище системи muLisp**

Середовище muLisp або поточний стан системи складається з усіх активних на даний момент структур даних, значень змінних та визначених функцій. Команда SAVE зберігає поточне середовище muLisp у вигляді SYS - файлу. Команда (SAVE ’C:HOME) зберігає середовище в файл HOME.SYS на диску C. Після успішного виконання команди запису повертається Т, інакше — NIL.

Середовище muLisp може бути завантажене за допомогою команди LOAD: (LOAD <file>). Якщо файл не знайдено, повертається NIL, інакше жодне значення не повертається, а mulisp починає працювати з новим середовищем.

Для завантаження SYS-файлів безпосередньо після запуску muLISP може використовуватися команда операційного середовища (ОС). Наприклад, команда ОС

> muLISP C:HOME

завантажує SYS-файл HOME.SYS із пристрою C після запуску muLISP із пристрою, взятого по замовченню. Відмітимо, що тип SYS-файла у команді не вказується. Якщо SYS-файл не знайдено при завантаженні з використанням команди ОС, на екран дисплею видається повідомлення: File not found.

Після завантаження SYS-файла кількість пам’яті, призначеної для кожної області даних, коректується у відповідності до поточного об’єму пам’яті. Це означає, що поточний об’єм пам’яті не обов’язково повинен бути точно таким, як і при створенні SYS-файлів. Але якщо пам’яті для розташування середовища SYS-файлів недостатньо, то виникає помилка типу "Недостатньо пам’яті, переривання", і muLISP буде завершено.

**6. Трасировка функцій в muLisp.**

Мова програмування muLisp для трасировки використовує програму debug.lsp, яка завантажується в середовище Ліспу. Для того, щоб дозволити трасировку функції <func>, необхідно дати команду (**TRACE-FUNCTION** <func>). Якщо після цього викликати функцію func з параметрами, то на екрані відобразиться шлях виконання функції. На кожному кроці буде виводитися ім’я функції та список фактичних параметрів. Після виконання функції на екран виводиться значення функції. Команда (**UNTRACE-FUNCTION** <func>) забороняє трасировку функції <func>. Якщо в тілі функції <func> існує виклик інших функцій, і ми хочемо побачити їх трасировку, необхідно дозволити їх трасировку. Вираз <func>=<value> в трасі означає те, що функція <func> повертає значення <value>.

Якщо вивід траси відбувається дуже швидко, для тимчасової зупинки траси можна використати <CTRL-S>.

Наприклад, розглянемо трасування функціі APPEND (злиття двох списків), яка була визначена раніше. Після виконання команд

$ (TRACE-FUNCTION ‘APPEND)

$ (APPEND ‘(q w e) (r t y u))

на екрані відобразиться траса:

APPEND [(Q W E), (R T Y U)]

 APPEND [(W E), (R T Y U)]

 APPEND [(E), (R T Y U)]

 APPEND [NIL, (R T Y U)]

 APPEND = (R T Y U)

 APPEND = (E R T Y U)

 APPEND = (W E R T Y U)

APPEND = (Q W E R T Y U)

(Q W E R T Y U)

Виведемо на екран трасу функції REVERSE з дозволом (лівий стовпчик) та без дозволу (правий стовпчик) трасировки функції APPEND для виразу (REVERSE ‘(q w)).

$ (TRACE-FUNCTION ‘REVERSE) $ (TRACE-FUNCTION ‘REVERSE)

$ (TRACE-FUNCTION ‘APPEND) $ (REVERSE ‘(q w))

$ (REVERSE ‘(q w)) REVERSE [(Q W)]

REVERSE [(Q W)] REVERSE [(W)]

 REVERSE [(W)] REVERSE [NIL]

 REVERSE [NIL] REVERSE = NIL

 REVERSE = NIL REVERSE = (W)

 APPEND [NIL, (W)] REVERSE = (W Q)

 APPEND = (W) (W Q)

 REVERSE = (W)

 APPEND [(W), (Q)]

 APPEND [NIL, (Q)]

 APPEND = (Q)

 APPEND = (W Q)

REVERSE = (W Q)

(W Q)