Функції виведення в потік

Функції виводу передають результат в поточний потік виводу.

1. **(PRIN1 <obj>).** Передає символьне представлення об'єкту в потік і повертає об'єкт. Друк відбувається згідно з поточною системою числення. Змінна *PRINT-ESCAPE* набуває значення T.

2. **(PRINC <obj>).** Працює як і PRIN1, але імена виводяться без контрольних символів. Значення контрольної змінної *PRINT-ESCAPE* при виклику PRINC стає рівним NIL.

3. **(PRINT <obj>).** Для виводу виразів можна використовувати функцію PRINT. Вона має один аргумент. При виклику цей аргумент обчислюється, а потім виводиться його значення. Перед виводом аргумента відбувається перехід на новий рядок, а після виводу аргумента друкується проміжок. Значенням функції є значення аргумента.

4. **(TERPRI).** Виводить у потік ознаку закінчення рядка. Сама функція повертає NIL.

5. **(FRESH-LINE).** Якщо ми знаходимося на початку рядка, функція просто повертає NIL. Інакше вона передає в потік ознаку закінчення рядка і повертає T.

6. **(WRITE-STRING <рядок>), (WRITE-LINE <рядок>).** В потік виводить заданий рядок. Функція WRITE-LINE після виводу рядка автоматично виконує перехід на новий рядок командою (TERPRI).

Використання описаних функцій проілюструємо в наступному параграфі.

Породження комбінаторних об'єктів.

Розглянемо задачі, в яких необхідно отримати всі елементи деякої множини.

1. Надрукувати всі послідовності довжини *n* з цифр 0, 1. При виклику функції перший аргумент *n*, аргумент *lst* – допоміжний, треба задавати NIL.

```
(DEFUN P01 (n lst)
 (COND
  ((ZEROP n) (PRIN1 lst) (TERPRI))
  ((P01 (- n 1) (CONS 0 lst)))
  ((P01 (- n 1) (CONS 1 lst)))
))
```

2. Надрукувати всі підмножини множини {1..n}.

Оскільки всі підмножини будь-якої множини {1..n} перебувають у взаємно однозначній відповідності зі всіма послідовностями з 0 та 1 довжини *n*, то ця задача зводиться до попередньої. Тільки замість виведення списку з 0 та 1 необхідно виводити номери всіх елементів списку, які дорівнюють 1. Функція (PRNNOM lst) виводить необхідні номери елементів.

```
(DEFUN PRNNOM (lst)
 (SETQ i 0)
 (LOOP WHILE (CAR lst)
   DO (SETQ i (+ 1 i))
   (IF (= 1 (POP lst)) (PROGN (PRIN1 i) (PRINC " "))) )  )
```

```
(DEFUN SUBSETS (n lst)
 (COND
  ((ZEROP n) (PRNNOM lst) (TERPRI))
  ((SUBSETS (- n 1) (CONS 0 lst)))
  ((SUBSETS (- n 1) (CONS 1 lst)))
))
```

Отримані розв'язки можна узагальнити на випадок довільної множини, заданої списком. Як і раніше, використаємо генератор послідовностей 0 та 1, удосконалимо функцію виведення множини (списку) на друк – тепер не вдасться обчислити елемент множини, його треба вибрати з відповідного списку.

```
;надрукувати всі підмножини заданої множини

;задана множина
(setq list '(cat dog cow fox))

;розпізнавач одиниці
(defun onep (x) (eql 1 x))

;друк тих елементів списку list, яким відповідають 1 в списку mask
(defun listprn (list mask)
 (loop while (car list)
  do (setq item (pop list))
     (if (onep (pop mask)) (progn (prin1 item) (princ " ")))
))

;генерування всеможливих послідовностей 1 0
(DEFUN P011 (n lst setlist)
 (COND
  ((ZEROP n) (listprn setlist lst) (TERPRI))
  ((P011 (- n 1) (CONS 0 lst) setlist))
  ((P011 (- n 1) (CONS 1 lst) setlist))
))

;парадна функція
(defun allsubsets (setlist)
 (p011 (length setlist) nil setlist))

;друк розв'язку і перевірка, чи змінився сам список
(allsubsets list)

(print list)
```

3. Надрукувати всі послідовності довжини *k* з чисел 1..*n*.

Друкуватимемо послідовності у лексикографічному порядку. За допомогою функції (GEN1 n) згенеруємо список з n елементів, кожен з яких дорівнює 1. Список lst зберігатиме поточну перестановку. Функція (NEXT lst n) знаходить перестановку, яка буде наступною після lst. Функція P-BEST є найкращим рекурсивним розв'язком цієї задачі.

```
(DEFUN GEN1 n)
((ZEROP n) NIL)
(CONS 1 (GEN1 (- n 1)))  )
```

```
(DEFUN NEXT (lst n)
((< (CAR lst) n) (CONS (+ (CAR lst) 1) (CDR lst)))
((NULL (CDR lst)) NIL)
(CONS 1 (NEXT (CDR lst) n))
```

Шукана функція має вигляд:

```
(DEFUN P12 k n)
(SETQ lst (GEN1 k))
(LOOP
   ((< (LENGTH lst) k))
   (PRIN1 lst) (TERPRI)
   (SETQ lst (NEXT lst n))
)  )
```

```
(DEFUN P12BEST (n k lst c)
((ZEROP n) (PRIN1 lst) (TERPRI))
(PUSH 1 c)
(LOOP WHILE
 (> (CAR c) k) DO
(P12BEST (- n 1) k (CONS (CAR c) lst) c)
(SETQ c (CONS (+ 1 (CAR c)) (CDR c)))
) (POP c)  )
```

Обчислювані функції

Обчислення виразів та виклики функцій інтерпретатор Lisp виконує автоматично. Обчислювані функції необхідні в тих випадках коли необхідно безпосередньо обчислити вираз або звернутися до функцій. Визначенням функції є список, який складається з трьох частин: імені типу функції, формальних параметрів та тіла функції.

CAR-елементом визначення функції є ім'я типу фукції – LAMBDA, NLAMBDA чи MACRO. Тип функції дає інтерпретаторові інформацію про те, як використовувати дану функцію.

Визначення функцій та їх обчислення в Ліспі основано на ***лямбда-численні*** Чорча. Лямбда вираз, який взято з лямбда числення, є важливим механізмом у програмуванні. В лямбда численні Чорча функція записується у вигляді:

$$\text{lambda } (x_1, x_2, ..., x_n) \text{ . } f$$

У Лісп лямбда вираз має вигляд:

```
(LAMBDA (x1 x2 ... xn) f)
```

Символ LAMBDA говорить нам про визначення функції. Символи *xi* – це формальні параметри, *f* – тіло функції. Тілом функції може бути довільна форма, значення якої може обчислити інтерпретатор Ліспа. Функцію, яка обчислює суму квадратів двох чисел, можна визначити так:

```
(LAMBDA (x y) (+ (* x x) (* y y)) )
```

Формальність параметрів вказує на те, що ми можемо замінити їх на інші символи, але від цього не зміниться сутність обчислення функції.

Лямбда вираз – це визначення обчислення та параметрів функції в чистому вигляді без фактичних параметрів або аргументів. Для застосування такої функції до певних аргументів, необхідно поставити лямбда вираз на місце імені функції:

(лямбда-вираз a_1 a_2 ... a_n)

Тут a_i – форми, що задають фактичні параметри. Наприклад, множення (* 3 4) можна записати з використанням лямбда виклику:

```
$ ((LAMBDA (x y) (* x y)) 3 4)
12
```

Наступний виклик будує список з двох аргументів:

```
$ ((LAMBDA (x y) (CONS x (CONS y NIL))) 'dog 'cat)
(dog cat)
```

Таку форму виклику називають ***лямбда викликом***. Обчислення лямбда виклику відбувається в два етапи. Спочатку обчислюються значення фактичних параметрів та відповідні формальні параметри зв'язуються з отриманими значеннями. На другому етапі обчислюється форма, яка є тілом лямбда виразу. Отримане значення повертається в якості значення лямбда виклику. По завершенню обчислення формальним параметрам повертаються зв'язки, які існували до лямбда виклику. Весь цей процес називається ***лямбда перетворенням***.

Пам'ятайте, що лямбда вираз без фактичних параметрів є лише визначення, а не форма, яку можна обчислити. Сам по собі лямбда вираз інтерпретатором не сприймається. Якщо ви введете: (LAMBDA (x y) (CONS x (CONS y NIL))), то інтерпретатор Ліспу видасть повідомлення про помилку.

Лямбда вираз є як чисто абстрактним механізмом для визначення та опису обчислення, так і механізмом для зв'язування формальних та фактичних параметрів під час виконання обчислення. Лямбда вираз є функцією без імені.

Ми вже говорили про те, як визначити нову функцію – це можна здійснити за допомогою функції DEFUN. Визначення функції викликається так:

(DEFUN <і'мя> <лямбда вираз>)

Для спрощення опустимо зовнішні дужки у лямбда виразі та сам атом LAMBDA. Тоді ми отримаємо знайоме нам визначення функції. Наступні визначення еквівалентні:

(DEFUN list2 (LAMBDA (x y) (CONS x (CONS y NIL))))

та

(DEFUN list2 (x y) (CONS x (CONS y NIL)))

Функція DEFUN з'єднує символ з лямбда виразом, після чого символ починає іменувати обчислення, яке визначається лямбда виразом. Значенням функції DEFUN є ім'я нової функції.

За допомогою структури **LET**, яка визначена в common.lsp, можна утворити локальний звя'зок. Значення змінним форми LET присвоюються одночасно. Ця структура має наступний вигляд:

(LET ((m1 a1) (m2 a2) ... (mN aN)) <форма1> <форма2> ... <формаN>),

яка в дійсності є лямбда викликом, де формальні та фактичні параметри знаходяться разом на початку структури:

((LAMBDA (m1 m2 ... mN) <форма1> <форма2> ... <формаN>) a1 a2 ... aN)

Наступні виклики еквівалентні:

```
$ (LET ((x 4)(y 2))(+ x y))          $ ((LAMBDA (x y) (+ x y)) 4 2)
6                                    6
```

Функція типу **MACRO** називається макро-функцією. Макроси є потужним робочим інструментом програмування. Синтаксис визначення макроса виглядає таким же чином як синтаксис визначення функції форми DEFUN:

(**DEFMACRO** <ім'я> <лямбда список> <тіло>)

Виклик макроса співпадає за формою з викликом функції, але його обчислення відрізняється від обчислення виклику функції. В макросі не обчислюються аргументи. Обчислення виклику макроса складається з двох послідовних етапів. Спочатку відбувається

обчислення тіла з аргументами (як і для функції). Цей етап називається ***розширенням*** або розкриттям макроса. На другому етапі обчислюється розкрита форма, значення якої повертається в якості значення всього макровиклику. Визначимо макрос PUSH1, який працює як відома нам функція PUSH (в дійсності PUSH є вмонтованим в середовище Лісп макросом).

```
$ (DEFMACRO PUSH1 (x y)                    $ (SETQ ‘a ‘(1 2 3)) (PUSH1 6 ‘a)
(LIST 'SETQ y (LIST 'CONS x y)))           a = (6 1 2 3)
```

При програмуванні розкритого макроса явно не видно, тому для їх тестування існує спеціальна функція – **MACROEXPAND**, яка здійснює тільки розкриття макроса. Повертається макророзширення виклику, яке можна вивчати.

```
$ (MACROEXPAND '(PUSH1 6 s))
(SETQ S (CONS 6 S))
```

1. **FUNCALL** (<функція> <arg1> <arg2> ... <argN>)

Виконує <функцію> над аргументами та повертається результат. <Функція> повинна бути або іменем обчислюваної функції, або тілом LAMBDA. Якщо <функція> – це ім’я макро або невизначеної функції, виникає переривання по помилці "невизначена функція".

```
$ (FUNCALL 'CONS 'a '(b c d))              $ (FUNCALL '(LAMBDA (n) (* n n)) 5)
(a b c d)                                  25
```

2. **EVAL** <об’єкт>

Інтерпретатор Лісп називається EVAL, його можна як і інші функції викликати з програми. У звичайній роботі інтерпретатор викликати не має потреби, оскільки його виклик має місце неявно. Зайвий виклик інтерпретатора може зняти ефект блокування (QUOTE), або дозволить знайти значення виразу. EVAL – це універсальна функція Лісп, яка може обчислити довільний правильно побудований лісп-вираз.

```
$ (SETQ a 'b b 'c)     $ (EVAL '(CONS 'A '(B C)))      $ (EVAL '(PRIN1 '(a b c)))
$ (EVAL a)             (A B C)                         (a b c) (a b c)
c

$ (DEFUN a (n)         $ (EVAL '(a 3))     $ (EVAL (LAMBDA (n) (* n n)) 7)
(+ n 2)  )             5                   49
```

Діалог з інтерпретатором Ліспа на верхньому (командному) рівні можна описати простим циклом:

(SETQ e (READ)) введення виразу
(SETQ v (EVAL e)) обчислення виразу
(PRINT v) друк результата

3. **CONSTANTP** <об’єкт>

Об’єкт є константою тоді і тільки тоді, коли (EVAL <об’єкт>) повертає <об’єкт>. Символ NIL, числа та списки, в яких CAR-елемент є символ QUOTE, в Lisp є константами. Якщо <об’єкт> — константа, функція CONSTANTP повертає T, інакше — NIL.

```
$ (CONSTANTP ())       $ (CONSTANTP 23.543)
T                       T
$ (CONSTANTP 'v)       $ (CONSTANTP '(a b c))
NIL                    NIL
$ (CONSTANTP '(QUOTE (a b c)))
T
```

4. **APPLY** <функція> <арг-список>

Застосовує функцію до списку аргументів. (APPLY f (x1 x2 ... xN)) еквівалентно (f x1 x2 ... xN). Використання функції APPLY є більш гнучким у порівнянні з прямим викликом функції. Діє як і функція FUNCALL, тільки аргументи функції приймає не окремо, а списком.

Якщо функція – ім'я визначеної користувачем функції або тіло LAMBDA, APPLY пов'язує формальні аргументи функції з фактичними аргументами, обчислює тіло функції, відтворює вихідні значення формальних аргументів і повертає значення обчислення тіла функції.

Якщо функція – не ім'я функції і не тіло LAMBDA, APPLY генерує переривання по помилці "невизначена функція".

```
$ (APPLY 'CONS '(a (b c d)))
(a b c d)
```

```
$ (SETQ z '(LAMBDA (n) (* n n)))
$ (APPLY z '(4))
16
```

Завдання

1. Надрукувати всі послідовності з k натуральних чисел, у яких i-ий член не перевищує i.
2. Надрукувати всі розстановки дужок в добутку множників. Порядок множників не змінюється, дужки однозначно визначають порядок дій. Наприклад, n=4: ((a b) c) d (a (b c)) d a ((b c) d) (a b)(c d) a (b (c d)).
3. Написати програму:

а) (INTEGRATE f a b n) – інтегрування функції f від a до b; відрізок [a; b] розбити на n частин.

б) (HALFDIV f a b e) – знайти корінь рівняння f(x)=0, який лежить в інтервалі [a; b] методом половинного ділення з похибкою e.

в) (SQRT2 n e) – обчислити квадратний корінь методом ітерації з похибкою e.

г) (DIFF f x) – обчислити значеняя похідної функції f в точці x.

Завдання 3. а) – хороша нагода продемонструвати застосування описаних вище інструментів. Якщо використати для інтегрування формулу лівих прямокутників, то функція матиме такий вигляд.

```
(defun integrate (f a b n)
  (setq s 0 i 0 h (/ (- b a) n))
  (loop while (< i n)
    do (setq s (+ s (funcall f (+ a (* i h)))) i (+ 1 i))
  ) (* s h)
)
```

А її використання, у тому числі з лямбда, такий:

```
$ (integrate 'sin 0 pi 100)
1.9998355038874436D0

$ (integrate (lambda (x) (* x x)) 0.0 1.0 500)
0.33233404
```