Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

- Доц. Баклан І.В.
- Email: iaa@ukr.net
- Web: baklaniv.at.ua

Лекція 3

Мова штучного інтелекту

LISP (продовження)

У Ліспі функції - це звичайнісінькі об'єкти, такі ж як символи, рядки або списки. Даючи функції ім'я за допомогою **function**, ми отримуємо асоційований об'єкт. Як і **quote**, **function** - це спеціальний оператор, і тому нам не потрібно брати в лапки його аргумент:

```
[1]> (function +)
#<SYSTEM-FUNCTION +>
```

Таким дивним чином відображаються функції в типовій реалізації Common Lisp.
До сих пір ми мали справу тільки з такими об'єктами, які при друку відображаються так само, як ми їх ввели. Ця угода не поширюється на функції. Вбудована функція **+** зазвичай є шматком машинного коду. У кожній реалізації Common Lisp може бути свій спосіб відображення функцій.

Як і будь-який інший об'єкт, функція може служити аргументом. При мером функції, аргументом якої є функція, є **apply**. Вона вимагає функцію і список її аргументів і повертає результат виклику цієї функції з заданими аргументами:

```
[13]>  (apply `+ `(1 2 3))
6
[14]>
```

Apply приймає будь-яку кількість аргументів, але останній з них обов'язково повинен бути списком:

```
[14]>  (apply `+ 1 2 5 `(1 2 3))
14
[15]>
```

Функція **funcall** робить те ж саме, але не вимагає, щоб аргументи були упаковані в список:

```
[17]> (funcall `+ 1 2 3)
6
```

Зазвичай створення функції і визначення її імені здійснюється за допомогою макросу **defun**. Але функція не обов'язково повинна мати ім'я, і для її визначення ми не зобов'язані використовувати **defun**. Як і більшість інших об'єктів Лиспа, ми можемо задавати функції буквально.
Щоб буквально послатися на число, ми використовуємо послідовність цифр. Щоб таким же чином послатися на функцію, ми використовуємо лямбда-вираз. Лямбда-вираз - це список, який містить символ **lambda** і наступні за ним список аргументів і тіло, що складається з 0 або більше виразів. Нижче наведено лямбда-вираз, що представляє функцію, яка складає два числа і повертає їх суму:

```
(lambda (x y)
(+ x y))
```

Список **`(x y)`** містить параметри, за ним слід тіло функції.

Лямбда-вираз можна вважати ім'ям функції. Як і звичайне ім'я функції, лямбда-вираз може бути першим елементом виклику функції:

```
[18]>  ((lambda (x) (+ x 100)) 1)
101
```

```
[23]>  (funcall (lambda (x) (+ x 100)) 1)
101
```

Крім іншого, такий запис дозволяє використовувати функції, не привласнюючи їм імена.

Що ж таке *Лямбда*?

В лямбда-виразі **`lambda`** не є оператором. Це просто символ. У ранніх діалектах Лиспа він мав свою мету: функції мали внутрішнє представлення у вигляді списків, і єдиним способом відрізнити функцію від звичайного списку була перевірка того, чи є перший його елемент символом **`lambda`**.

В Common Lisp ви можете задати функцію у вигляді списку, але вони будуть мати відмінне від списку внутрішнє уявлення, тому **`lambda`** більше не потрібно. Було б цілком можливо за приписувати функції, наприклад, так:

```
((x) (+ x 100))
```

замість

```
(lambda (x) (+ x 100))
```

але Лісп-програмісти звикли починати функції символом **`lambda`**, і Common Lisp також наслідує цієї традиції.

λ-числення

Усі мови функціонального програмування походять прямо чи опосередковано з роботи Алонцо Черчі та Стівена Кліне. Лямбда-числення було визначено Черчем і Кліне в 1930-х роках, до існування комп'ютерів. На той час математики були зацікавлені в формальному вираженні обчислень у письмовій формі, відмінній від англійської чи іншої неформальної мови. Лямбда-числення було розроблено як спосіб вираження тих речей, які можна обчислити. Це дуже маленька, функціональна мова програмування. У лямбда-числення функція - це відображення від елементів домену до елементів кодомену, заданого правилом.

Розглянемо функцію **куб(x)=x³**. Яке значення куба ідентифікатора **куб** у визначенні **куб(x)=x³**? Чи можна визначити цю функцію, не даючи їй імені?

λx.x³ визначає функцію, яка відображає кожне **x** у домені до **x³**. Можна сказати, що це визначення або *лямбда-абстракція*, **λx.x³**, є значенням, прив'язаним до куба ідентифікатора. Ми говоримо, що **x³** - тіло лямбда-абстракції. Кожна лямбда-абстракція в лямбда-позначеннях є функцією одного ідентифікатора. Однак лямбда-вирази можуть містити більше одного ідентифікатора.

Вираз **y^2+x** можна виразити як лямбда-абстракцію одним із двох способів:

$$\lambda x.\lambda y.y^2 + x$$

$$\lambda y.\lambda x.y^2 + x$$

У першій лямбда-абстракції **x** є першим параметром, який подається до виразу. У другій лямбда-абстракції параметр **y** є параметром для отримання значення першим. У будь-якому випадку абстракцію часто скорочують, викидаючи зайвий **λ**. У скороченій формі дві абстракції стали б **$\lambda xy.y^2+x$** та **$\lambda yx.y^2+x$**.

Сказати, що лямбда-числення або будь-яка мова має *нормальну форму*, означає, що кожен вираз, який можна скоротити, має найпростішу форму. Це означає, що ми можемо якимось механічним чином звести складніші вирази до більш простих. Лямбда-числення має властивість, що називається *злиттям*.

Злиття означає, що одна або кілька стратегій скорочення (або змішування їх) завжди призводять до однакової нормальної форми виразу, припускаючи, що вираз може бути зменшений стратегією скорочення. Ця властивість злиття була доведена в теоремі *Черча – Россера*.

Предикат **`(Ex) T(a,a,x)`** нерозв'язний, тобто функція:

$$\chi(a) = \begin{cases} 0, & \text{if}(Ex)T(a,a,x) \\ 1, & \text{else} \end{cases}$$

необчислювана.

Дане формулювання використовує поняття обчислюваності по Тьюрингу.

Застосування функції (тобто виклик функції) в лямбда-нотації записується з лямбда-абстракцією, за якою слідує значення, з яким потрібно викликати абстракцію. Таке поєднання називається *редекс*.

Для виклику $\lambda x.x^3$ зі значенням **2** для **x** ми б написали $(\lambda x.x^3)2$

Ця комбінація лямбда-абстракції та значення називається *редексом*.

Редекс - це лямбда-вираз, який може бути зменшений. Зазвичай лямбда-вираз містить кілька редексів, які можна вибрати для зменшення. Застосування функції є лівоасоціативним, що означає, що якщо на одному рівні вкладених дужок доступно більше одного редекса, то спочатку слід зменшити крайній лівий редекс. Якщо крайній лівий зовнішній редекс завжди вибирається для зменшення першим, порядок зменшення називається нормальним зменшенням порядку.

Коли редекс скорочується за допомогою застосування лямбда-числення, еквівалентного застосуванню функції, це називається *β-скороченням* (вираженим бета-скороченням). Зменшення нормального порядку

(λxyz.xz (yz)) (λx.x) (λxy.x)

наведено на рис. 10.2. Редекс, який буде β-зменшено на кожному кроці, підкреслено.

$$
\begin{aligned}
&\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)\\
\Rightarrow\ &\underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda xy.x)}\\
\Rightarrow\ &\lambda z.\underline{(\lambda x.x)z}((\lambda xy.x)z)\\
\Rightarrow\ &\lambda z.z(\underline{(\lambda xy.x)z})\\
\Rightarrow\ &\lambda z.z(\lambda y.z)\square
\end{aligned}
$$

Рис.3.1 Зниження нормального порядку

Для доступу до частин списків в Common Lisp є ще кілька функцій, які визначаються за допомогою **car** і **cdr**. Щоб отримати елемент з певним індексом, викличемо функцію **nth**:

```
[26]> (nth 0 `(a b c))
A
[27]> (nth 2 `(a b c))
C
```

Щоб отримати n-й хвіст списку, викличемо **nthcdr**:

```
[28]> (nthcdr 2 `(a b c))
(C)
```

Функції **nth** і **nthcdr** ведуть відлік елементів списку з **0**. Взагалі кажучи, в Common Lisp будь-яка функція, яка звертається до елементів структур даних, починає відлік з нуля.

Ці дві функції дуже схожі, і виклик **nth** відповідає виклику **car** від **nthcdr**. Визначимо **nthcdr** без обробки можливих помилок:

```
(defun our-nthcdr (n lst)
     (if (zerop n)
     lst
     (our-nthcdr (- n 1) (cdr lst))))
```

Функція **zerop** всього лише перевіряє, чи рівний нулю її аргумент.

Функція **last** повертає останню **cons**-комірку списку:

```
[32]> (last '(1 2 3))
(3)
```

Common Lisp визначає кілька операцій для застосування будь-якої функції до кожного елементу списку. Найчастіше для цього використовується **mapcar**, яка викликає задану функцію поелементно для одного або декількох списків і повертає список результатів:

```
[11]> (mapcar (function (lambda (x) (+ x 10)))`(1 2 3))
(11 12 13)
```

```
(mapcar (function (lambda (x) (+ x 10)))`(1 2 3))
```

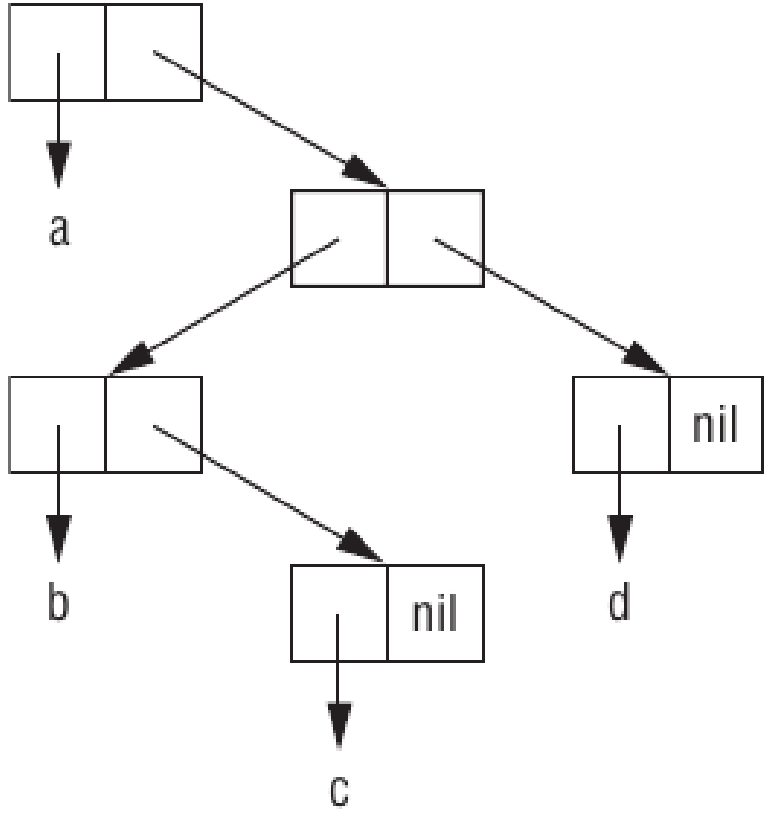
З останнього прикладу видно, як **mapcar** обробляє випадок зі списками різної довжини. Обчислення обривається після закінчення самого короткого списку.

Схожим чином діє **maplist**, проте застосовує функцію послідовно ні до **car**, а до **cdr** списку, починаючи з усього списку цілком.

```
[21]> (maplist (function (lambda (x) x)) `(a b c))
((A B C) (B C) (C))
```

(maplist (function (lambda (x) x)) `(a b c))

Cons-клітинки також можна розглядати як двійкові дерева: **`car`** відповідає праве піддерево, а **`cdr`** - ліве. Наприклад, список **`(a (b c) d)`** представлений у вигляді дерева на малюнку нижче.



В Common Lisp є кілька вбудованих функцій для роботи з деревами. Наприклад, **copy-tree** приймає дерево і повертає його копію. Визначимо аналогічну функцію самостійно:

```
[22]> (defun our-copy-tree (tr)
(if (atom tr)
tr
(cons (our-copy-tree (car tr))
(our-copy-tree (cdr tr)))))
OUR-COPY-TREE
```

```
(defun our-copy-tree (tr)
     (if (atom tr)
          tr
          (cons (our-copy-tree (car tr))
               (our-copy-tree (cdr tr)))))
```

Бінарні дерева без внутрішніх вузлів навряд чи виявляться корисними. Common Lisp включає в себе функції для операцій з деревами не тому, що без дерев не можна обійтися, а тому що ці функції дуже корисні для роботи зі списками та підсписків. Наприклад, припустимо, що у нас є список:

```
(and (integerp x) (zerop (mod x 2)))
```

І ми хочемо замінити **x** на **y**. Замінити елементи в послідовності можна за допомогою **substitute**:

```
[25]>  (substitute `y `x `(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP X) (ZEROP (MOD X 2)))
```

```
(substitute 'y 'x '(and (integerp x) (zerop (mod x 2))))
```

Як бачите, використання **substitute** не дало результатів, так як список містить три елементи, жоден з яких не є **x**. Тут нам знадобиться функція **subst**, що працює з деревами:

```
[26]>  (subst `y `x `(and (integerp x) (zerop (mod x 2))))
(AND (INTEGERP Y) (ZEROP (MOD Y 2)))
```

```
(subst 'y 'x '(and (integerp x) (zerop (mod x 2))))
```

Наше визначення **subst** буде дуже схоже на **copy-tree**:

```
(defun our-subst (new old tree)
     (if (eql tree old)
          new
          (if (atom tree)
               tree
               (cons (our-subst new old
                                (car tree))
                     (our-subst new old (cdr
                                tree))))))
```

Будь-які функції, які оперують з деревами, будуть виглядати схожим чином, рекурсивно викликаючи себе з **`car`** і **`cdr`**. Така рекурсія називається подвійною.

Для розгалудження функцій використовують також класичну функцію **cond**, яка дуже схожа на **if**.

```
(cond ((eql x 2) 30)
      ((eql x 3) 40)
      (T NIL))
```

Фактично складається з пар: умова-дія. Якщо умова істинна, то виконується дія і функція завершує своє виконання. Якщо умова є брехнею, то функція переходить до наступної пари. І так до тих пір, поки якась умова буде істинною.

Для того, щоб функція мала закінчення, в останній парі замість умови ставлять константу істини **T**, тим самим дія останньої пари буде завжди виконуватися, якщо умови попередніх пар брехливі.

Розглянемо приклад визначення функції факторіалу від цілого числа **x** за допомогою функції **cond**.

```
[1]> (defun our-f (x) (cond ((equal x 1) 1) (T (* x (our-f (- x 1))))))
OUR-F
[2]> (our-f 5)
120
[3]> (our-f 2)
2
```

```
(defun our-f (x)
     (cond ((equal x 1) 1)
           (T (* x (our-f (- x 1))))))
```

Щоб переконатися, що рекурсія робить те, що ми думаємо, досить запитати, чи покриває вона все варіанти. Подивимося, наприклад, на рекурсивну функцію для визначення довжини списку:

```
(defun len (lst)
     (if (null lst)
     0
     (+ (len (cdr lst)) 1)))
```

В цьому випадку ми використовуємо функцію розгалудження **if**.

Можна переконатися в коректності функції, перевіривши дві речі:

1. Вона працює зі списками нульової довжини, повертаючи `0`.
2. Якщо вона працює зі списками, довжина яких дорівнює `n`, то буде справедлива також і для списків довжиною `n + 1`.

Якщо обидва випадки вірні, то функція поводиться коректно на всіх можливих списках.

Перше твердження абсолютно очевидно: якщо `lst` - це `nil`, то функція тут же повертає `0`. Тепер припустимо, що вона працює зі списком довжиною `n`. Згідно з визначенням, для списку довжиною `n + 1` вона поверне число, на `1` більше довжини `cdr` списку, тобто `n + 1`.

Це все, що нам потрібно знати. Представляти всю послідовність викликів зовсім не обов'язково, так само як необов'язково шукати парні дужки в визначеннях функцій. Для більш складних функцій, наприклад подвійний рекурсії, випадків буде більше, але процедура залишиться колишньою. Наприклад, для функції `our-copy-tree` потрібно розглянути три випадки: атоми, прості осередки, дерева, що містять `n + 1` комірок.

Перший випадок носить назву *базового (base case)*. Якщо рекурсивна функція поводиться не так, як очікувалося, причина часто полягає в некоректній перевірки базового випадку або ж у відсутності перевірки, як в прикладі з функцією **member**:

```
(defun our-member (obj lst)
     (if (eql (car lst) obj)
         lst
         (our-member obj (cdr lst))))
```

У цьому визначенні необхідна перевірка списку на порожнечу, інакше в разі відсутності шуканого елемента в списку рекурсивний виклик буде виконуватися нескінченно.

Списки - хороший спосіб представлення невеликих множин. Щоб перевірити, чи належить елемент множині, що задається списком, можна скористатися функцією **member**:

```
[4]> (member `b `(a b c))
(B C)
```

(member 'b '(a b c))

Якщо шуканий елемент знайдений, то **member** повертає не **t**, а частину списку, яка починається з знайденого елемента. Звичайно, непорожній список логічно відповідає істині, але така поведінка **member** дозволяє отримати більше інформації. За замовчуванням member порівнює аргументи за допомогою **eql**. Предикат порівняння можна задати вручну за допомогою аргументу по ключу. Аргументи по *ключу (keyword)* - досить поширений в Common Lisp спосіб передачі аргументів. Такі аргументи передаються не у відповідності з їх становищем в списку параметрів, а за допомогою особливих міток, які називаються ключовими словами. Ключовим словом вважається будь-який символ, що починається з двокрапки.

Одним з аргументів по ключу, прийнятих **member**, є **:test**. Він дозволяє використовувати в якості предиката порівняння замість **eql** довільну функцію, наприклад **equal**:

```
[5]> (member `(a) `((a) (z)) :test `equal)
((A) (Z))
```

```
(member `(a) `((a) (z)) :test `equal)
```

Аргументи по ключу не є обов'язковими і слідують останніми у виклику функції, причому їх порядок не має значення.

Інший аргумент по ключу функції **member** - **:key**. З його допомогою можна задати функцію, яка застосовується до кожного елементу перед порівнянням:

```
[6]>  (member `a `((a b) (c d)) :key `car)
((A B) (C D))
```

¶

(member 'a '((a b) (c d)) :key 'car)

У цьому прикладі ми шукали елемент, **car** якого дорівнює **a**.

При бажанні використовувати обидва аргументи по ключу можна задавати їх в довільному порядку:

```
(member 2 '((1) (2)) :key 'car :test
'equal)
(member 2 '((1) (2)) :test 'equal :key
'car)
```

За допомогою **member-if** можна знайти елемент, що задовольняє безпідставного предикату, наприклад **oddp** (істинного, коли аргумент непарний):

```
[7]> (member-if `oddp `(2 3 4))
(3 4)
```

```
(member-if `oddp `(2 3 4))
```

Наша власна функція **member-if** могла б виглядати наступним чином:

```
(defun our-member-if (fn lst)
     (and (consp lst)
     (if (funcall fn (car lst)
          lst
           (our-member-if fn (cdr lst)))))
```

Функція **adjoin** - свого роду умовний cons. Вона приєднує заданий елемент до списку, але тільки якщо його ще немає в цьому списку (тобто не **member**):

```
[8]> (adjoin `a `(a b c))
(A B C)
[9]> (adjoin `z `(a b c))
(Z A B C)
```

У загальному випадку **adjoin** приймає ті ж аргументи по ключу, що і **member**.

Common Lisp визначає основні логічні операції з безлічі ми, такі як об'єднання, перетин, доповнення, для яких визначені відповідні функції: **union, intersection, set-difference**.

Ці функції працюють рівно з двома списками і мають ті ж аргументи по ключу, що і **member**.

```
[10]> (union `(1 2) `(2 3 4))
(1 2 3 4)
[11]> (intersection `(a b c) `(b b c))
(B C)
[12]> (intersection `(a b b c) `(b b c))
(B B C)
[13]> (set-difference `(a b c d e) `(b e))
(A C D)
```

Оскільки у величезних кількостях немає такого поняття, як впорядкування, ці функції не зберігають порядок елементів у вихідних списках. Наприклад, виклик s**`et-difference`** з прикладу може з тим же успіхом повернути **`(d c a)`**.

Списки також можна розглядати як послідовності елементів, що слідують один за одним у фіксованому порядку. В Common Lisp крім списків до послідовностей також відносяться вектори.
Довжина послідовності визначається за допомогою **`length`**:

```
[14]> (length `(a b c d))
4
```

Скопіювати частина послідовності можна за допомогою **subseq**. Другий аргумент (обов'язковий) задає початок підпослідовності, а третій (необов'язковий) - індекс першого елемента, що не підлягає копіюванню.

```
[15]> (subseq `(a b c d) 1 2)
(B)
[16]> (subseq `(a b c d) 1)
(B C D)
```

Якщо третій аргумент пропущено, то підпослідовність закінчується разом з вихідною послідовністю.

Функція **reverse** повертає послідовність, яка містить вихідні елементи в зворотному порядку:

```
> (reverse '(a b c))
(C B A)
```

За допомогою **reverse** можна, наприклад, шукати паліндроми, тобто послідовності, читаються однаково в прямому і зворотному порядку (наприклад, `(a b b a)`). Дві половини паліндрома з парною кількістю аргументів будуть дзеркальними відображеннями один одного.

Використовуючи **length**, **subseq** і **reverse**, визначимо функцію **mirror?**:

```
(defun mirror? (s)
     (let ((len (length s)))
       (and (evenp len)
         (let ((mid (/ len 2)))
           (equal (subseq s 0 mid)
            (reverse (subseq s mid)))))))

[18]> (mirror? '(a b b a))
T
```

```
[19]> (mirror? `(a r o z a u p a l a n a l
a p u a z o r a))
NIL
[21]> (length `(a r o z a u p a l a n a l
a p u a z o r a))
21
[22]> (mirror? `(a r o z a u p a l a a l a
p u a z o r a))
T
```

Для сортування послідовностей в Common Lisp є вбудована функція **sort**. Вона приймає список, що підлягає сортуванню, і функцію порівняння від двох аргументів:

```
[23]> (sort `(0 5 3 7 2 8 1) `>)
(8 7 5 3 2 1 0)
[24]> (sort `(0 5 3 7 2 8 1) `<)
(0 1 2 3 5 7 8)
```

З функцією **sort** слід бути обережними, тому що вона деструктивна. З міркувань продуктивності **sort** не створює новий список, а модифікує вихідний. Тому якщо ви не хочете змінювати вихідну послідовність, передайте в функцію її копію.

Використовуючи sort і nth, запишемо функцію, яка приймає ціле число n і повертає n-й елемент в порядку убування:

```
(defun nthmost (n lst)
     (nth (- n 1)
          (sort (copy-list lst) >)))

[25]> (defun nthmost (n lst) (nth (- n 1)
(sort (copy-list lst) `>)))
NTHMOST
[26]> (nthmost 2 '(0 2 1 3 8))
3
```

Функції **every** і **some** застосовують предикат до однієї або декількох послідовностей. Якщо передана тільки одна послідовність, вони перевіряють, чи задовольняє кожен її елемент цього предикату:

```
[27]> (every `oddp `(1 3 5))
T
[28]> (some `evenp `(1 2 3))
T
```

Якщо задано кілька послідовностей, предикат повинен приймати кількість аргументів, що дорівнює кількості послідовностей, і з кожної послідовності аргументи беруться по одному:

```
[29]> (every `> `(1 3 5) `(0 2 4))
T
```

Подання списків у вигляді осередків дозволяє легко використовувати їх в якості стопки (stack). В Common Lisp є два макроси для роботи зі списком як зі стопкою:

`(push x y)` кладе об'єкт **`x`** на вершівку стопки **`y`**,

`(pop x)` знімає зі стопки верхній елемент. Обидва ці макросу можна визначити за допомогою функції **`setf`**.

Виклик **`(push obj lst)`** транслюється до

```
(setf lst (cons obj lst))
```

А виклик **`(pop lst)`** до

```
(let ((x (car lst))
(setf lst (cdr lst))
x)
```

Наступна лекція буде присвячена продовженню розгляду реалізаціям на мові Common Lisp.