Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

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

Лекція 4

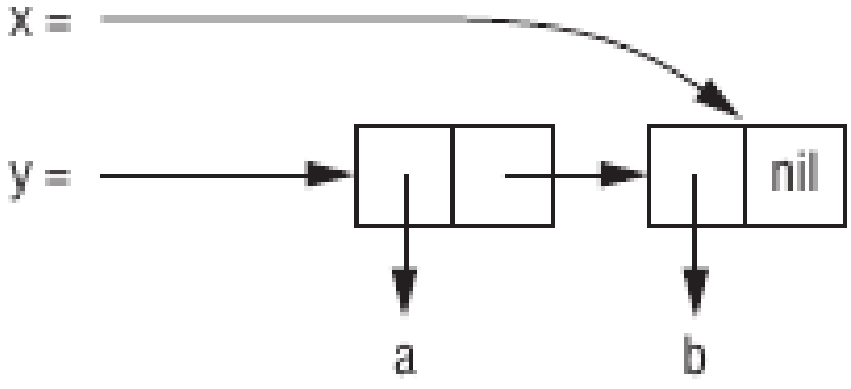
Мова штучного інтелекту

LISP (продовження 2)

Продовжимо розгляд роботи зі стековою памяттю (стопками). Розглянемо приклад.

```
[1]> (setf x `(b))
(B)
[2]> (push `a x)
(A B)
[3]> x
(A B)
[4]> (setf y x)
(A B)
[5]> (pop x)
A
[6]> x
(B)
[7]> y
(A B)
```

Структура осередків після виконання наведених виразів показана на малюнку нижче.



За допомогою **push** можна також визначити ітеративний варіант функції **reverse** для списків:

```
(defun our-reverse (lst)
     (let ((acc nil))
          (dolist (elt lst)
               (push elt acc))
          acc))
```

В цьому варіанті ми починаємо з порожнього списку і послідовно кладемо на нього, як на стопку, елементи вихідного. Останній елемент виявиться на вершині стопки, тобто на початку списку.

Макрос **pushnew** схожий на **push**, проте використовує **adjoin** замість **cons**:

```
[8]> (let ((x `(a b)))
       (pushnew `c x)
       (pushnew `a x)
       x)
(C A B)
```

Елемент **a** вже присутній в списку, тому не додається.

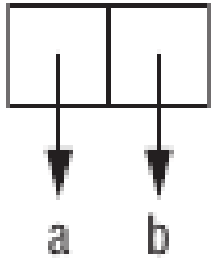
Списки, які можуть бути побудовані за допомогою **`list`**, називаються *правильними списками (proper list)*. Знову ж правильним списком вважається або **`nil`**, або **`cons`**-клітинка, **`cdr`** якої - також правильний список. Таким чином, можна визначити предикат, який повертає істину тільки для правильного списку:

```
[9]> (defun proper-list? (x)
(or (null x)
(and (consp x)
(proper-list? (cdr x)))))
PROPER-LIST?
```

Виявляється, за допомогою **cons** можна створювати не тільки правильні списки, а й структури, що містять рівно два елементи. При цьому **car** відповідає першому елементу структури, а **cdr** — другого.

```
[10]> (setf pair (cons `a `b))
(A . B)
```

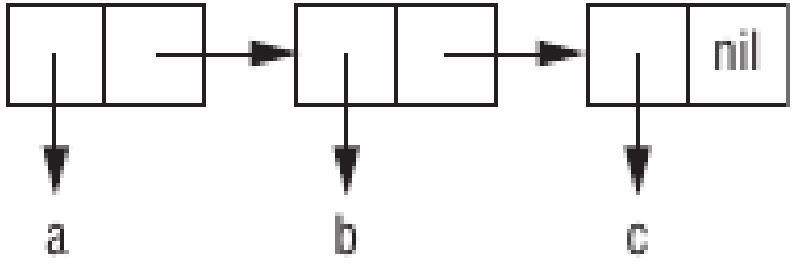
Оскільки ця **cons**-клітинка не є правильним списком, при відображенні її **car** і **cdr** розділяються крапкою. Такі клітинки називаються *точковими парами*.

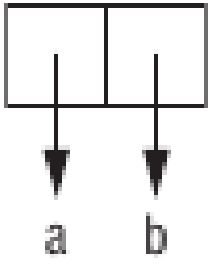


Правильні списки можна задавати і у вигляді набору точкових пар, але вони будуть відображатися у вигляді списків:

```
[11]> `(a . (b . (c . nil)))
(A B C)
```

Зверніть увагу, як співвідносяться коміркова і точкова нотації

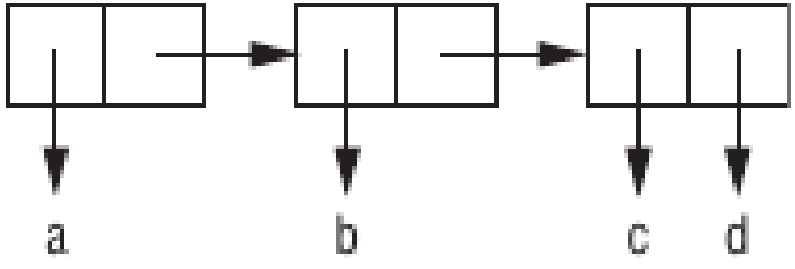




Допустима також змішана форма запису:

```
[14]> (cons `a (cons `b (cons `c `d)))
(A B C . D)
```

Структура такого списку показана нижче:



Таким чином, список **(a b)** може бути записаний аж чотирма способами:

```
(a  . (b . nil))        (a . (b))
(a  b . nil)            (a b)
```

і при цьому Лисп відобразить їх однаково.

Також цілком природно задіяти **cons**-клітинки для подання відображень. Список точкових пар називається *асоціативним списком (assoc-list, alist)*. За допомогою нього легко визначити набір будь-яких правил і відповідностей, наприклад:

```
[18]> (setf trans `((+ . "add") (- . "subtract")))
((+ . "add") (- . "subtract"))
```

Асоціативні списки повільні, але вони зручні на початкових етапах роботи над програмою. В Common Lisp є вбудована функція **assoc** для отримання по ключу відповідної йому пари в такому списку:

```
[19]> (assoc '+ trans)
(+ . "add")
[20]> (assoc '* trans)
NIL
```

Якщо **assoc** нічого не знаходить, повертається **nil**.
Спробуємо визначити спрощений варіант функції **assoc**:

```
(defun our-assoc (key alist)
     (and (consp alist)
          (let ((pair (car alist)))
            (if (eql key (car pair))
             pair
            (our-assoc key (cdr alist))))))
```

Як і **member**, реальна функція **assoc** приймає кілька аргументів по ключу,включаючи **:test** і **:key**. Також Common Lisp визначає **assoc-if**, яка працює за аналогією з **member-if**.

Приклад програми пошуку мінімального шляху

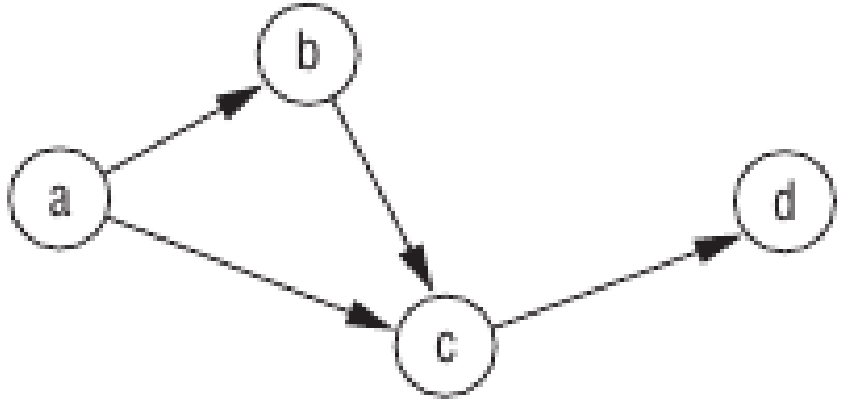
```
(defun shortest-path (start end net)
     (bfs end (list (list start)) net))
(defun bfs (end queue net)
 (if (null queue)
  nil
  (let ((path (car queue)))
   (let ((node (car path)))
    (if (eql node end)
     (reverse path)
       (bfs end
           (append (cdr queue)
              (new-paths path node net))
            net))))))
```

```
(defun new-paths (path node net)
  (mapcar #'(lambda (n)
              (cons n path))
          (cdr (assoc node net))))
```

Вище показана програма, що обчислює найкоротший шлях на графі (або мережі). Функції **`shortest-path`** необхідно повідомити початкову і кінцеву точки, а також саму мережу, і вона поверне найкоротший шлях між ними, якщо він взагалі існує.

У цьому прикладі вузлів відповідають символи, а сама мережа представле на як асоціативний список елементів виду **`(вузол . сусіди)`**.

Невелика мережа



може бути подана таким чином:

```
(setf min '((a b c) (b c) (c d)))
```

Стоїть задача знайти вузли, в які можна потрапити з вузла **a**, і в цьому нам допоможе функція **assoc**:

```
[28]> (setf min '((a b c)(b c)(c d)))
((A B C) (B C) (C D))
[29]> (cdr (assoc `a min))
(B C)
```

Програма вище (слайди 13-14) реалізує *пошук в ширину (breadth-first search)*. Кожен шар мережі досліджується по черзі один за одним, поки не буде знайдено потрібний елемент або досягнутий кінець мережі. Послідовність досліджуваних вузлів представляється у вигляді черги.

Наведений вище код злегка ускладнює цю ідею, дозволяючи не тільки прийти до пункту призначення, але ще і зберегти запис про те, як ми туди дісталися. Таким чином, ми оперуємо ні з чергою вузлів, а з чергою пройдених шляхів.

Пошук виконується функцією **`bfs`**. Спочатку в черзі тільки один елемент - шлях до початкового вузла. Таким чином, **`shortest-path`** викликає **`bfs`** з **`(list (list start))`** в якості вихідної черги.

Перше, що повинна зробити **bfs**, - перевірити, чи залишилися ще непройденние вузли. Якщо чергу порожня, **bfs** повертає **nil**, сигналізуючи, що шлях не був знайдений. Якщо ж ще є неперевірені вузли, **bfs** бере перший з черги. Якщо **car** цього вузла містить шуканий елемент, значить, ми знайшли шлях до нього, і ми повертаємо його, попередньо розгорнувши. В іншому випадку ми додаємо всі дочірні вузли в кінець черги. Потім ми рекурсивно викликаємо **bfs** і переходимо до наступного шару.

Так як **bfs** здійснює пошук в ширину, то перший знайдений шлях буде одночасно найкоротшим або одним з найкоротших, якщо є й інші шляхи такої ж довжини:

```
> (shortest-path 'a 'd min)
(A C D)
```

А ось як виглядає відповідна чергу під час кожного з викликів **`bfs`**:

```
((A))
((B A) (C A))
((C A) (C B A))
((C B A) (D C A))
((D C A) (D C B A))
```

У кожної наступної черги другий елемент попередньої черги стає першим, а перший елемент стає хвостом **`(cdr)`** будь-яких нових елементів в кінці наступної черги.

Раніше нами були розглянуті списки - найбільш універсальні структури для зберігання даних. Далі в цій лекції будуть розглянуті інші способи зберігання даних в Ліспі: масиви (а також вектори і рядки), структури і хеш-таблиці. Вони не настільки гнучкі, як списки, але дозволяють здійснювати більш швидкий доступ і займають менше місця.

В Common Lisp масиви створюються за допомогою функції **`make-array`**, першим аргументом якої виступає список розмірностей. Створимо масив **`2 × 3`**:

```
[40]> (setf arr (make-array `(2
3) :initial-element nil))
#2A((NIL NIL NIL) (NIL NIL NIL))
```

Багатовимірні масиви в Common Lisp можуть мати щонайменше 7 розмірностей, а в кожному вимірі підтримується зберігання не менше 1 023 елементів.
Аргумент **`:initial-element`** не є обов'язковим. Якщо він використовується, то встановлює початкове значення кожного елемента масиву. Поведінка системи при спробі отримати значення елемента масиву, що не ініціалізувати початковим значенням, не визначене.

Щоб отримати елемент масиву, скористаємося **aref**. Як і більшість інших функцій доступу в Common Lisp, **aref** починає відлік елементів з нуля:

```
[41]> (aref arr 0 0)
NIL
```

Нове значення елемента масиву можна встановити, використовуючи **setf** разом з **aref**:

```
[42]> (setf (aref arr 0 0) `b)
B
[43]> (aref arr 0 0)
B
```

Як і списки, масиви можуть бути задані буквально за допомогою синтаксису **#na**, де **n** - кількість розмірностей масиву. Наприклад, поточний стан масиву arr може бути задано так:

```
# 2a ((b nil nil) (nil nil nil))
```

Якщо глобальна змінна ***print-array*** встановлена в **t**, масиви будуть друкуватися в такому вигляді:

```
[44]>  (setf *print-array* t)
T
[45]> arr
#2A((B NIL NIL) (NIL NIL NIL))
```

Для створення одновимірного масиву можна замість списку розмірностей через перший аргумент передати функції **make-array** ціле число:

```
[46]> (setf vec (make-array 4 :initial-element nil))
#(NIL NIL NIL NIL)
```

Одновимірний масив також називають вектором. Створити і заповнити вектор можна за допомогою функції **vector**:

```
[47]> (vector "a" `b 3)
#("a" B 3)
```

Як і масив, який може бути заданий буквально за допомогою синтаксису **`#na`**, вектор може бути заданий буквально за допомогою синтаксису **`# ()`**.
Хоча доступ до елементів вектора може здійснити **`aref`**, для роботи з векторами є більш швидка функція **`svref`**:

```
[48]>  (svref vec 0)
NIL
```

Префікс «**`sv`**» розшифровується як «*simple vector*». За замовчуванням всі вектори створюються як прості вектори.

Приклад: бінарний пошук

Зараз в якості прикладу покажемо, як написати функцію пошуку елемента в відсортованому векторі. Якщо нам відомо, що елементи вектора розташовані в певному порядку, то пошук потрібного елемента може бути виконаний швидше, ніж за допомогою функції **`find`**. Замість того щоб послідовно перевіряти елемент за елементом, ми відразу переміщаємося в середину вектора. Якщо середній елемент відповідає шуканого, то пошук закінчений. В іншому випадку ми продовжували пошук в правій або лівій половині в залежності від того, більше чи менше шуканого значення цей середній елемент вектора. Нижче наведена програма, яка працює подібним чином. Вона складається з двох функцій: **`bin-search2`** визначає межі пошуку і передає управління функції **`finder`**, яка шукає відповідний елемент між позиціями **`start`** і **`end`** вектора **`vec`**.

Пошук в відсортованому векторі:

```
(defun bin-search (obj vec)
  (let ((len (length vec)))
   (and (not (zerop len))
        (finder obj vec 0 (- len 1)))))
```

```
(defun finder (obj vec start end)
  (let ((range (- end start)))
    (if (zerop range)
     (if (eql obj (aref vec start))
       obj
       nil)
         (let ((mid (+ start (round (/
range 2)))))
         (let ((obj2 (aref vec mid)))
         (if (< obj obj2)
     (finder obj vec start (- mid 1))
         (if (> obj obj2)
     (finder obj vec (+ mid 1) end)
          obj)))))))
```

Пояснимо, що відбувається в наших функціях.
Коли область пошуку скорочується до одного елемента, повертається сам елемент в разі його відповідності згаданої значенням **`obj`**, в іншому випадку - **`nil`**. Якщо область пошуку складається з декількох елементів, визначається її середній елемент - **`obj2`** (функція round повертає найближче ціле число), який порівнюється з шуканим елементом **`obj`**. Якщо **`obj`** менше **`obj2`**, пошук триває рекурсивно в лівій половині вектора, в іншому випадку - в правій половині. Залишається варіант **`obj = obj2`**, але це означає, що шуканий елемент знайдений і ми просто його повертаємо.

Якщо вставити наступний рядок в початок визначення функції **finder**,

```
(format t "~A~%" (subseq vec start (+ end 1)))
```

ми зможемо спостерігати за процесом відсікання половин на кожному кроці:

```
> (bin-search 3 #(0 1 2 3 4 5 6 7 8 9))
#(0 1 2 3 4 5 6 7 8 9)
#(0 1 2 3)
#(3)
3
```

Ми з вами ще підіймали один пласт програмування на Ліспі — рядки і знаки.
Рядки - це вектори, що складаються із знаків. Рядком прийнято називати набір знаків, укладений в подвійні лапки. Одиночний знак, наприклад **`c`**, задається так: **`#\c`**.
Кожен знак відповідає певному цілому числу, як правило, (хоча і не обов'язково) відповідно до ASCII. У більшості реалізацій є функція **`char-code`**, яка повертає пов'язане зі знаком число, і функція **`code-char`**, що виконує зворотне перетворення.
Для порівняння знаків використовуються наступні функції: **`char<`**(менше), **`char<=`** (менше або дорівнює), **`char=`** (дорівнює), **`char>=`** (більше або дорівнює), **`char>`** (більше) і **`char/=`** (не дорівнює) . Вони працюють так само, як і функції порівняння чисел.

```
[50]> (sort "elbow" `char<)
"below"
```

Оскільки рядки - це масиви, то до них застосовні всі операції з масивами. Наприклад, отримати знак, що знаходиться в конкретній позиції, можна за допомогою **aref**:

```
[51]>  (aref "abc" 1)
#\b
```

Однак ця операція може бути виконана швидше за допомогою спеціалізованої функції **char**:

```
[55]> (char "abc" 1)
#\b
```

Функція **char**, як і **aref**, може бути використана разом з **setf** для заміни елементів:

```
[56]>  (let ((str (copy-seq "Merlin")))
(setf (char str 3) #\k)
str)
"Merkin"
```

Щоб порівняти два рядки, можна скористатися відомою вам функцією **equal**, але є також і спеціалізована **string-equal**, яка до того ж не враховує регістр букв:

```
[1]>  (equal "fred" "fred")
T
[2]>  (equal "fred" "Fred")
NIL
[3]>  (string-equal "fred" "Fred")
T
```

Є кілька способів створення рядків. Самий загальний - за допомогою функції **format**. При використанні **nil** в якості її першого аргументу **format** поверне рядок, замість того щоб її надрукувати:

```
[4]>  (format nil "~A or ~A" "truth"
"dare")
"truth or dare"
```

Але якщо вам потрібно просто з'єднати кілька рядків, можна скористатися **concatenate**, яка приймає тип результату і одну або кілька послідовностей:

```
[5]>  (concatenate `string  "not " "to
worry")
"not to worry"
```

Тип *послідовність (sequence)* в Common Lisp включає в себе списки і вектори (а значить, і рядки). Багато функцій з тих, які ми раніше використовували для списків, насправді визначені для будь-яких послідовностей. Це, наприклад, **`remove`**, **`length`**, **`subseq`**, **`reverse`**, **`sort`**, **`every`**, **`some`**. Таким чином, функція **`mirror?`**, визначена нами раніше, буде працювати і з іншими видами послідовностей:

```
> (mirror? "abba")
T
```

Ми вже знаємо деякі функції для доступу до елементів послідовностей: **`nth`** для списків, **`aref`** і **`svref`** для векторів, **`char`** для рядків. Доступ до елементу послідовності будь-якого типу може бути здійснений за допомогою **`elt`**:

```
> (elt '(a b c) 1)
B
```

Спеціалізовані функції працюють швидше, і використовувати **`elt`** рекомендується тільки тоді, коли тип послідовності заздалегідь не відомий.
За допомогою **`elt`** функція **`mirror?`** може бути оптимізована для векторів:

```
(defun mirror? (s)
  (let ((len (length s)))
    (and (evenp len)
     (do ((forward 0 (+ forward 1))
      (back (- len 1) (- back 1)))
      ((or (> forward back)
       (not (eql (elt s forward)
           (elt s back))))
       (> forward back))))))
```

Ця версія, як і раніше буде працювати і зі списками, однак вона більш пристосована для векторів. Регулярне використання послідовного доступу до елементів списку досить затратно, а безпосереднього доступу до потрібного елементу вони не надають. Для векторів ж вартість доступу до будь-якого елементу не залежить від його положення.

Багато функцій, що працюють з послідовностями, мають кілька аргументів по ключу:

Параметр	Призначення	За замовчу-ванням
`:key`	Функція, що застосовується до кожного елементу	i**`dentity`**
`:test`	Предикат для порівняння	**`eql`**
`:from-end`	Якщо **`t`**, робота з кінця	**`nil`**
`:start`	Індекс елемента, з якого починається виконання	**`0`**
`:end`	Якщо заданий, то індекс елемента, на якому слід зупинитися	**`nil`**

Одна з функцій, яка приймає всі ці аргументи, - **`position`**. Вона повертає положення певного елемента в послідовності або **`nil`** в разі його відсутності. Подивимося на роль аргументів по ключу на прикладі **`position`**:

```
[10]>(position #\a "fantasia" :from-end t)
7
```

ми отримуємо позицію елемента, найближчого до кінця послідовності. Але позиція елемента обчислюється як зазвичай, тобто від початку списку (проте пошук елемента проводиться з кінця списку).

Параметр **`:key`** визначає функцію, яка застосовується до кожного елементу перед порівнянням його з шуканим:

```
[15]> (position `a `((c d)(a b)) :key `car)
1
```

У цьому прикладі ми поцікавилися, **`car`** якого елемента містить **`a`**.

Параметр **:test** визначає, за допомогою якої функції будуть порівнюватися елементи. За замовчуванням використовується **eql**. Якщо вам необхідно порівнювати списки, доведеться скористатися функцією **equal**:

```
> (position '(a b) '((a b)(c d)))
NIL
> (position '(a b) '((a b)(c d)) :test 'equal)
0
```

Аргумент **:test** може бути будь-якою функцією від двох елементів. Наприклад, за допомогою **<** можна знайти перший елемент, більший заданого:

```
> (position 3 '(1 0 7 5) :test '<)
2
```

Пошук елементів, що задовольняють заданій предикату, здійснюється за допомогою **position-if**. Вона приймає функцію і послідовність, повертаючи положення першого зустрінутого елемента, який задовольняє предикату:

```
[24]> (position-if `oddp `( 2 3 4 5))
1
```

Ця функція приймає всі перераховані вище аргументи по ключу, за винятком **:test**.

Також для послідовностей визначені функції, аналогічні **member** і **member-if**. Це **find** (приймає всі аргументи по ключу) і **find-if** (приймає всі аргументи, крім **:test**):

```
[26]>  (find #\a "cat")
#\a
[28]>  (find-if `characterp "ham")
#\h
```

На відміну від **member** і **member-if**, вони повертають тільки сам знайдений елемент.

Замість **find-if** іноді краще використовувати **find** з ключем **:key**. Наприклад, вираз:

```
(find-if #'(lambda (x)
             (eql (car x) 'complete))
         lst)
```

буде виглядати більш зрозумілою у вигляді:

```
(find 'complete lst :key 'car)
```

```
[30]> (remove-duplicates "abracadabra")
"cdbra"
```

Функції **remove** і **remove-if** працюють з послідовностями будь-якого типу. Різниця між ними точно така ж, як між **find** і **find-if**. Пов'язана з ними функція **remove-duplicates** видаляє всі повторювані елементи послідовності, крім останнього:

```
[30]> (remove-duplicates "abracadabra")
"cdbra"
```

Ця функція використовує всі аргументи по ключу, розглянуті в таблиці вище.

Функція **`reduce`** зводить послідовність в одне значення. Вона приймає функцію, по крайней мере, з двома аргументами і послідовність. Задана функція спочатку застосовується до перших двом елементам послідовності, а потім послідовно до отриманого результату і наступного елемента послідовності. Останнє отримане значення буде повернуто як результат **`reduce`**. Таким чином, виклик:

```
(reduce 'fn' (a b c d))
```

буде еквівалентний

```
(fn (fn (fn 'a 'b) 'c)'d)
```

Хороше застосування **reduce** - розширення набору аргументів для функцій, які приймають тільки два аргументи. Наприклад, щоб отримати перетин трьох або більше списків, можна написати:

```
[30]> (reduce `intersection `((b r a d `s) (b a d) (c a t)))
(A)
```

Структура може розглядатися як більш просунутий варіант вектора. Припустимо, що нам потрібно написати програму, яка відслідковує положення набору паралелепіпедів. Кожне таке тіло можна представити у вигляді вектора, що складається з трьох елементів: висота, ширина і глибина. Програму буде простіше читати, якщо замість простих **`svref`** ми будемо використовувати спеціальні функції:

```
(defun block-height (b) (svref b 0))
```

і так далі. Можете вважати структуру таким вектором, у якого всі ці функції вже задані.

Визначити структуру можна за допомогою **defstruct**. У найпростішому випадку досить задати імена структури і її полів:

```
[31]> (defstruct point
x
y)
POINT
```

Ми визначили структуру **point**, маючі два поля, **x** і **y**. Крім того, неявно були задані функції: **make-point**, **point-p**, **copy-point**, **point-x**, **point-y**.

Раніше ми згадували про здатність Лісп-програм писати інші Лісп-програми. Це один з наочних прикладів: при виклику **`defstruct`** самостійно визначає всі необхідні функції. Навчившись працювати з макросами, ви самі зможете робити схожі речі. (Ви навіть змогли б написати свою версію **`defstruct`**, якби в цьому була необхідність.)

Кожен виклик **`make-point`** повертає новостворений екземпляр структури point. Значення полів можуть бути спочатку задані за допомогою відповідних аргументів по ключу:

```
(setf p (make-point :x 0 :y 0))
#S(POINT :X 0 :Y 0)
```

Функції доступу до полів структури визначені не тільки для читання полів, але і для завдання значень за допомогою **setf**:

```
[33]> (point-x p)
0
[34]> (setf (point-y p) 2)
2
[35]> p
#S(POINT :X 0 :Y 2)
```

Визначення структури також призводить до визначення однойменного типу. Кожен екземпляр **point** належить типу **point**, потім **structure**, потім **atom** і **t**. Таким чином, використання **point-p** рівносильно перевірці типу:

```
[37]>  (point-p p)
T
[38]> (typep p `point)
T
```

Функція **typep** перевіряє об'єкт на приналежність до заданого типу.

Також можна задати значення полів за замовчуванням, якщо укласти ім'я відповідного поля в список і помістити в нього вираз для обчислення цього значення.

```
[39]> (defstruct polemic
(type (progn
(format t "What kind of polemic was it? ")
(read)))
(effect nil))
POLEMIC
```

Виклик **make-polemic** без додаткових аргументів встановить вихідні значення полів:

```
[40]>  (make-polemic)
What kind of polemic was it? scathing
#S(POLEMIC :TYPE SCATHING :EFFECT NIL)
```

Крім того, можна управляти такими речами, як спосіб відображення структури і префікс імен функцій для доступу до полів. Ось більш розвинений варіант визначення структури **point**:

```
[41]> (defstruct (point (:conc-name p)
(:print-function print-point))
(x 0)
(y 0))
POINT
[42]> (defun print-point (p stream depth)
(format stream "#<~A, ~A>" (px p) (py p)))
PRINT-POINT
```

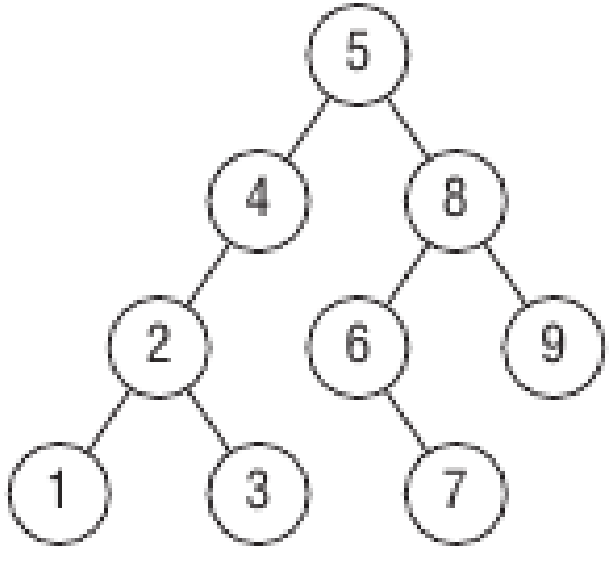
Аргумент: **`conc-name`** задає префікс, з якого будуть починатися імена функцій для доступу до полів структури. За замовчуванням він дорівнює **`point-`**, а в новому визначенні це просто **`p`**. Відхід від варіанту за замовчуванням робить код менш читабельним, тому використовувати більш короткий префікс стоїть, тільки якщо вам належить постійно користуватися функціями доступу до полів.
Параметр: **`print-function`** - це ім'я функції, яка буде викликатися для друку об'єкта, коли його потрібно буде відобразити (наприклад, в top level). Така функція повинна приймати три аргументи: сам об'єкт; потік, куди він буде надрукований; третій аргумент зазвичай не потрібно і може бути проігноровано. Слід сказати, що другий аргумент, потік, може бути переданий функції **`format`**.

Функція **print-point** відображатиме структуру в такий скороченій формі:

```
[43]>  (make-point)
#<0, 0>
```

Розглянемо приклад побудови двійкових дерев пошуку. Оскільки в Common Lisp є вбудована функція **`sort`**, вам, швидше за все, не доведеться самостійно писати процедури пошуку. Ми розглянемо, як вирішити схоже завдання, для якої немає вбудованої функції: підтримання набору об'єктів в відсортованому вигляді.
Ми розглянемо метод зберігання об'єктів в двійковому дереві пошуку (BST). Збалансоване BST дозволяє шукати, додавати або видаляти елементи за час, пропорційне **`log n`**, де **`n`** - кількість об'єктів в наборі.

BST - це бінарне дерево, в якому для кожного елемента і деякої функції впорядкування (нехай це буде функція **<**) дотримується правило: лівий дочірній елемент **<** елемента-батька, і сам елемент **>** правого дочірнього елемента. На малюнку нижче показаний приклад BST, упорядкованого за допомогою функції **<**.



```
(defstruct (node (:print-function
                  (lambda (n s d)
                    (format s "#<~A>" (node-elt n)))))
  elt (l nil) (r nil))

(defun bst-insert (obj bst <)
  (if (null bst)
      (make-node :elt obj)
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (make-node
                  :elt elt
                  :l   (bst-insert obj (node-l bst) <)
                  :r   (node-r bst))
                (make-node
                  :elt elt
                  :r   (bst-insert obj (node-r bst) <)
                  :l   (node-l bst)))))))

(defun bst-find (obj bst <)
  (if (null bst)
      nil
      (let ((elt (node-elt bst)))
        (if (eql obj elt)
            bst
            (if (funcall < obj elt)
                (bst-find obj (node-l bst) <)
                (bst-find obj (node-r bst) <))))))

(defun bst-min (bst)
  (and bst
       (or (bst-min (node-l bst)) bst)))

(defun bst-max (bst)
  (and bst
       (or (bst-max (node-r bst)) bst)))
```

Програма на малюнку вище містить утиліти для вставки і пошуку об'єктів в BST. В якості основної структури даних використовуються вузли.
Кожен вузол має три поля: в одному зберігається сам об'єкт, в двох інших - лівий і правий нащадки. Можна розглядати вузол як **`cons`**-осередок з одним **`car`** і двома **`cdr`**.
BST може бути або **`nil`**, або вузлом, піддерева якого (**`l`** і **`r`**) також є BST. Продовжимо подальшу аналогію зі списками. Як список може бути створений послідовністю викликів **`cons`**, так і бінарне дерево може бути побудовано за допомогою викликів **`bst-insert`**.

Цій функції необхідно повідомити об'єкт, дерево і функцію впорядкування.

```
> (setf nums nil)
NIL
> (Dolist (x `(5 8 4 2 1 9 6 7 3))
(setf nums (bst-insert x nums `<)))
NIL
```

Тепер дерево **nums** відповідає малюнку на слайді 57.

Функція **bst-find**, яка шукає об'єкти в дереві, приймає ті ж аргументи, що і **bst-insert**. Аналогія зі списками стане ще зрозуміліше, якщо ми порівняємо визначення **bst-find** і **our-member**.

Як і **member**, **bst-find** повертає не саме елемент, а його піддерево:

```
> (bst-find 12 nums # '<)
NIL
> (bst-find 4 nums # '<)
# <4>
```

Таке уявлення дозволяє нам розрізняти випадки, в яких шуканий елемент не знайдений (**nil**) і в яких успішно знайдений елемент **nil**.

Знаходження найбільшого і найменшого елементів BST також не складає особливих труднощів. Щоб знайти ми мінімальними елемент, ми йдемо по дереву, завжди вибираючи ліву гілку (**bst-mi**n). Аналогічно, слідуючи правим піддерев, ми отримаємо найбільший елемент (**bst-max**):

```
> (bst-min nums)
# <1>
> (bst-max nums)
# <9>
```

Видалення елемента з бінарного дерева виконується так само швидко, але відповідний код виглядає складніше.

```
(defun bst-remove (obj bst <)
(if (null bst)
nil
(let ((elt (node-elt bst)))
(if (eql obj elt)
(percolate bst)
(if (funcall < obj elt)
(make-node
:elt elt
:l (bst-remove obj (node-l bst) <)
:r (node-r bst))
(make-node
:elt elt
:r (bst-remove obj (node-r bst) <)
:l (node-l bst)))))))
```

```
(defun percolate (bst)
(let ((l (node-l bst)) (r (node-r bst)))
(cond ((null l) r)
((null r) l)
(t (if (zerop (random 2))
(make-node :elt (node-elt (bst-max l))
:r r
:l (bst-remove-max l))
(make-node :elt (node-elt (bst-min r))
:r (bst-remove-min r)
:l l))))))
```

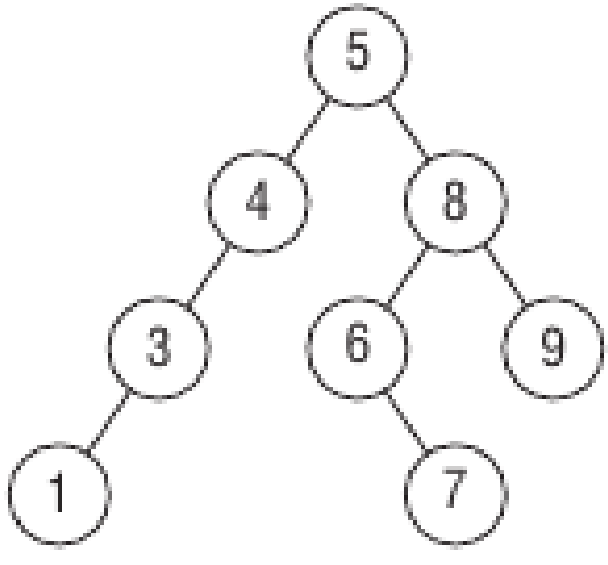
```
(defun bst-remove-min (bst)
(if (null (node-l bst))
(node-r bst)
(make-node :elt (node-elt bst)
:l (bst-remove-min (node-l bst))
:r (node-r bst))))

(defun bst-remove-max (bst)
(if (null (node-r bst))
(node-l bst)
(make-node :elt (node-elt bst)
:l (node-l bst)
:r (bst-remove-max (node-r bst)))))
```

Функція **bst-remove1** приймає об'єкт, дерево і функцію впорядкування і повертає цей же дерево без заданого елемента. Як і **remove**, **bst-remove** не змінюйте початкове дерево:

```
> (setf nums (bst-remove 2 nums '<))
# <5>
> (bst-find 2 nums '<)
NIL
```

Тепер дерево **nums** відповідає малюнку нижче



Видалення - більш витратна процедура, так як під видаляється об'єктом з'являється незайняте місце, яке повинно бути заповнене одним з піддерев цього об'єкта. Цим займається функція **`percolate`**.

Вона заміщає елемент дерева одним з його піддерев, потім заміщає це піддерево одним з його піддерев і так надалі.

Щоб збалансувати дерево, **`percolate`** випадковим чином вибирає одне з двох піддерев. Вираз **`(random 2)`** поверне або **`0`**, або **`1`**, в результаті **`(zerop (random 2))`** було це слово в половині випадків.

Тепер, коли ми перетворили набір об'єктів в бінарне дерево, послідовний обхід його елементів дасть нам них в порядку зростання.

```
(defun bst-traverse (fn bst)
(when bst
(bst-traverse fn (node-l bst))
(funcall fn (node-elt bst))
(bst-traverse fn (node-r bst))))
```

Для цієї мети визначена функція **bst-traverse**, яка застосовує до кожного елементу дерева функцію **fn**.

```
> (bst-traverse `princ nums)
13456789
NIL
```

(Функція **princ** всього лише відображає окремий об'єкт.)

Код, представлений в цьому розділі, є основою для реалізації довічних дерев пошуку. Ймовірно, ви захочете якось удосконалити його відповідно до ваших потреб. Наприклад, кожен вузол в поточній реалізації має лише одне поле elt, в той час як може виявитися корисним введення двох полів - ключ і значення.
Дана версія хоча і не підтримує таку можливість, але дозволяє її легко реалізувати.

Двійкові дерева пошуку можуть використовуватися не тільки для управління відсортованим набором об'єктів. Їх застосування оптимально, коли вставки і видалення вузлів мають рівномірний розподіл. Це означає, що для роботи, наприклад, з чергами, BST не найкращий вибір.
Хоча вставки в чергах цілком можуть бути розподілені рівномірно, видалення будуть завжди здійснюватися з кінця. Це буде приводити до розбалансування дерева, і замість очікуваної оцінки `O(log n)` ми отримаємо `O(n)`. Крім того, для моделювання черг зручніше використовувати звичайний список просто тому, що BST буде вести себе в кінцевому рахунку так само, як і список.

І в кінці лекції розглянемо особливості реалізації хеш-таблиць. Раніше було показано, що списки можуть використовуватися для подання множин і відображень. Для досить великих масивів даних (починаючи вже з 10 елементів) використання хеш-таблиць істотно збільшить продуктивність. Хеш-таблицю можна створити за допомогою функції **`make-hash-table`**, яка не вимагає обов'язкових аргументів:

```
> (setf ht (make-hash-table))
# <hash-Table BF0A96>
```

Хеш-таблиці, як і функції, при друку відображаються у вигляді **`# <...>`**.

Хеш-таблиця, як і асоціативний список, - це спосіб асоціювання пар об'єктів. Щоб отримати значення, пов'язане із заданим ключем, досить викликати **gethash** з цим ключем і таблицею. За замовчуванням **gethash** повертає **nil**, якщо не знаходить шуканого елемента.

```
[5]> (gethash `color ht)
NIL ;
NIL
```

Тут ми вперше стикаємося з важливою особливістю Common Lisp: вираз може повертати декілька значень. Функція **`gethash`** повертає два. Перше значення асоційоване з ключем. Друге значення, якщо воно **`nil`** (як в нашому прикладі), означає, що шуканий елемент не був знайдений. Чому ми не можемо судити про це з першого **`nil`**? Справа в тому, що елементом, пов'язаним з ключем **`color`**, може виявитися **`nil`**, і **`gethash`** поверне його, але в цьому випадку в якості другого значення - **`t`**.

Більшість реалізацій виводить послідовно всі повернені значення, але якщо результат багатозначною функції використовується інший функцією, то їй передається лише перше значення.

Щоб зіставити нове значення якого-небудь ключу, використовуємо **setf** разом з **gethash**:

```
[6]> (setf (gethash `color ht) `red)
RED
```

Тепер **gethash** поверне знову встановлене значення:

```
[7]> (gethash `color ht)
RED ;
T
```

Друге значення підтверджує, що **gethash** повернув реально наявний в таблиці об'єкт, а не значення за замовчуванням.

Об'єкти, що зберігаються в хеш-таблицях, можуть мати будь-який тип. Наприклад, при бажанні зіставити кожної функції її короткий опис можна створити таблицю, в якій ключами будуть функції, а значеннями — рядки:

```
[8]>  (setf bugs (make-hash-table))
#S(HASH-TABLE :TEST FASTHASH-EQL)
[9]> (push "Doesn't take keyword
arguments. "
(gethash `our-member bugs))
("Doesn't take keyword arguments. ")
```

Так як за замовчуванням **gethash** повертає **nil**, виклик **push** еквівалентний **setf**, і ми просто кладемо нашу рядок на порожній список.

Хеш-таблиці також можна використовувати замість списків для подання множин. Вони істотно прискорюють пошук значень і їх видалення у випадках великих обсягів даних. Щоб додати елемент в множину, представлену у вигляді хеш-таблиці, використовуйте **`setf`** разом з **`gethash`**:

```
> (setf fruit (make-hash-table))
#<Hash-Table BFDE76>
> (setf (gethash `apricot fruit) t)
T
```

Перевірка на приналежність елемента множині виконується за допомогою **`gethash`**:

```
> (gethash `apricot fruit)
T
T
```

За замовчуванням **gethash** повертає **nil,** тому новостворена хеш-таблиця являє собою порожньою множиною. Щоб видалити елемент з множини, можна скористатися **remhash**:

```
> (Remhash 'apricot fruit)
T
```

Повертаючи **t**, **remhash** сигналізує, що шуканий елемент був знайдений і успішно видалений.

Для ітерації по хеш-таблиці існує **maphash**, якій необхідно передати функцію двох аргументів і саму таблицю. Ця функція буде викликана з кожної наявної в таблиці парою ключ-значення в довільному порядку.

```
[14]> (setf (gethash `shape ht) `spherical
(gethash `size ht) `giant)
GIANT
[20]> (maphash (lambda (k v)(format t "~A
= ~A~%" k v))
ht)
SIZE = GIANT
SHAPE = SPHERICAL
COLOR = RED
NIL
```

Функція **`maphash`** завжди повертає **`nil`**, однак ви можете зберегти дані, якщо передасте функцію, яка, наприклад, буде накопичувати результати в списку.

Наступна лекція буде присвячена завершенню розгляду особливостям реалізаціям на мові Common Lisp.