Інженерія знань

6 курс, осінь 2021

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

Лекція 4

Особливості мови LISP для створення систем подання знань

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)=x**^{3}. Яке значення куба ідентифікатора **куб** у визначенні **куб(x)=x**^{3}? Чи можна визначити цю функцію, не даючи їй імені?

λx.x^3 визначає функцію, яка відображає кожне **x** у домені до **x**^{3}. Можна сказати, що це визначення або *лямбда-абстракція*, **λx.x**^{3}, є значенням, прив'язаним до куба ідентифікатора. Ми говоримо, що **x**^{3} - 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:

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

необчіслювана.

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))
```

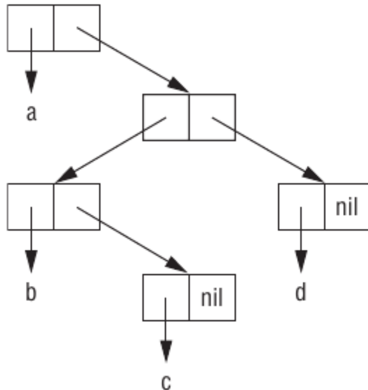
З останнього прикладу видно, як **mapcar** обробляє випадок зі списками різної довжини. Обчислення обривається після закінчення самого короткого списку.

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 безлічами, 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 `set-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))
```

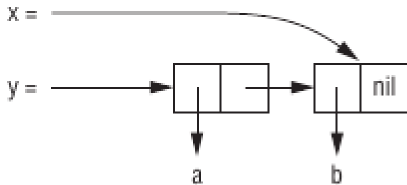
А виклик `(pop lst)` до

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

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.

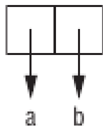
Списки, які можуть бути побудовані за допомогою **list**, називаються *правильними списками (proper list)*. Знову ж правильним списком вважається або **nil**, або **cons**-клітинка, **cdr** якої - також правильний список. Таким чином, можна визначити предикат, який повертає істину тільки для правильного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

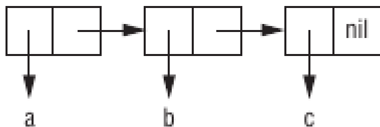
Оскільки ця **`cons`**-клітинка не є правильним списком, при відображенні її **`car`** і **`cdr`** розділяються крапкою. Такі клітинки називаються *точковими парами*.

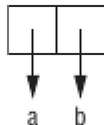


Правильні списки можна задавати і у вигляді набору точкових пар, але вони будуть відображатися у вигляді списків:

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

Зверніть увагу, як співвідносяться коміркова і точкова нотації

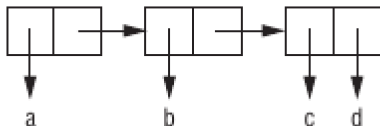




Допустима також змішана форма запису:

```
[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

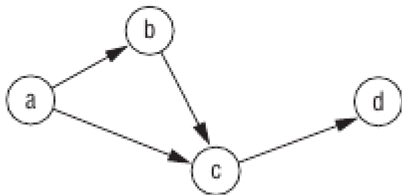
```
(defun shortest-path (start end net)
     (bfs end (list (list start)) net))
(defun bfs (end queue net)
 (if (null queue)
  nil
  (let ((path (car queue)))
   (let ((node (car path)))
    (if (eql node end)
     (reverse path)
       (bfs end
           (append (cdr queue)
              (new-paths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

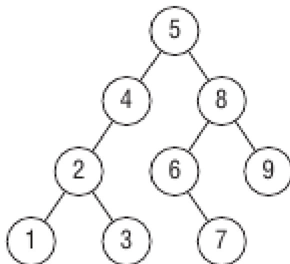
Функція **print-point** відображатиме структуру в такий скороченій формі:

```
[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-min**). 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))))))
```

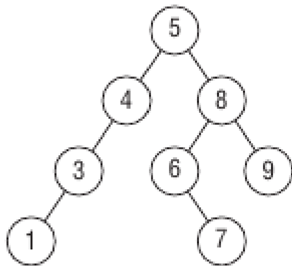
```
(defun bst-remove-min (bst)
(if (null (node-l bst))
(node-r bst)
(make-node :elt (node-elt bst)
:l (bst-remove-min (node-l bst))
:r (node-r bst))))

(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 іншій 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.

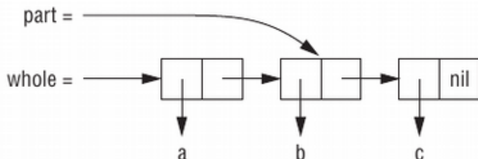
Функція, як і будь-який інший об'єкт, може повертатися як результат вираження. Нижче наведено приклад функції, яка повертає функцію, яка застосовується для поєднання об'єктів того ж типу, що і її аргумент:

```
(defun combiner (x)
(typecase x
(number ’+)
(list ’append)
(t  ’list)))
```

Списки можуть спільно використовувати одні й ті ж осередки. У найпростішому випадку один список може бути частиною іншого. Після виконання

```
[1]>  (setf part (list `b `c))
(B C)
[2]>  (setf whole (cons `a part))
(A B C)
```

перша осередок стає частиною (а точніше, **cdr**) другий. У подібних випадках прийнято говорити, що два списки розподілюють одну структуру. Структура, що лежить в основі двох таких списків, представлена нижче (розподільна структура)



Подібні ситуації виявляє предикат **tailp**. Він приймає два списки і повертає істину, якщо зустріне перший список при обході другого.

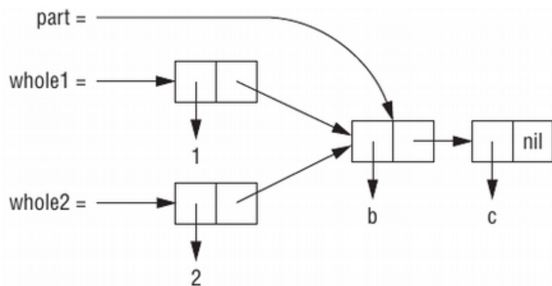
```
[3]>  (tailp part whole)
T
```

Ми можемо реалізувати його самостійно:

```
[4]> (defun our-tailp (x y)
(or (eql x y)
(and (consp y)
(our-tailp x (cdr y)))))
OUR-TAILP
```

Згідно з цим визначенням кожен список є хвостом самого себе, а **nil** є хвостом будь-якого правильного списку.

У більш складному випадку два списки можуть розділяти загальну структуру, навіть коли один з них не є хвостом іншого. Це відбувається, коли вони ділять загальний хвіст, як показано на малюнку нижче (розподілений хвіст).



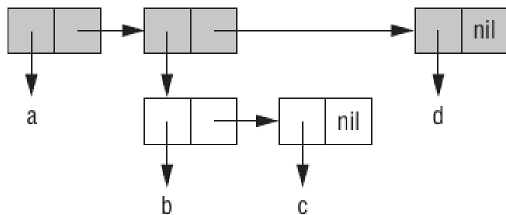
Створимо подібну ситуацію:

```
[6]> (setf part (list `b `c)
whole1 (cons 1 part)
whole2 (cons 2 part))
(2 B C)
```

Тепер **whole1** і **whole2** поділяють структуру, але один не є хвостом іншого.

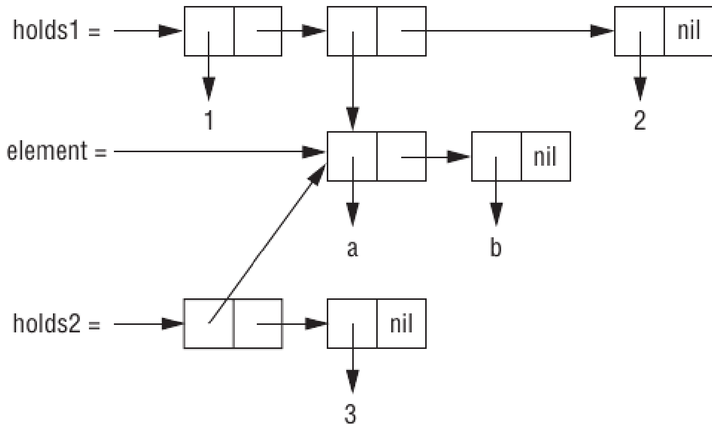
У разі вкладених списків важливо відрізняти списки з розділяється структурою від їх елементів з розділяється структурою. Структура списку верхнього рівня включає осередки, з яких складається сам список, але не включає будь-які осередки, з яких складаються окремі елементи списку.

Приклад структури верхнього рівня вкладеного списку наведено на малюнку нижче



Чи мають два осередки розділяється структуру, залежить від того, вважаємо ми їх списками або деревами. Два вкладених списку можуть розділяти одну структуру як дерева, але не розділяти її як списки. Наступний код створює ситуацію, зображену на малюнку нижче, де два списки містять один і той же елемент-список (розподільне піддерево):

```
[7]> (setf element (list `a `b)
holds1 (list 1 element 2)
holds2 (list element 3))
((A B) 3)
```



Хоча другий елемент **holds1** розділяє структуру з першим елементом **holds2** (в дійсності, він йому ідентичний), **holds1** і **holds2** не ділять між собою загальну структуру як списки. Два списку поділяють структуру як списки, тільки якщо вони ділять загальну структуру верхнього рівня, чого не роблять **holds1** і **holds2**.

Уникнути використання розділяється структури можна за допомогою копіювання. Функція **copy-list**, яка визначається як

```
[8]> (defun our-copy-list (lst)
(if (null lst)
nil
(cons (car lst) (our-copy-list (cdr lst)))))
OUR-COPY-LIST
```

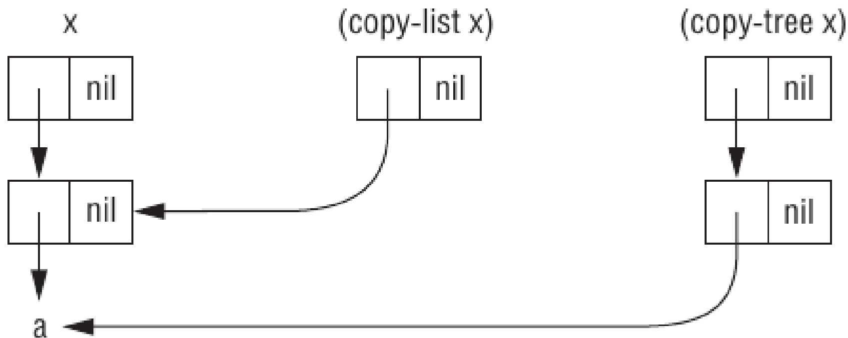
поверне список, чи не розділяє структуру верхнього рівня з вихідним списком.

Функція **copy-tree**, яка може бути визначена наступним чином:

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

поверне список, який не поділяє структуру всього дерева з вихідним списком.

Малюнок нижче демонструє різницю двох способів копіювання між викликом **copy-list** і **copy-tree** для вкладеного списку.



Чому варто уникати використання розділяється структури? До сих пір це явище розглядалося лише як забавна головоломка, а в написаних раніше програмах ми прекрасно обходилися і без нього. Колективна структура викликає проблеми, якщо об'єкти, які мають таку структуру, модифікуються. Справа в тому, що модифікація одного з двох списків із загальною структурою спричинить за собою ненавмисне зміна іншого.
Нагадаємо, як зробити один список хвостом іншого:

```
[11]> (setf whole (list `a `b `c)
tail (cdr whole))
(B C)
```

Зміна списку **tail** спричинить симетричне зміна хвоста **whole**, і навпаки, так як по суті це одна і та ж комірка:

```
[12]>  (setf (second tail) `e)
E
[13]> tail
(B E)
[14]> whole
(A B E)
```

Зрозуміло, те ж саме буде відбуватися і для двох списків, які мають загальний хвіст.

Зміна двох об'єктів одночасно не завжди є помилкою. Іноді це саме те, що потрібно. Однак якщо така зміна відбувається ненавмисно, воно може привести до некоректної роботи програми. Досвідчені програмісти вміють уникати подібних помилок і негайно розпізнавати такі ситуації. Якщо список без видимої причини змінює свій вміст, ймовірно, він має розділяється структуру. Небезпечна не сама колективна структура, а можливість її зміни.
Щоб гарантувати відсутність подібних помилок, просто уникайте використання **`setf`** (а також аналогічних операторів типу **`pop`**, **`rplaca`** та інших) для списків. Якщо змінність списків все ж потрібно, то необхідно з'ясувати, звідки взявся змінюваний список, щоб переконатися, що він не поділяє структуру з чимось, що не можна міняти. Якщо ж це не так або вам невідомо походження списку, то модифікувати необхідно не сам список, а його копію.

Потрібно бути подвійно обережним при використанні функцій, написаних кимось іншим. Поки не встановлено протилежне, майте на увазі, що все, що передається функції:

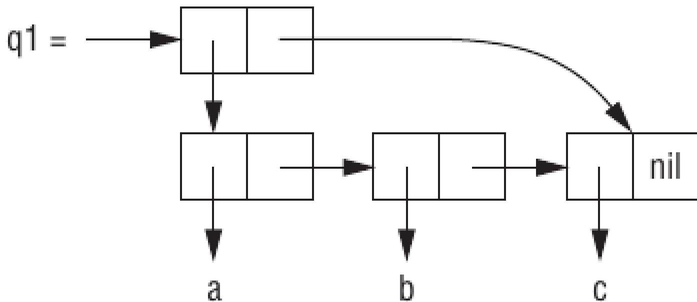
1. Чи може бути передано деструктивним операторам.
2. Може бути збережено де-небудь, і зміна цього об'єкта призведе до зміни в інших частинах коду, що використовує даний об'єкт.

В обох випадках правильним рішенням є копіювання аргументів.

В Common Lisp виклик будь-якої функції, що виконується під час проходу по структурі (наприклад, функції-аргументу **`mapcar`** або **`remove-if`**), не повинен змінювати цю структуру. В іншому випадку наслідки виконання такого коду не визначені.

Розглянемо приклад програмування черги.
Спільні структури - це не тільки привід для занепокоєння. Іноді вони можуть бути корисні. Зараз покажемо, як за допомогою поділюваних структур уявити черги. Черга - це сховище об'єктів, з якого вони можуть бути вилучені по одному в тому ж порядку, в якому вони були туди записані. Таку модель прийнято називати FIFO - скорочення від *«першим прийшов, першим пішов» («first in, first out»)*.
За допомогою списків легко уявити стопку, так як додавання і отримання елементів відбувається з одного кінця. Завдання уявлення черзі більш складна, оскільки додавання та вилучення об'єктів відбувається з різних кінців. Для ефективної її реалізації необхідно якимось чином забезпечити управління обома кінцями списку.

Одна з можливих стратегій наводиться на малюнку нижче, де показана чергу з трьох елементів: **a**, **b** і **c**. Чергою вважаємо точкову пару, що складається зі списку і останньої клітинки цього ж списку. Будемо називати їх початок і кінець. Щоб отримати елемент з черги, необхідно просто витягти початок. Щоб додати новий елемент, необхідно створити нову комірку, зробити її **cdr** кінця черги і потім зробити її ж кінцем.



Таку стратегію реалізує код:

```
[15]> (defun make-queue () (cons nil nil))
MAKE-QUEUE
[16]> (defun enqueue (obj q)
(if (null (car q))
(setf (cdr q) (setf (car q) (list obj)))
(setf (cdr (cdr q)) (list obj)
(cdr q) (cdr (cdr q))))
(car q))
ENQUEUE
[17]> (defun dequeue (q)
(pop (car q)))
DEQUEUE
```

Вона використовується наступним чином:

```
[18]>  (setf q1 (make-queue))
(NIL)
[19]>  (progn (enqueue `a q1)
(enqueue `b q1)
(enqueue `c q1))
(A B C)
```

Тепер **q1** представляє чергу, зображену на малюнку вище

```
[20]> q1
((A B C) C)
```

Спробуємо забрати з черги кілька елементів:

```
[21]>  (dequeue q1)
A
[22]>  (dequeue q1)
B
[23]>  (enqueue `d q1)
(C D)
```

Common Lisp включає в себе кілька функцій, які можуть змінювати структуру списків і за рахунок цього працювати швидше. Вони називаються деструктивними. І хоча ці функції можуть змінювати осередки, передані їм в якості аргументів, вони роблять це не заради побічних ефектів.
Наприклад, **delete** є деструктивним аналогом **remove**. Хоча їй і дозволено псувати переданий список, вона не дає ніяких обіцянок, що так і буде робити. Подивимося, що відбувається в більшості реалізацій:

```
[24]> (setf lst `(a b r a c a d a b r a))
(A B R A C A D A B R A)
[25]> (delete `a lst)
(B R C D B R)
[26]> lst
(A B R C D B R)
```

Як і у випадку з **remove**, щоб зафіксувати побічний ефект, необхідно використовувати **setf**:

```
[27]> (setf lst (delete `a lst))
(B R C D B R)
```

Прикладом того, як деструктивні функції модифікують списки, є **nconc**, деструктивна версія **append**. Наведемо її версію для двох аргументів, що демонструє, яким чином зшиваються списки:

```
[28]> (defun nconc2 (x y)
(if (consp x)
(progn
(setf (cdr (last x)) y)
x)
y))
NCONC2
```

cdr останньої клітинки першого списку стає дороговказом на другий список.

Функція **mapcan** схожа на **mapcar**, але з'єднує в один список повернені значення (які повинні бути списками) за допомогою **nconc**:

```
[29]> (mapcan `list `(a b c) `(1 2 3 4))
(A 1 B 2 C 3)
```

Ця функція може бути визначена наступним чином:

```
[30]> (defun our-mapcan (fn &rest lsts)
(apply `ncnc (apply `mapcar fn lsts)))
OUR-MAPCAN
```

Використовуйте **mapcan** з обережністю, враховуючи її деструктивний характер. Вона з'єднує повертаються списки за допомогою **nconc**, тому їх краще більше ніде не задіяти. Функція **mapcan** корисна, зокрема, в задачах, інтерпретованих як збір всіх вузлів одного рівня якогось дерева. Наприклад, якщо **children** повертає список чиїхось дітей, тоді ми зможемо визначити функцію для отримання списку онуків так:

```
[31]> (defun grandchildren (x)
(mapcan `(lambda (c)
(copy-list (children c)))
(children x)))
GRANDCHILDREN
```

Ця функція застосовує **copy-list** до результату виклику **children**, так як він може повертати вже існуючий об'єкт, а не виробляти новий.

Також можна визначити недеструктивний варіант mapcan:

```
[32]> (defun mappend (fn &rest lsts)
(apply `append (apply `mapcar fn lsts)))
MAPPEND
```

Використовуючи **mappend**, ми можемо обійтися без викликів **copy-list** у визначенні **grandchildren**:

```
[33]> (defun grandchildren (x)
(mappend `children (children x)))
GRANDCHILDREN
```

У деяких ситуаціях доречніше використовувати деструктивні операції, ніж Недеструктивні. У попередніх лекціях було показано, як управляти двійковими деревами пошуку (BST). Всі використані там функції були Недеструктивні, але якщо потрібно застосувати BST на практиці, така обережність зайва. Нижче приведена деструктивна версія `bst-insert`. Вона приймає точно такі ж аргументи і повертає точно таке ж значення, як і вихідна версія. Єдиною відмінністю є те, що вона може змінювати дерево, яке передається другим аргументом.

Двійкові дерева пошуку: деструктивна вставка

```
[34]> (defun bst-insert! (obj bst <)
(if (null bst)
(make-node :elt obj)
(progn (bsti obj bst <)
bst)))
BST-INSERT!
```

```
[35]> (defun bsti (obj bst <)
(let ((elt (node-elt bst)))
(if (eql obj elt)
bst
(if (funcall < obj elt)
(let ((l (node-l bst)))
(if l
(bsti obj l <)
(setf (node-l bst)
(make-node :elt obj))))
(let ((r (node-r bst)))
(if r
(bsti obj r <)
(setf (node-r bst)
(make-node :elt obj))))))))
BSTI
```

Трохи раніше ми попереджували про те, що деструктивні функції викликаються не заради побічних ефектів. Тому якщо ви хочете побудувати дерево за допомогою **bst-insert!**, вам потрібно викликати її так само, як якщо б ви викликали справжню **bst-insert**:

```
> (setf *bst* nil)
NIL
> (dolist (x '(7 2 9 8 4 1 5 12))
(setf *bst* (bst-insert! x *bst* '<)))
NIL
```

Нижче представлений деструктивний варіант функції **bst-delete**, яка пов'язана з **bst-remove** так само, як **delete** пов'язана з **remove**.
Як і **delete**, вона не має на увазі виклик заради побічних ефектів. Використовувати **bst-delete** необхідно так само, як і **bst-remove**:

```
> (setf *bst* (bst-delete 2 *bst* '<))
#<7>
> (bst-find 2 *bst* '<)
NIL
```

Двійкові дерева пошуку: деструктивне видалення

```
[41]> (defun bst-delete (obj bst <)
(if (null bst)
nil
(if (eql obj (node-elt bst))
(del-root bst)
(progn
(if (funcall < obj (node-elt bst))
(setf (node-l bst) (bst-delete obj (node-l
bst) <))
(setf (node-r bst) (bst-delete obj (node-r
bst) <)))
bst))))
BST-DELETE
```

```
[42]> (defun del-root (bst)
(let ((l (node-l bst)) (r (node-r bst)))
(cond ((null l) r)
((null r) l)
(t
 (if (zerop (random 2))
(cutnext r bst nil)
(cutprev l bst nil))))))
DEL-ROOT
```

```
[43]> (defun cutnext (bst root prev)
(if (node-l bst)
(cutnext (node-l bst) root bst)
(if prev
(progn
(setf (node-elt root) (node-elt bst)
(node-l prev) (node-r bst))
root)
(progn
(setf (node-l bst)
 (node-l root))
bst))))
CUTNEXT
```

```
[44]> (defun cutprev (bst root prev)
(if (node-r bst)
(cutprev (node-r bst) root bst)
(if prev
(progn
(setf (node-elt root) (node-elt bst)
(node-r prev)
root)
(progn
(setf (node-r bst)
bst))))
(node-l bst))
(node-r root))
CUTPREV
```

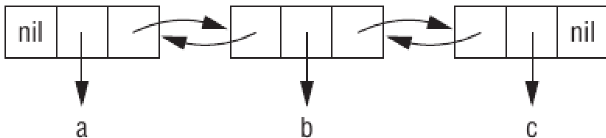
```
[45]> (defun replace-node (old new)
(setf (node-elt old) (node-elt new)
(node-l old) (node-l new)
(node-r old) (node-r new)))
REPLACE-NODE
```

```
[46]> (defun cutmin (bst par dir)
(if (node-l bst)
(cutmin (node-l bst) bst :l)
(progn
(set-par par dir (node-r bst))
(node-elt bst))))
CUTMIN
[47]> (defun cutmax (bst par dir)
(if (node-r bst)
(cutmax (node-r bst) bst :r)
(progn
(set-par par dir (node-l bst))
(node-elt bst))))
CUTMAX
```

```
[48]> (defun set-par (par dir val)
(case dir
(:l (setf (node-l par) val))
(:r (setf (node-r par) val))))
SET-PAR
```

Звичайні списки в Ліспі є однозв'язного. Це означає, що рух за вказівниками відбувається тільки в одному напрямку: ви можете перейти до наступного елементу, але не можете повернутися до попереднього. Двусвязного списки мають також і зворотний покажчик, з цього можна переміщатися в обидва боки. Далі ми покажемо, як створювати і використовувати двусвязного списки.

На малюнку нижче показана їх можлива реалізація. **cons**-комірки мають два поля: **car**, який вказує на дані, і **cdr**, який вказує на наступний елемент. Елемент двусвязного списку повинен мати ще одне поле, яке вказує на попередній елемент. Виклик **defstruct** створює об'єкт з трьох частин, названий **dl** (від «doubly linked»), який ми будемо використовувати для створення двусвязного списків. Поле **data** в **dl** відповідає **car** в **cons**-комірці, а поле **next** відповідає **cdr**. Поле **prev** схоже на **cdr**, але вказує в зворотному напрямку. Порожньому двусвязного списку, як і звичайного, відповідає **nil**.



Виклик **defstruct** також визначає функції для двусвязного списків, аналогічні **car**, **cdr** і **consp**: **dl-data**, **dl-next** і **dl-p**. Функція друку **dl-> list** повертає звичайний список з тими ж значеннями, що і двусвязний.
Функція **dl-insert** схожа на **cons**. По крайней мере, вона, як і **cons**, є основною функцією-конструктором. На відміну від **cons**, вона змінює двусвязний список, переданий другим аргументом. У даній ситуації це абсолютно нормально. Щоб помістити новий об'єкт в початок звичайного списку, вам не потрібно його змінювати, однак щоб помістити об'єкт в початок двусвязного списку, необхідно присвоїти полю **prev** покажчик на новий об'єкт.

```
[56]> (defstruct (dl (:print-function print-
dl))
prev data next)
DL
```

```
[57]> (defun print-dl (dl stream depth)
(declare (ignore depth))
(format stream "#<DL ~A>" (dl->list dl)))
PRINT-DL
```

```
[58]> (defun dl->list (lst)
(if (dl-p lst)
(cons (dl-data lst) (dl->list (dl-next lst)))
lst))
DL->LIST
```

```
[59]> (defun dl-insert (x lst)
(let ((elt (make-dl :data x :next lst)))
(when (dl-p lst)
(if (dl-prev lst)
(setf (dl-next (dl-prev lst)) elt
(dl-prev elt) (dl-prev lst)))
(setf (dl-prev lst) elt))
elt))
DL-INSERT
```

```
[60]> (defun dl-remove (lst)
(if (dl-prev lst)
(setf (dl-next (dl-prev lst)) (dl-next lst)))
(if (dl-next lst)
(setf (dl-prev (dl-next lst)) (dl-prev lst)))
(dl-next lst))
DL-REMOVE

[61]> (defun dl-list (&rest args)
(reduce `dl-insert args
:from-end t :initial-value nil))
DL-LIST
```

Іншими словами, кілька звичайних списків можуть мати загальний хвіст. Але для пари двусвязного списків це неможливо, так як хвіст кожного з них має різні покажчики на голову. Якби функція **dl-insert** була деструктивна, їй би доводилося завжди копіювати свій другий аргумент.
Інше цікаве відмінність між одно- і двусвязного списками за лягає у способі доступу до їхніх елементів. Працюючи з однозв'язного списком, ви зберігаєте покажчик на його початок. При роботі з двусвязного списком, оскільки в ньому елементи з'єднані з обох кінців, ви можете використовувати покажчик на будь-який з елементів. Тому **dl-insert**, на відміну від cons, може додавати новий елемент в будь-яке місце двусвязного списку, а не тільки в початок.

Функція **dl-list** є **dl**-аналогом **list**. Вона отримує будь-яку кількість аргументів і повертає складається з них **dl**:

```
> (dl-list 'a 'b 'c)
#<DL (A B C)>
```

У ній використовується **reduce** з параметрами: **:from-end**, встановленим в **t**, і **:initial-value**, встановленим в **nil**, що робить наведений вище виклик еквівалентним наступній послідовності:

```
(dl-insert 'a (dl-insert 'b (dl-insert 'c nil)))
```

Замінивши 'dl-insert на 'cons у визначенні dl-list, ця функція буде вести себе аналогічно list:

```
> (setf dl (dl-list 'a 'b))
#<DL (A B)>
> (setf dl (dl-insert 'c dl))
#<DL (C A B)>
> (dl-insert 'r (dl-next dl))
#<DL (R A B)>
> dl
#<DL (C R A B)>
```

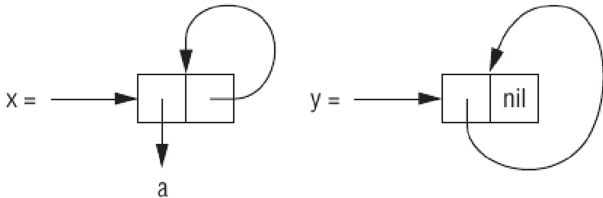
Нарешті, для видалення елемента з двусвязного списку визначена dl-remove. Як і dl-insert, вона зроблена деструктивною.

Змінюючи структуру списків, можна створювати циклічні списки. Вони бувають двох видів. Найбільш корисними є ті, які мають замкнуту структуру верхнього рівня. Такі списки називаються циклічними по хвосту (**cdr**-circular), так як цикл створюється **cdr**-частинами комірок.

Щоб створити такий список, що містить один елемент, необхідно встановити покажчик **cdr** на самого себе:

```
> (setf x (list 'a))
(A)
> (progn (setf (cdr x) x) nil)
NIL
```

Тепер **x** - циклічний список. Його структура зображена на малюнку нижче.



При спробі надрукувати такий список символ **a** буде виводитися до нескінченності. Цього можна уникнути, встановивши значення *** print-circle *** в **t**:

```
> (setf * print-circle * t)
T
> x
# 1 = (A. # 1 #)
```

Списки з циклічним хвостом можуть бути корисні для подання, наприклад, буферів або обмежених наборів якихось об'єктів (пулів). Пул - це набір ініціалізованих ресурсів, які підтримуються в готовому до використання стані, а не виділяються на вимогу.
Наступна функція перетворить довільний нециклический непорожній список в циклічний з тими ж елементами:

```
[62]> (defun circular (lst)
(setf (cdr (last lst)) lst))
CIRCULAR
```

Інший тип циклічних списків - циклічні по голові (**car-circular**). Список такого типу можна розуміти як дерево, що є піддерево самого себе. Його назва обумовлена тим, що в ньому міститься цикл, замкнутий на car осередки. Нижче ми створимо циклічний по голові список, другий елемент якого є він сам:

```
> (let ((y (list 'a)))
(setf (car y) y)
y)
#1=(#1#)
```

Результат зображений був раніше. Незважаючи на циклічність, цей циклічний по голові список (**car-circular**) як і раніше є правильним списком, на відміну від циклічних по хвосту (**cdr-circular**), які правильними бути не можуть.

Список може бути циклічним по голові і хвоста одночасно. `car` і `cdr` такої комірки вказуватимуть на неї саму:

```
> (let ((c (cons 1 1)))
(setf (car c) c
(cdr c) c)
c)
#1=(#1# . #1#)
```

Складно уявити, для чого можуть використовуватися подібні об'єкти. Насправді, головне, що потрібно винести з цього, - необхідно уникати ненавмисного створення циклічних списків, так як більшість функцій, які працюють зі списками, будуть йти в нескінченний цикл, якщо отримають в якості аргументу список, циклічний по тому напрямку, по якому вони здійснюють прохід.

Циклічна структура може бути проблемою не тільки для списків, але і для інших типів об'єктів, наприклад для масивів:

```
> (setf *print-array* t)
T
> (let ((a (make-array 1)))
(setf (aref a 0) a)
a)
#1=#(#1#)
```

І дійсно, практично будь-який об'єкт, що складається з елементів, може включати себе в якості одного з них. Зрозуміло, структури, створювані **defstruct**, також можуть бути циклічними. Наприклад, структура **c**, що представляє елемент дерева, може мати поле parent, що містить іншу структуру **p**, чиє поле **child** посилається назад на **c**:

```
> (progn (defstruct elt
(parent nil) (child nil))
(let ((c (make-elt))
(p (make-elt)))
(setf (elt-parent c) p
(elt-child p) c)
c))
#1=#S(ELT PARENT #S(ELT PARENT NIL CHILD #1#)
CHILD NIL)
```

Ми з вами переконалися, що на Ліспі можна створювати дуже складні структури та визначати свої функції для цих структур. Саме ці властивості і були використані багатьма створювачами мов представлення знань.

Наступна лекція буде присвячена системі подання знань у вигляді фреймів FRL.