Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

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

Лекція 5

Мова штучного інтелекту

LISP (завершення)

В цій лекції ми розглянемо ще ряд особливостей програмування на мові Лісп:
1) розподільні структури
2) цикличні структури

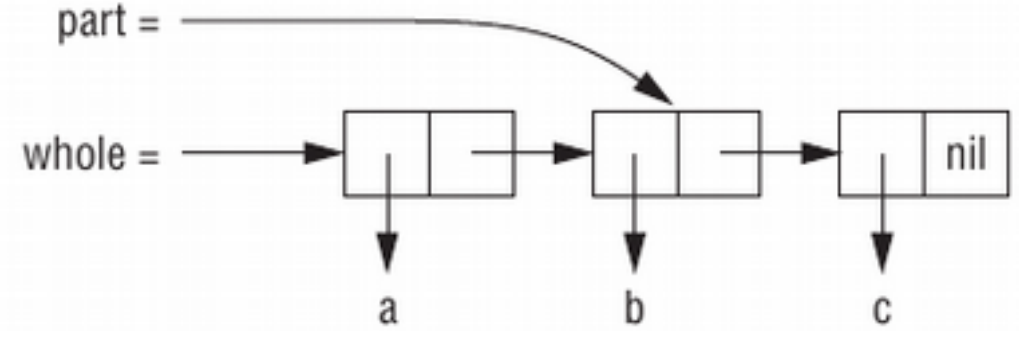
Функція, як і будь-який інший об'єкт, може повертатися як результат вираження. Нижче наведено приклад функції, яка повертає функцію, яка застосовується для поєднання об'єктів того ж типу, що і її аргумент:

```
(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.

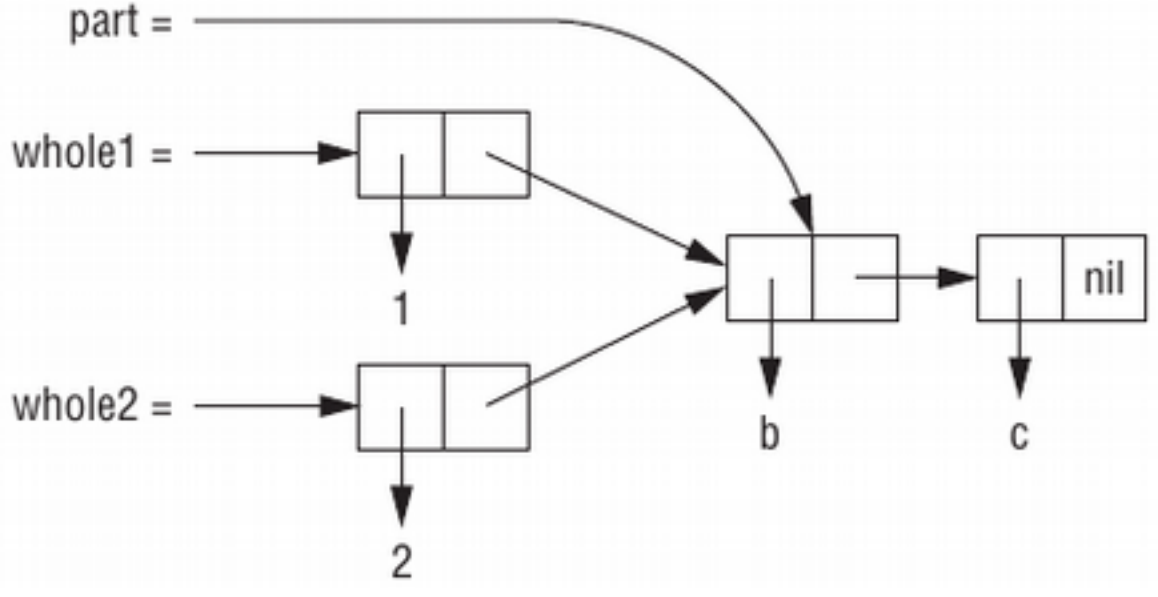
```
[3]>  (tailp part whole)
T
```

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).



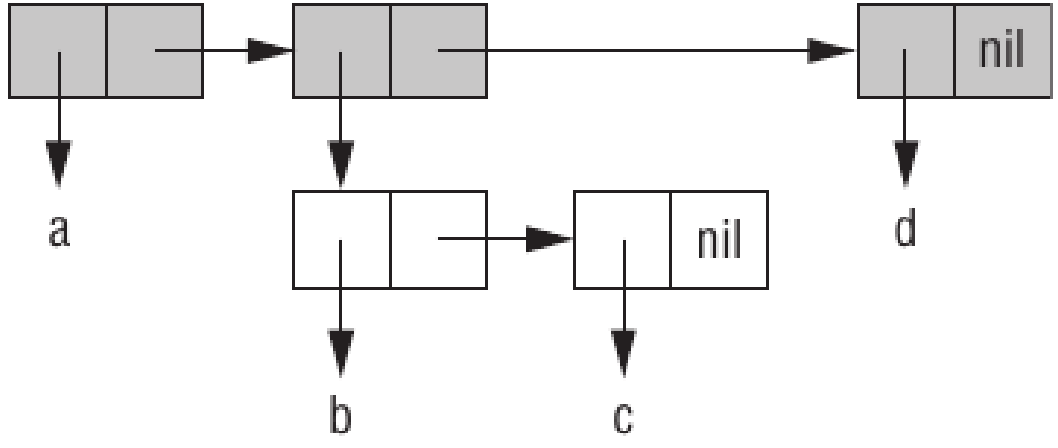
Створимо подібну ситуацію:

```
[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.

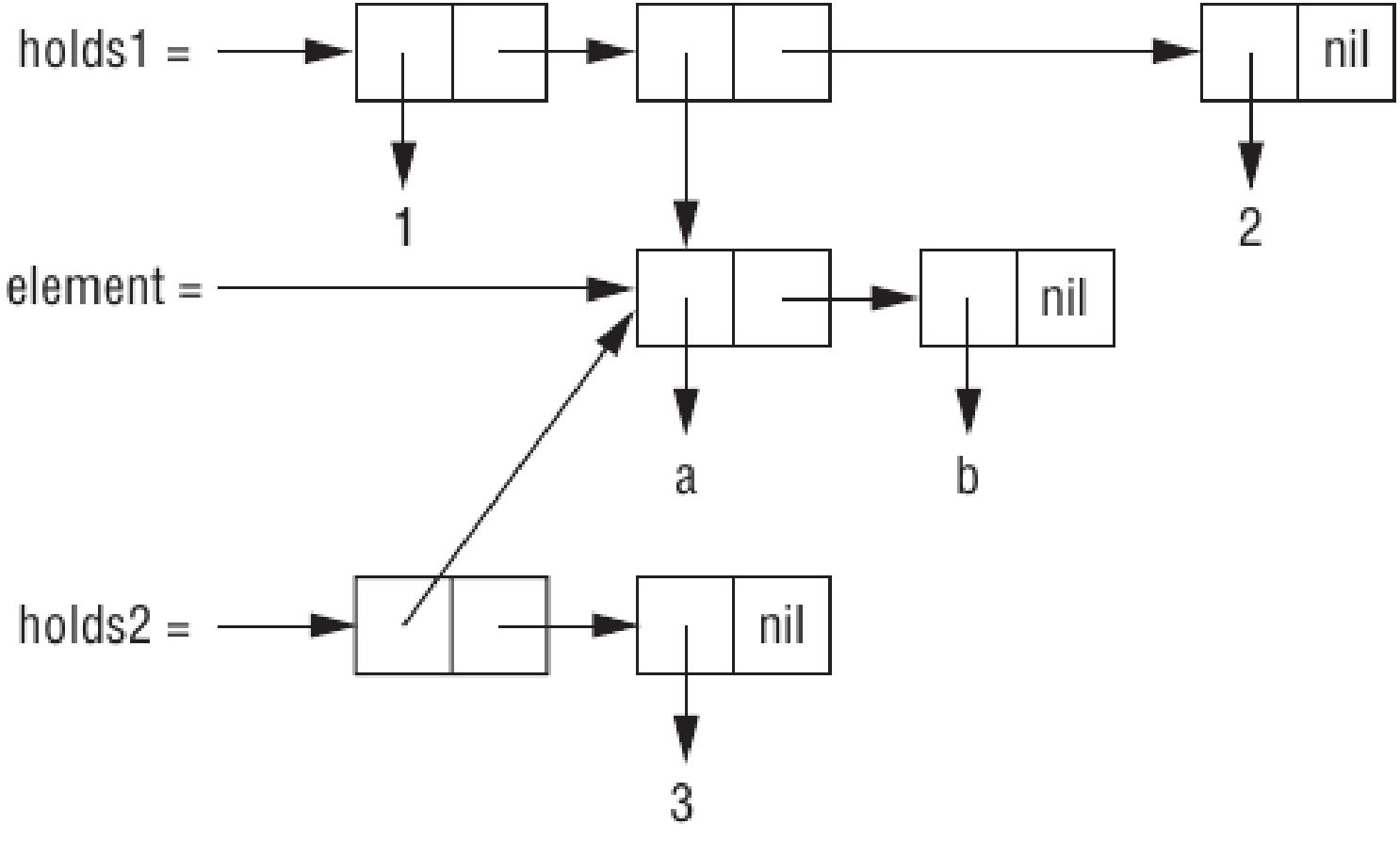
У разі вкладених списків важливо відрізняти списки з розділяється структурою від їх елементів з розділяється структурою. Структура списку верхнього рівня включає осередки, з яких складається сам список, але не включає будь-які осередки, з яких складаються окремі елементи списку.

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
```

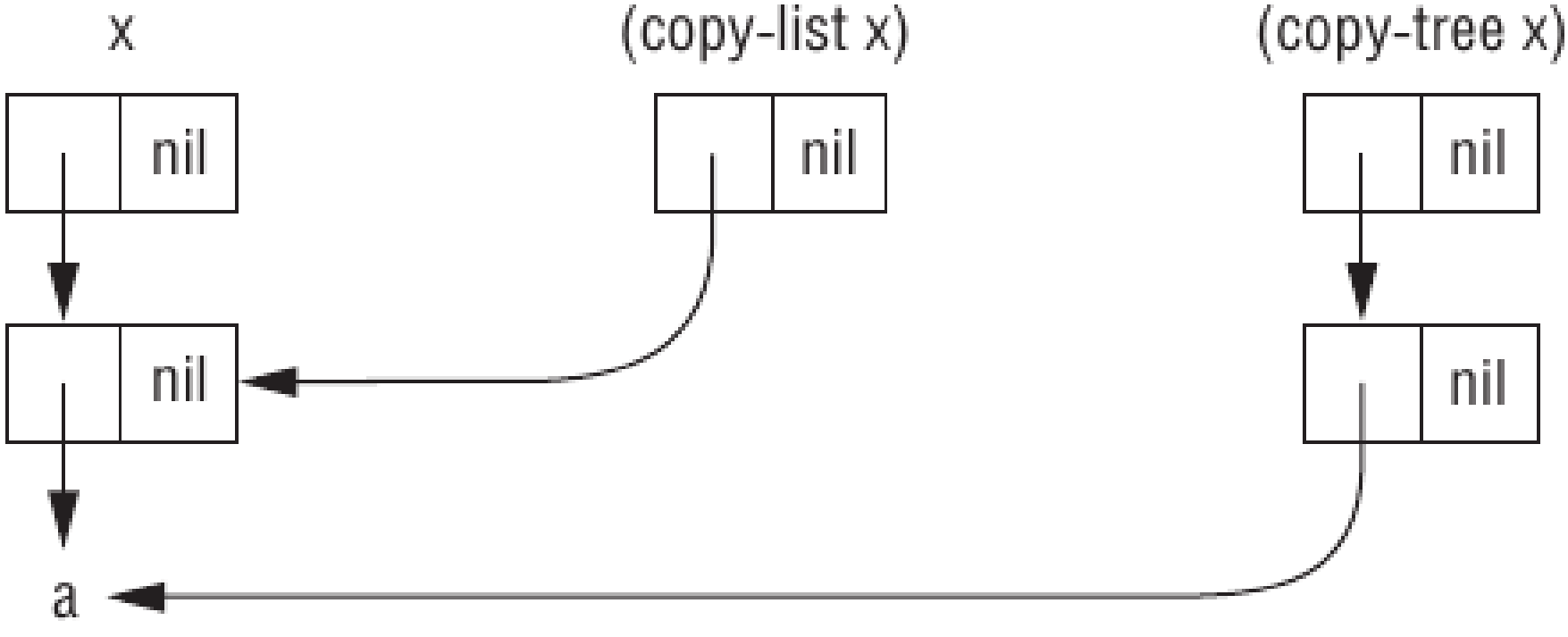
поверне список, чи не розділяє структуру верхнього рівня з вихідним списком.

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)
```

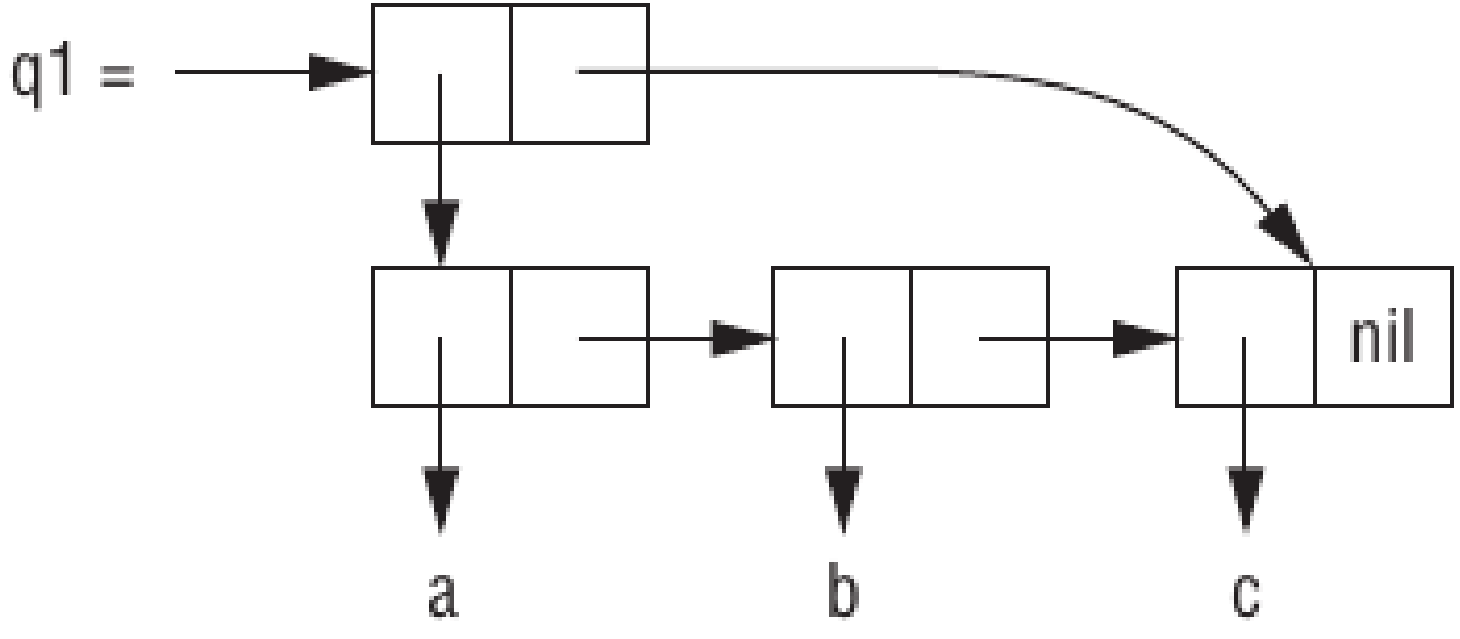
Зрозуміло, те ж саме буде відбуватися і для двох списків, які мають загальний хвіст.

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
```

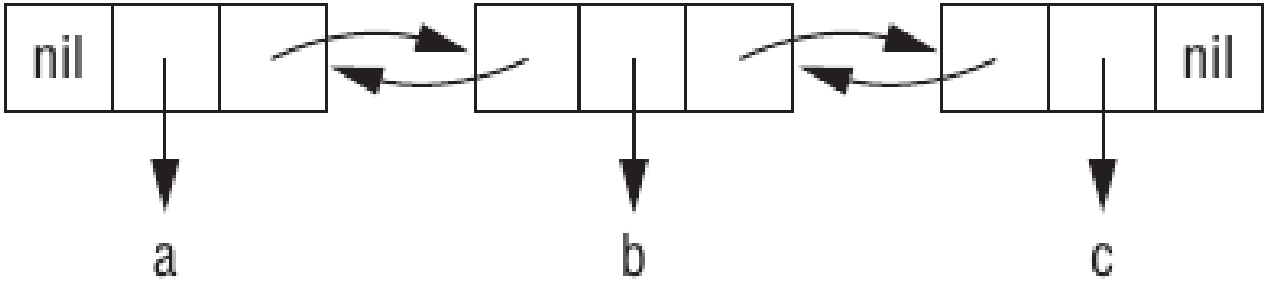
```
[45]> (defun replace-node (old new)
(setf (node-elt old) (node-elt new)
(node-l old) (node-l new)
(node-r old) (node-r new)))
REPLACE-NODE
```

```
[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)>
```

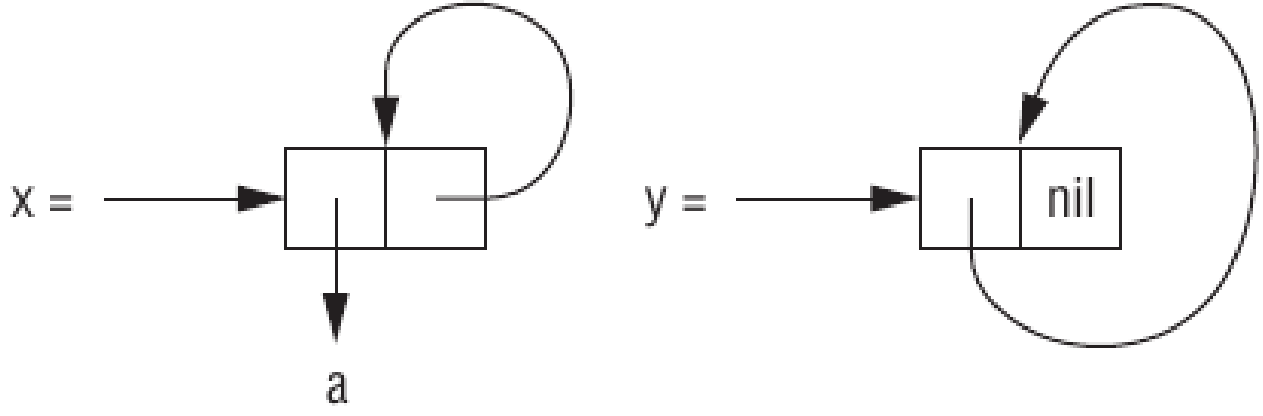
Нарешті, для видалення елемента з двусвязного списку визначена **dl-remove**. Як і **dl-insert**, вона зроблена деструктивною.

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 завершенню розгляду системи подання знань на основі FRL.