Common LISP
Biblioteka Standardowa

Jakub Kowalski

Instytut Informatyki, Uniwersytet Wrocławski

10 listopada 2010

Wprowadzenie

Biblioteka standardowa jest oczywiście zbyt rozległa aby opowiedzieć o niej szczegółowo w 45 minut.

Przedstawię więc materiał który pokaże czego można się spodziewać po jej zawartości.

A mianowicie opowiem o:

- konwencjach nazw i parametrów,
- funkcjach charakterystycznych dla sekwencji,
- różnych sposobach wykorzystania list,
- napisach i konwersji związanej z napisami,
- metodach przypisywania.

Konwencje

Funkcje destrukcyjne *(n...)*

Alternatywne wersje funkcji (zwyczajowo (ale nie zawsze, np. `remove-delete!`) o nazwie poprzedzonej literą *n*), zwracające tą samą wartość, ale dodakowo modyfikujące pamięć – tzn. zazwyczaj nadpisujące miejsca w pamięci zajmowane przez argumenty.

```
(defparameter *xs* (list 0 'a 2 'a 0))
```

```
> (subst 1 'a *xs*)
(0 1 2 1 0)
>*xs*
```

```
> (nsubst 1 'a *xs*)
(0 1 2 1 0)
>*xs*
```

Funkcje destrukcyjne *(n...)*

Alternatywne wersje funkcji (zwyczajowo (ale nie zawsze, np. `remove-delete!`) o nazwie poprzedzonej literą *n*), zwracające tą samą wartość, ale dodakowo modyfikujące pamięć – tzn. zazwyczaj nadpisujące miejsca w pamięci zajmowane przez argumenty.

```
(defparameter *xs* (list 0 'a 2 'a 0))
```

```
> (subst 1 'a *xs*)
(0 1 2 1 0)
>*xs*
(0 A 2 A 0)
```

```
> (nsubst 1 'a *xs*)
(0 1 2 1 0)
>*xs*
(0 1 2 1 0)
```

Funkcje destrukcyjne *(n...)*

Alternatywne wersje funkcji (zwyczajowo (ale nie zawsze, np. `remove-delete!`) o nazwie poprzedzonej literą *n*), zwracające tą samą wartość, ale dodakowo modyfikujące pamięć – tzn. zazwyczaj nadpisujące miejsca w pamięci zajmowane przez argumenty.

```
(defparameter *xs* (list 0 'a 2 'a 0))
```

```
> (subst 1 'a *xs*)
(0 1 2 1 0)
>*xs*
(0 A 2 A 0)
```

```
> (nsubst 1 'a *xs*)
(0 1 2 1 0)
>*xs*
(0 1 2 1 0)
```

Tak więc `nsubst` działa jak

```
(setf *xs*
  (subst 1 'a *xs*))
```

Funkcje destrukcyjne a współdzielenie

Aby bezpiecznie wykorzystywać funkcje destrukcyjne należy wiedzieć w jaki sposób działają (współdzielą pamięć) inne funkcje korzystające z tych samych argumentów.

```
(defparameter *xs* (list 0 1 2))
(defparameter *ys* (list 3 4))
(defparameter *xys* (append *xs* *ys*))
```

```
> *xys*
(0 1 2 3 4)
```

Funkcje destrukcyjne a współdzielenie

Aby bezpiecznie wykorzystywać funkcje destrukcyjne należy wiedzieć w jaki sposób działają (współdzielą pamięć) inne funkcje korzystające z tych samych argumentów.

```
(defparameter *xs* (list 0 1 2))
(defparameter *ys* (list 3 4))
(defparameter *xys* (append *xs* *ys*))
```

```
> *xys*
(0 1 2 3 4)
> (setf (first *xs*) 777) (setf (first *ys*) 666)
> *xys*
```

Funkcje destrukcyjne a współdzielenie

Aby bezpiecznie wykorzystywać funkcje destrukcyjne należy wiedzieć w jaki sposób działają (współdzielą pamięć) inne funkcje korzystające z tych samych argumentów.

```
(defparameter *xs* (list 0 1 2))
(defparameter *ys* (list 3 4))
(defparameter *xys* (append *xs* *ys*))
```

```
> *xys*
(0 1 2 3 4)
> (setf (first *xs*) 777) (setf (first *ys*) 666)
> *xys*
(0 1 2 666 4)
```

Predykaty *(...p)*

Mianem predykatów określa się funkcje których interpretacją jest potwierdzenie lub zaprzeczenie pewnej tezie. Zazwyczaj nazwy predykatów powstają z dołącznia do jakiegoś kluczowego wyrazu litery *p*.

```
;; Przynależność typowa
listp symbolp floatp integerp numberp complexp
realp characterp arrayp vectorp ...
;; Własności arytmetyczne
evenp minusp oddp plusp zerop ...
;; Własności znaków
alpha-char-p alphanumericp lower-case-p ...
;; Własności list
consp endp tailp ...
;; Inne
(subtypep typ1 typ2) ; typ1 jest podtypem typu2
(typep obiekt typ) ; obiekt jest typu typ
(equalp obiekt1 obiekt2) ; oba obiekty są równe
(boundp symbol) ; jest nazwą specjalnej zmiennej
```

Operatory porównań (...< .../ = ...)

Nazwy funkcji porównujących są złożone z nazwy typu (puste dla porównania liczb) oraz symbolu porównania.

```
;; Dla liczb
= /= > < <= >=

;; Dla znaków
char= char/= char> char< char<= char>=

;; Dla napisów
string= string/= string> string< string<= string>=
```

Operatory porównań a predykaty

Dla znaków oraz napisów istnieją predykaty odpowiadające operatorom porównania, lecz ignorujące różnice w wielkościach liter.

```
char-greaterp      ≡  >
char-lessp         ≡  <
char-not-greaterp  ≡  <=
char-not-equalp    ≡  /=
...

string-equal       ≡  =
string-greaterp    ≡  >
string-lessp       ≡  <
string-not-lessp   ≡  >=
...
```

Warunkowe wersje funkcji *(...-if ...-if-not)*

Funkcje które działają na sekwencjach sprawdzając równość elementów w większości posiadają swoje alternatywne wersje, kierujące się spełnieniem lub nie spełnieniem danego predykatu.

```
(defparameter *xs* (list 1 2 3 4 1 2))
```

```
>(remove 2 *xs*)
(1 3 4 1)
> (remove-if #'oddp *xs*)
(2 4 2)
> (remove-if-not #'(lambda (x) (> x 2)) *xs*)
(3 4)
```

Argumenty nazwane *(:key)*

Argumentem jest funkcja pobierająca z obiektu dane do porównania. Domyślnie przyjmuje wartość `nil` (działa wtedy jak identyczność).

```
(defparameter *xss* '((1 1) (2 1) (1 2) (2 2)))
```

```
> (remove 1 *xss* :key #'second)
((1 2) (2 2))
```

Powyższa konstrukcja jest równoważna

```
> (remove-if #'(lambda (xs) (= 1 (second xs)))
             *xss*)
```

Argumenty nazwane *(:from-end)*

Jeśli argument jest różny od `nil` to funkcja działa od końca sekwencji. Domyślnie `nil`.

```
(defparameter *xs* '(1 2 3 4 3 2 1))
```

```
> (remove-duplicates *xs*)
(4 3 2 1)
> (remove-duplicates *xs* :from-end T)
(1 2 3 4)
```

Argumenty nazwane *(:start :end)*

Argumenty pobierają indeksy wyznaczające fragment sekwencji dla którego funkcja ma zostać wywołana.
Domyślnie `start` wynosi 0, zaś `end` `nil` - które oznacza indeks za ostatnim elementem.

```
(defparameter *xs* '(1 1 2 3 1 5 6 1 1))
```

```
> (remove 1 *xs*)
(2 3 5 6)
> (remove 1 *xs* :start 2 :end 7)
(1 1 2 3 5 6 1 1)
```

Argumenty nazwane *(:test :test-not)*

Argumenty te pozwalają na zmianę funkcji używanej do porównywania elementów (domyślnie jest to `eql` dla *test*).

```
(defparameter *xss* '((1 1) (2 2) (1 1) (3 3)))
```

```
> (remove '(1 1) *xss*)
((1 1) (2 2) (1 1) (3 3))
> (remove '(1 1) *xss* :test #'equal)
((2 2) (3 3))
> (remove '(1 1) *xss* :test-not #'equal)
((1 1) (1 1))
```

Równoważnie można zapisać te funkcje jako odpowiednio:

```
(remove-if #'(lambda (xs) (equal xs '(1 1))) *xss*)
(remove-if #'(lambda (xs)
                    (not (equal xs '(1 1)))) *xss*)
```

Argumenty nazwane *(:count)*

Ustala limit dokonanych przez funkcję zmian.
Domyślnie nie są nakładane żadne ograniczenia.

```
(defparameter *xs* '(0 1 0 2 0 3 0 4 0 5 0 6))
```

```
> (remove 0 *xs* :count 4)
(1 2 3 4 0 5 0 6)
```

Zagadka

Jak mają się do siebie funkcje:

```
(defparameter *ss* '("aB" "AB" "bA" "Ab"))
```

```
> (delete-duplicates
    (remove "aB" *ss*
      :test #'string=)
  :test #'string-equal)

> *ss*
```

```
> (remove "aB" *ss*
    :test #'string-equal
    :end 2)

> *ss*
```

A. Robią dokładnie to samo **B.** Zwracają jedynie tą samą wartość
C. Zwracają różne wartości

Zagadka

Jak mają się do siebie funkcje:

```
(defparameter *ss* '("aB" "AB" "bA" "Ab"))
```

```
> (delete-duplicates
    (remove "aB" *ss*
      :test #'string=)
  :test #'string-equal)
("bA" "Ab")
> *ss*
("aB" "AB" "bA" "Ab")
```

```
> (remove "aB" *ss*
    :test #'string-equal
    :end 2)

("bA" "Ab")
> *ss*
("aB" "AB" "bA" "Ab")
```

A. Robią dokładnie to samo **B.** Zwracają jedynie tą samą wartość
C. Zwracają różne wartości

Sekwencje

Podstawowe funkcje

```
;; Zwraca element sekwencji o danym indeksie lub błąd. „Setowalny".
(elt sekwencja indeks)
;; Zwraca długość listy
(length sekwencja)
;; Łączy ze sobą kolejne sekwencje traktując je jako wartości odpowiedniego typu
(concatenate typ sekwencja1 ... sekwencjaN)
;; Kopiuje wszystkie elementy sekwencji
(copy-seq sekwencja)
;; Tworzy sekwencję danego typu i długości wypełnioną elementami
(make-sequence typ n &key (initial-element nil))
```

```
> (elt "qwerty" 3)
#\r
> (length #(1 2 3 4 5))
5
> (concatenate 'list '(1 2) '(3) '(4 5))
(1 2 3 4 5)
```

Funkcje iterujące

Dodatkowe parametry nazwane: *key test test-not from-end start end*.
Dostępne także wersje `...-if` `...-if-not` oraz destrukcyjne.

```
(count obiekt sekwencja) ; Liczy liczbę wystąpień.
(find obiekt sekwencja) ; Obiekt lub nil.
(position obiekt sekwencja) ; Indeks w sekwencji lub nil.
(remove obiekt sekwencja) ; Sekwencja bez obiektu.
;; Usuwa duplikaty, zostawia ostatnie wystąpienia.
(remove-duplicates sekwencja)
;; Zamienia wystąpienia obiektu2 na obiekt1.
(substitute obiekt1 obiekt2 sekwencja)
```

Funkcje sortujące

Dodatkowe parametry nazwane: *key*.

```
(sort sekwencja porządek) ; Sortuje zgodnie z porządkiem
;; Sortuje stabilnie zgodnie z porządkiem
(stable-sort sekwencja porządek)
;; Łączy sekwencje zachowując porządek.
(merge typ sekwencja1 sekwencja2 porządek)
```

```
> (sort "cbadfe" #'char<)
"abcdef"
> (stable-sort '((4 1) (3 1) (2 0) (3 2) (4 2))
               #'< :key #'first)
((2 0) (3 1) (3 2) (4 1) (4 2))
> (merge 'vector #(1 3 5) #(2 4 6) #'<)
#(1 2 3 4 5 6)
> (merge 'list   #(4 2 1) #(3 5 6) #'<)
(3 4 2 1 5 6)
```

Predykaty sekwencyjne

Funkcje testujące zachowanie predykatów na sekwencjach.
Arność predykatu musi być równa liczbie podanych sekwencji.

```
> (every    #'evenp #(1 2 3 4 5))
NIL
> (some     #'evenp #(1 2 3 4 5))
T
> (notany   #'evenp #(1 2 3 4 5))
NIL
> (notevery #'evenp #(1 2 3 4 5))
T

> (every #'> #(1 2 3 4) #(5 4 3 2))
NIL
> (some  #'> #(1 2 3 4) #(5 4 3 2))
T
```

Podciągi

```
;; Wybiera podciąg o zadanym indeksie początkowym i końcowym.
(subseq sekwencja start &optional end)
;; Zwraca pierwszą pozycję wystąpienia podciągu. nil gdy nie występuje.
(search sekwencja1 sekwencja2)
;; Zwraca pierwszą pozycję na różniącą ciągi. Gdy takie same nil.
(mismatch sekwencja1 sekwencja2)
```

```
> (defparameter *my-string* (string "ABCD EFGH"))
> (subseq *my-string* 0 4)
"ABCD"
> (setf (subseq *my-string* 0 4) "123456")
> *my-string*
"1234 EFGH"
> (search "baba" "abbababaa")
2
> (mismatch #(2 3 5 7) #(2 3 4 5 6))
2
```

Konstrukcje funkcjonalne

```
;; Mapuje funkcję o odpowiedniej arności na podane sekwencje.
(map typ funkcja sekwencja &rest sekwencje)
;; Modyfikuje pierwsze elementy rezultatu umieszczając tam wynik.
(map-into rezultat funkcja sekwencja &rest sekwencje)
;; foldl
(reduce funkcja sekwencja)
```

```
> (map 'vector #'+ #(2 3 4) #(4 3 2))
#(6 6 6)
> (reduce #'- #(8 4 2 1))
1
```

from-end	*initial-value*	reduce f (a b c)
false	brak	*(f (f a b) c)*
false	*i*	*(f (f (f i a) b) c)*
true	brak	*(f a (f b c))*
true	*i*	*(f a (f b (f c i)))*

Zagadka

Z których zestawów wyrażeń można złożyć funkcję która dla sekwencji *xs* zwraca liczbę elementów większych niż 6.

A. **length**, **remove**, :test-not

B. **count**, -, lambda

C. lambda, count-if

Zagadka

Z których zestawów wyrażeń można złożyć funkcję która dla sekwencji *xs* zwraca liczbę elementów większych niż 6.

A. `length`, `remove`, `:test-not`

TAK `(length (remove 6 *xs* :test-not #'<=))`

B. `count`, `-`, `lambda`

C. `lambda`, `count-if`

Zagadka

Z których zestawów wyrażeń można złożyć funkcję która dla sekwencji *xs* zwraca liczbę elementów większych niż 6.

A. `length`, `remove`, `:test-not`
TAK `(length (remove 6 *xs* :test-not #'<=))`
B. `count`, `-`, `lambda`
NIE
C. `lambda`, `count-if`

Zagadka

Z których zestawów wyrażeń można złożyć funkcję która dla sekwencji *xs* zwraca liczbę elementów większych niż 6.

A. `length`, `remove`, `:test-not`
TAK `(length (remove 6 *xs* :test-not #'<=))`
B. `count`, `-`, `lambda`
NIE
C. `lambda`, `count-if`
TAK `(count-if #'(lambda (x) (> x 6)) *xs*)`

Listy

Lista jako zbiór

Funkcje nie gwarantują zachowania jakiegokolwiek porządku elementów.
Każda z funkcji dopuszcza argumenty nazwane: *key test test-not*.

(**member** x A) ; $x \in A$
(member-if p A) ; $\{x \mid x \in A \wedge p(x)\}$
(member-if-not p A) ; $\{x \mid x \in A \wedge \neg p(x)\}$

(**union** A B) ; $x \cup A$
(**nunion** A B)
(**intersection** A B) ; $x \cap A$
(**nintersection** A B)
(set-difference A B) ; $A \setminus B$
(nset-difference A B)
(set-exclusive-or A B) ; $A \div B$
(nset-exclusive-or A B)

(**adjoin** x A) ; A jeśli $x \in A$, $\{x\} \cup A$ w p.p.
(**subsetp** A B) ; $A \subseteq B$

Lista jako stos

Funkcje pobierają listę (adres) i modyfikują ją (tzn. działają destrukcyjnie).

```
;; Skraca listę o głowę, którą następnie zwraca.
(pop lista)

;; W miejscu listy tworzy (cons obiekt lista).
(push obiekt lista)

;; W miejscu zajmowanym przez listę umieszcza wartość zwróconą przez adjoin.
(pushnew obiekt lista &key key test test-not)
```

Lista własnościowa (property list)

Listy postaci $(p_1\ v_1\ p_2\ v_2\ \ldots\ p_n\ v_n)$ gdzie p_i jest kluczem (własnością) o wartości v_i.

```
;; Pobiera pierwszą wartość o własności równej kluczowi. Jeśli taka nie istnieje zwraca default.
(getf listaw klucz &optional (default nil))

;; Jeśli znajdzie w liście element odpowiadający jakiejś własności to zwraca trzy wartości: własność, wartość i ogon listy własnościowej rozpoczynający się od znalezionego klucza; w przeciwnym wypadku zwraca trzy wartości nil.
(get-properties listaw lista)

;; Usuwa z listy pierwszą własność o podanym kluczu. Zwraca fałsz jeśli nic nie zostało usunięte, a prawdę w przeciwnym przypadku.
(remf listaw klucz)
```

Lista jako słownik (listy asocjacyjne)

Słownik jest to lista par (klucz wartość).
Oczywiście każda lista list jest słownikiem w którym kluczami są głowy, a wartościami ogony.

```
;; Zwraca pierwszy element słownika którego głowa odpowiada kluczowi
(assoc klucz słownik &key key test test-not)
(assoc-if predykat słownik &key key)
(assoc-if-not predykat słownik &key key)

;; Zwraca pierwszy element słownika którego ogon odpowiada kluczowi
(rassoc klucz słownik &key key test test-not)
(rassoc-if predykat słownik &key key)
(rassoc-if-not predykat słownik &key key)

;; Odpowiada (cons (cons klucz wartość) słownik)
(acons klucz wartość słownik)
```

Lista jako słownik (listy asocjacyjne)

```
(sublis słownik drzewo &key key test test-not)
(nsublis słownik drzewo &key key test test-not)
```

Zwraca drzewo w którym występujące w słowniku klucze zostają zamienione na wartości.

```
> (sublis '((a . 1) (b . 2)) '(a (b 3) (4 a)))
(1 (2 3) (4 1))
```

```
(pairlis klucze wartości &optional słownik)
```

Poszerza słownik o pary (klucz wartość) stworzone z listy kluczy i listy wartości

```
> (pairlis '(a b c) '(1 2 3) '((X . 666)))
((C . 3) (B . 2) (A . 1) (X . 666))
```

Lista jako para

Lista której ogon niekoniecznie jest listą (wizualizowane w notacji kropkowej).

```
> (cons 'a 'b)
(A . B)
> (list 'a 'b)
(A B)
> (cons 'a '(b))
(A B)
```

Różnice w interpretacji są znaczące.

```
> (sublis '((a 1) (b 2)) '(a (b 3) (4 a)))
((1) ((2) 3) (4 (1)))
> (sublis '((a . 1) (b . 2)) '(a (b 3) (4 a)))
(1 (2 3) (4 1))
```

Mapowania

```
(map... funkcja lista &rest listy)

mapcar ; Konkatenuje wyniki funkcji na kolejnych elementach list
maplist ; Konkatenuje wyniki funkcji na kolejnych ogonach list
;; Jak mapcar ale funkcja musi zwracać listy które są łączone ncons
mapcan
;; Jak maplist ale funkcja musi zwracać listy które są łączone ncons
mapcon
mapc ; Jak mapcar ale zwraca zawsze pierwszą listę.
mapl ; Jak maplist ale zwraca zawsze pierwszą listę.
```

```
> (maplist #'(lambda (x) x) '(a b c))
((A B C) (B C) (C))
> (mapcan #'(lambda (x) (list 1 x)) '(a b c))
(1 A 1 B 1 C)
> (mapcon #'(lambda (x) (copy-list x)) '(a b c))
(A B C B C C)
```

Kopiowanie list

Efekt kopiowania listy zależy od jej interpretacji (lista, słownik, drzewo).

```
(defparameter *test* (list (cons 4 0) '(0 1 2 3)))
```

```
> *test*
((4 . 0) (0 1 2 3))
```

```
(defparameter *list*  (copy-list *test*))
(defparameter *alist* (copy-alist *test*))
(defparameter *tree*  (copy-tree *test*))
(setf (first (first *test*)) 'E)
(setf (third (second *test*)) 'C)
```

```
> *list*
((E . 0) (0 1 C 3))
> *alist*
((4 . 0) (0 1 C 3))
> *tree*
((4 . 0) (0 1 2 3))
```

Zagadka

Dopasuj zwracaną wartość i skutek uboczny do odpowiedniej funkcji.

```
(defparameter *xs* '(1 2 3 4 5 6))
A. (maplist #'(lambda (xs) (pop xs)) *xs*)
B. (maplist #'(lambda (xs) (setf *xs* (pop xs)))*xs*)
C. (maplist #'cdr *xs*)
D. (maplist #'(lambda (xs) (setf *xs* (cdr xs)))*xs*)
```

Funkcja	Zwracana wartość	*xs*
	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	(1 2 3 4 5 6)
	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	NIL
	(1 2 3 4 5 6)	6
	(1 2 3 4 5 6)	(1 2 3 4 5 6)

Zagadka

Dopasuj zwracaną wartość i skutek uboczny do odpowiedniej funkcji.

```
(defparameter *xs* '(1 2 3 4 5 6))
A. (maplist #'(lambda (xs) (pop xs)) *xs*)
B. (maplist #'(lambda (xs) (setf *xs* (pop xs)))*xs*)
C. (maplist #'cdr *xs*)
D. (maplist #'(lambda (xs) (setf *xs* (cdr xs)))*xs*)
```

Funkcja	Zwracana wartość	*xs*
C.	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	(1 2 3 4 5 6)
	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	NIL
	(1 2 3 4 5 6)	6
	(1 2 3 4 5 6)	(1 2 3 4 5 6)

Zagadka

Dopasuj zwracaną wartość i skutek uboczny do odpowiedniej funkcji.

```
(defparameter *xs* '(1 2 3 4 5 6))
A. (maplist #'(lambda (xs) (pop xs)) *xs*)
B. (maplist #'(lambda (xs) (setf *xs* (pop xs)))*xs*)
C. (maplist #'cdr *xs*)
D. (maplist #'(lambda (xs) (setf *xs* (cdr xs)))*xs*)
```

Funkcja	Zwracana wartość	*xs*
C.	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	(1 2 3 4 5 6)
D.	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	NIL
	(1 2 3 4 5 6)	6
	(1 2 3 4 5 6)	(1 2 3 4 5 6)

Zagadka

Dopasuj zwracaną wartość i skutek uboczny do odpowiedniej funkcji.

```
(defparameter *xs* '(1 2 3 4 5 6))
A. (maplist #'(lambda (xs) (pop xs)) *xs*)
B. (maplist #'(lambda (xs) (setf *xs* (pop xs)))*xs*)
C. (maplist #'cdr *xs*)
D. (maplist #'(lambda (xs) (setf *xs* (cdr xs)))*xs*)
```

Funkcja	Zwracana wartość	*xs*
C.	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	(1 2 3 4 5 6)
D.	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	NIL
B.	(1 2 3 4 5 6)	6
	(1 2 3 4 5 6)	(1 2 3 4 5 6)

Zagadka

Dopasuj zwracaną wartość i skutek uboczny do odpowiedniej funkcji.

```
(defparameter *xs* '(1 2 3 4 5 6))
A. (maplist #'(lambda (xs) (pop xs)) *xs*)
B. (maplist #'(lambda (xs) (setf *xs* (pop xs)))*xs*)
C. (maplist #'cdr *xs*)
D. (maplist #'(lambda (xs) (setf *xs* (cdr xs)))*xs*)
```

Funkcja	Zwracana wartość	*xs*
C.	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	(1 2 3 4 5 6)
D.	((2 3 4 5 6) (3 4 5 6) (4 5 6) (5 6) (6) NIL)	NIL
B.	(1 2 3 4 5 6)	6
A.	(1 2 3 4 5 6)	(1 2 3 4 5 6)

Napisy

Kontrola wielkości liter

Dodatkowe parametry nazwane: *start end*.
Dostępne także wersje destrukcyjne `n...`.

```
> (string-upcase "teSTOwy naPis")
"TESTOWY NAPIS"
> (string-downcase "teSTOwy naPis")
"testowy napis"
> (string-capitalize "teSTOwy naPis")
"Testowy Napis"
```

Przy braku zmian rezultatem może być napis lub jego kopia.

```
> (defparameter *napis* "XYZ")
> (defparameter *napisUp* (string-upcase *napis*))
> (setf (char *napis* 1) #\y)
> *napis*
"XyZ"
> *napisUp*
"XYZ" ∨ "XyZ"  ; Zależne od implementacji
```

Trimming

Występuje niejednoznaczość specyfikacji, analogiczna jak przy funkcjach kontrolujących wielkość liter.

```
;; Usuwa z obu końców napisu znaki występujące w sekwencji.
(string-trim sekwencja napis)
;; Jak string-trim ale obcina tylko z lewej.
(string-left-trim sekwencja napis)
;; Jak string-trim ale obcina tylko z prawej.
(string-right-trim sekwencja napis)
```

```
> (string-trim " -*" " ***to jest*napis-*-")
"to jest*napis"
> (string-right-trim '(#\0 "123" #\6) "666abw 260")
"666abw 2"
```

Konwersje

Konwersja liczby do napisu:

```
> (write-to-string 6.66)
"6.66"
> (write-to-string (/ 1 3))
"1/3"
> (write-to-string 10 :base 2)
"1010"
```

Konwersja napisu do liczby całkowitej:

```
> (parse-integer " 128 ")
128
5
> (parse-integer " 128 z hakiem" :junk-allowed t)
128
4
```

Czytanie danych z napisu

Funkcja `read-from-string` aplikuje do napisu cały lispowy *reader*.

```
> (read-from-string "(+ 1.2e2 6/8) makaron")
(+ 120.0 3/4)
13
> (read-from-string
    "#.(mapcar #'(lambda (x) (+ x 1)) '(1 2 3))")
(2 3 4)
42
> (read-from-string "#.(defparameter *x* 666)")
*X*
24
> *x*
666
```

Napisy a symbole

Tworzenie symbolu z napisu:

```
> (intern "XS") ; Lub po prostu read-from-string.
XS
NIL
```

Tworzenie napisu z symbolu:

```
>(symbol-name 'xs)
"XS"
```

Odczytanie symbolu o zmienionej nazwie:

```
> (defparameter xs-p 666)
XS-P
> (eval (read-from-string (concatenate
                'string (symbol-name 'xs) "-p")))
666
```

Przypisania

Przypisanie zmiennej

;; Wylicza kolejne wyrażenia i przypisuje ich wartości odpowiednim zmiennym, które są traktowane jak cytowane (quoted).
`(setq {zmienna wyrażenie}*)`

```
> (setq x (+ 1 2 3) y (cons x nil))
(6)
> x
6
> y
(6)
```

;; Działa jak `setq`, ale wszystkie przypisania odbywają się równocześnie.
`(psetq {zmienna wyrażenie}*)`

```
> (setq x 1) (setq y 2) (psetq x y y x)
> x
2
> y
1
```

Przypisanie miejscu pamięci

;; Wylicza kolejne wyrażenia i przypisuje ich wartości odpowiednim miejscom w pamięci.
(**setf** {miejsce wartość}*)

Wskazanie miejsca może się odbyć za pomocą wielu funkcji umożliwiających dostęp do elementów, np. `car`, `nth`, `cdr`, `elt`...

```
> (setf x '(1 2 3))
> x
(1 2 3)
> (setf (car x) 0)
> x
(0 2 3)
> (setf (symbol-value 'x) 666)
> x
666
```

;; Działa jak `setf`, ale wszystkie przypisania odbywają się równocześnie.
(**psetf** {miejsce wartość}*)

Przypisanie symbolowi

```
;; Przypisuje symbolowi obiekt zwrócony przez wyrażenie, symbol musi być cytowany.
(set symbol obiekt)
;; równoważne (setf (symbol-value symbol) obiekt)
```

```
> (set (if (eq 1 2) 'c 'd) 'foo)
FOO
> c
error
> d
FOO
```

Podsumowanie

Bibliografia


Paul Graham, *ANSI Common Lisp*, Prentice Hall 1996


Peter Seibel, *Practical Common Lisp*, Apress 2005


The Common Lisp Cookbook

Koniec