Zmienne, pętle, listy

Zmienne

Zmienne w Lispie zapisujemy za pomocą funkcji setq.

```
(setq x 10) => 10
X  => 10(let
(setq y (* x 20)) => 200
```

Wartość po znakach => określa wartość zwracaną przez intepreter Lispa.

Zmienne lokalne let*

Zmienne lokalne zapisujemy za pomocą konstrukcji ***let.***

```
(setq x 10)
(let ((x 20))
  (* x 30))
=> 600

(let ((zmienna1 wartość)
      (zmienna2 wartość))
   wyrażenie)
```

W powyższym kodzie zmienna x została przysłonięta przez konstrukcję let, dlatego w jej wnętrzu zmienna x ma wartość 20 nie 10. Po wyjściu z konstrukcji let zmienna nie jest już widoczna.
Aby usunąć zmienną używamy funkcji (makunbound 'zmienna).
Kostrukcja let* działa tak jak let z tym, że zmienne są przypisywane sekwencyjnie tzn. zmienne mogą się odwoływać do już przypisanych zmiennych, w przypadku let może się to odbywać równolegle.

```
(let* ((x 10)
       (y (+ x 10)))
    (* x y))
```

Stałe

Aby przypisać stałą należy użyć funkcji defconstant. Przyjęło się, że stałe zapisuje się ze znakiem +.

```
(defconstant +pi+ 3.14159265358979)
```

Zmienne globalne (dynamiczne)

Zmienne globalne przypisuje się za pomocą funkcji **defva**r lub **defparameter.** Przyjmuje się że zmienne globalne zapisujemy ze znakami *.

```
(defvar *lista* '(1 2 3 4))
(defparameter *zmienna* 10)
```

Funkcja **defvar** w odróżnieniu od **defparameter** przypisuje zmiennej wartość tylko raz na samym początku.

Wszystkie zmienne są zmiennymi leksykalnymi, tzn. widocznymi tylko wewnątrz struktury, w której są definiowane, chyba że zapisano inaczej. Za pomocą funkcji special mamy możliwość zadeklarowania zmiennych jako dynamiczne.

```
(defun foo (x)
  (declare (special x))
  (bar))

(defun bar ()
  (+ x 3))
```

W powyższym kodzie jeśli wywołamy funkcje (foo 3) zostanie zwrócona wartość 6, chociaż w funkcji bar nie zdefiniowano zmiennej x, zmienna przyjmie wartość 3.

Petle
Pętla dotimes.

```
(dotimes (zmienna_n opcjonalna-zwracana-wartość)
  wyrażenie
  ...)
```

Pętla dotimes wywoła wyrażenie n razy. W każdej iteracji pętli zmienna przyjmie kolejną wartość (od zera do n-1). Opcjonalnie można w instrukcji dotimes określić wartość zwracaną przez pętle. Jeśli nie określimy tej wartość pętla zwróci wartość nil.

```
(dotimes (i 10)
  (print i))
0
1
2
3
4
5
6
7
8
9
=> NIL
```

Powyższy kod wyświetli liczby od 0 do 9 i zwróci wartość nil.

Pętla dolist.

```
(dolist (e '(1 2 3 4 5 6) opcjonalna-zwracana-wartość)
  wyrażenie
  ...)
```

Pętla dolist wywoła się tyle razy ile jest elementów listy. W każdej iteracji zmienna e przyjmie kolejną wartość z listy. Opcjonalnie można określić wartość zwracaną przez pętle. Jeśli nie określimy tej wartość to pętla zwróci wartość nil.

Pętla do.

```
(do ((zmienna1 wartość-początkowa step)
     ...
     (zmienna2 wartość-początkowa step))
    (warunek-zakończenia wartość-zwracana)
  wyrażenia
  ...)
```

Pętla do może iterować po kilku zmiennych. Dla każdej z nich określa się wartość początkową oraz wyrażenie step, które zostanie wywołane po każdej iteracji. Zakończenie pętli następuje wtedy, gdy zostanie spełniony warunek. Pętla zwróci określoną wartość.

```
(do ((i 0 (incf i))
     (j 10 (decf j)))
    ((zerop j) 'done)
    (print (+ i j)))
```

Powyższy kod iteruje po dwóch zmiennych, z których jedna jest zwiększana w każdej iteracji a druga zmniejszana. Pętla zostanie przerwana, jeśli zmienna j osiągnie wartość zero. Zwrócona zostanie wtedy wartość 'done. W każdej iteracji pętli zostanie wyświetlona suma dwóch liczników "i" i "j" (zostanie wyświetlone 10 razy liczba 10). Makra incf i decf odpowiednio zwiększają i zmniejszą wartość swojego argumentu o 1.

Pętla loop w prostej postaci:

```
(loop
  wyrażenie
  ...)
```

jest to pętla nieskończona.

```
(let ((i 10))
  (loop
   (when (zerop i) (return))
   (print (decf i))))
9
8
7
6
5
4
3
```

```
2
1
0
=> NIL
```

Za pomocą makra return można przerwać pętlę.
Istnieje możliwość zastosowania rozszerzonej pętli loop. Poniżej przedstawiono jej możliwości.

- **Aby iterować po kolejnych liczbach od 1 do n:**

```
(loop
  for i from 1 to n ...)
(loop for x from 1 to 5
      for y = (* x 2)
      collect y)
```

- **Aby iterować po liczbach od 0 do n-1:**

```
(loop
  for i from 1 below n ...)
```

- **Aby iterować po liście wartości:**

```
(loop
  for element in (list 1 2 3 4 5 6) do wyrażenie ...)

(loop for x in '(a b c d e)
      do (print x) )

(loop for x in '(a b c d e)
      for y in '(1 2 3 4 5)
      collect (list x y) )
```

Wyrażenie do określa co ma zostać wywołane w każdej iteracji.

- **Aby iterować po tablicy, wektorze lub ciągu znaków :**

```
(loop
  for c across string collect c)

(loop
  for c across “ala” collect c)
```

collect c tworzy listę z kolejnych wartości c.

- **Aby iterować po kilku listach używamy wyrażenie and:**

```
(loop
  for i in list1 and j in list2 collect (list i j))

(loop
  for i in ‘(1 2 3) and j in ‘(a b c) collect (list i j))
```

- **Aby iterować po tablicy asocjacyjnej używamy notacji kropka:**

```
(loop
  for (k . v) in (pairlis '(a b c) '(1 2 3)) do
  (format t "~a => ~a~%" k v))
C => 3
B => 2
A => 1
```

- **Aby iterować po parach cons używamy wyrażenia on:**

```
(loop
  for para on (list 1 2 3 4 5 6) do
  (format t "~a => ~a~%" (car para) (cadr para)))
1 => 2
2 => 3
3 => 4
4 => 5
5 => 6
6 => NIL
```

Funkcja cadr zwraca drugi element listy.

- **Aby iterować po kluczach tablicy haszującej można użyć:**

```
(loop
  for k being the hash-keys of tablica-haszująca collect k)
```

- **Aby iterować po wartościach tablicy haszującej należy użyć poniższego kodu:**

```
(loop
  for v being the hash-values of tablica-haszująca collect v)
```

Poniższy kod wyświetla wszystkie klucze i wartości tablicy haszującej.

```
(loop
  for k being the hash-keys of tablica-haszująca do
  (format t "~a => ~a~%" k (gethash k tablica-haszująca)))
```

Tworzenie listy jeśli warunek jest spełniony:

```
(loop
  for i in (list 0 1 2 3 4 5 6)
  when (evenp i) collect i)
=> (0 2 4 6)
```

Powyższy kod utworzy listę wszystkich liczb parzystych (funkcja evenp).

Wykonywanie operacji jeśli warunek jest spełniony.

```
(loop
  for i from 0 while (< i 10) collect i)
=> (0 1 2 3 4 5 6 7 8 9)
```

W powyższym kodzie za pomocą collect tworzymy listę od 0 do 9.

Listy

Podstawowym typem danych w Lispie są listy, listy składają się z par. Aby utworzyć parę używamy konstrukcji cons. Pierwszy element pary jest to bieżący element listy, natomiast drugim elementem listy jest następna para tzn. reszta listy. Ostatni element listy musi być listą pustą () lub nil (synonim).

```
(cons 1 2)        => (1 . 2)
```

Aby utworzyć listę należy użyć konstrukcji:

```
(cons 1 (cons 2 (cons 3 nil)))          => (1 2 3)
'(1 . (2 . (3 . nil)))                  => (1 2 3)
```

Lub użyć funkcji list lub cytowania listy - znaku apostrofu '.

```
(list 1 2 3 4)          => (1 2 3 4)
'(1 2 3 4)              => (1 2 3 4)
```

Aby odwołać się do elementów listy należy używamy funkcji car lub first która zwraca pierwszy element listy, cdr lub rest zwracająca resztę listy.

```
(car '(1 2 3 4 5))           => 1
(cdr '(1 2 3 4 5))           => (2 3 4 5)
  (
```

Można też użyć jednej z funkcji: second, third, fourth, fifth, sixth, seventh, eighth, ninth, tenth, które zwracają odpowiednio elementy od 2 do 10 lub użyć funkcji (nth numer lista), która zwraca n-ty element listy. Trzeba pamiętać że listy przetwarzane są sekwencyjnie.
Aby skopiować listę używamy funkcji copy-list.

```
(copy-list lista)
```

Funkcja null zwraca wartość fałszu jeśli argument jest nie jest listą.
Funkcja listp zwraca prawdę jeśli argument jest listą.
Funkcja endp zwraca wartość prawdy jeśli jest to koniec listy.

Użycie listy jak stosu

Za pomocą funkcji push oraz pop możemy odpowiednio odłożyć i zdjąć elementy ze stosu (listy).

```
(setq stos nil)
(declare (special stos))
(push 10 stos)
(push 20 stos)
(push 30 stos)
(pop stos)
stos
=> (20 10)
```

Wektory

Wektory tworzymy za pomocą funkcji vector, znaku # lub funkcji make-sequance. Wektory są indeksowane od zera.

```
(vector 1 2 3 4)        => #(1 2 3 4)
#(1 2 3 4)              => #(1 2 3 4)
(make-sequence 'vector 10 :initial-element 0)  => #(0 0 0 0 0 0 0 0 0 0)
```

Funkcja make-sequence przyjmuje argument kluczowy (keyword argument) :initial-element, którego wartością jest element jakim zostanie wypełniony wektor.
Aby odwołać się do elementów wektora używamy funkcji aref lub elt, aby przypisać wartość do elementów wektora używamy funkcji setf która działa jak setq, z tym że jako drugi argument przyjmuje miejsce gdzie zostanie zapisana wartość przekazana jako drugi argument.

```
(setq v #(1 2 3 4))
(setf (elt v 1) 10)
v
=> #(1 10 3 4)
```

Ciągi znaków
Ciągi znaków zapisujemy używając podwójnego cudzysłowu " lub funkcji make-string.

```
"jakiś napis"           => "jakiś napis"
(make-string 10)        => ""
```

Aby odwołać się do elementu ciągu należy użyć funkcji elt lub aref jak w przypadku wektorów.
Aby przekonwertować ciąg znaków na liczbę można użyć funkcji parse-integer

```
(parse-integer "256")         => 256
```

Funkcja char-code zwraca kod ASCII dla danego znaku.

```
(char-code #\a)
=> 97
```

Poszczególne znaki zapisujemy stosując znaki #\
Funkcja code-char zwraca znak dla danej wartości liczbowej.

```
(code-char 100)
=> #\d
```

Tablice
Aby utworzyć tablicę należy użyć funkcji make-array.

```
(setq array (make-array '(4 4) :initial-element 0))
=> #2A((0 0 0 0) (0 0 0 0) (0 0 0 0) (0 0 0 0))
```

Powyższy kod utworzy tablicę (macierz) o wymiarze 4x4.
Aby odwołać się do elementów tablicy (macierzy) należy użyć funkcji aref.

```
(setf (aref array 1 1) 1)
```

```
array
=> #2A((0 0 0 0) (0 1 0 0) (0 0 0 0) (0 0 0 0))
```

Tablice mogą mieć zmienną liczbę elementów, aby utworzyć taką tablicę należy użyć parametru kluczowego :adjustable z wartością t.

```
(setq aa (make-array 10 :adjustable t :fill-pointer 0))
```

Aby dodać element do takiej tablicy należy użyć funkcji (vector-push-extend wartość tablica).
Tablice asocjacyjne (alisty)
Listy asocjacyjne składają się z par klucz wartość.

```
(setq alist '((a . 1) (b . 2) (c . 3)))
```

Funkcja assoc zwraca parę klucz wartość aby odwołać się do wartości należy użyć funkcji cdr.

```
(cdr (assoc 'a alist))
=> 1
```

Dodawanie elementów do list asocjacyjnej.

```
(setf alist (acons klucz wartość alist))
```

Aby utworzyć tablicę asocjacyjną można użyć funkcji pairlis.

```
(setq alist (pairlis '(a b c) '(1 2 3)))
=> ((a . 1) (b . 2) (c . 3))
```

Aby można było użyć łańcuchów znaków jako kluczy należy funkcji assoc przekazać parametr kluczowy :test z wartością #'equal. Funkcja assoc standardowo do porównań używa funkcji eq.

Listy właściwości (Plisty)
Listy właściwości są zapisywane za pomocą słów kluczowych.

```
(setq plist '(:a 1 :b 2 :c 3))
```

Aby odwołać się do elementów listy właściwości używamy funkcji getf.

```
(setf (getf plist :a) 10)
```

Aby usunąć element z listy używamy funkcji remf.

```
(remf plist :b)
```

plist

Tablice haszujące

Są podobne do tablic asocjacyjnych tylko są bardziej efektywne. Aby utworzyć tablicę haszującą należy użyć funkcji make-hash-table.

```
(setq hash (make-hash-table :test #'equal))
```

Funkcja make-hash-table przyjmuje dodatkowy parametr kluczowy test, za pomocą którego sprawdzane będą klucze, użycie funkcji #'equal umożliwi użycie napisów jako kluczy.

Aby odwołać się do tablicy używamy funkcji gethash.

```
(setf (gethash 'foo hash) "Napis")
(setf (gethash 'bar hash) "text")
(gethash 'bar hash)
=> "text"
```

Aby usunąć element z listy należy użyć funkcji remhash.

```
(remhash 'foo hash)
```

Wyczyszczenie całej tablicy następuje po wywołaniu funkcji (clrhash tablica). Funkcja (hash-table-count tablica) zwraca liczbę wpisów w tablicy haszującej.

Sekwencje

Sekwencja jest to grupa typów danych takich jak listy, wektory, tablice, ciągi znaków. Poniżej przedstawiono funkcje działające na sekwencjach:

`make-sequance`

```
(make-sequance typ rozmiar)
```

Funkcja tworzy sekwencje o podanym typie (list, array, string, vector) o danym rozmiarze. Można tworzyć tylko tablice jednowymiarowe.

`concatenate`

```
(concatenate typ sekwencja sekwencja)
```

Funkcja łączy dwie sekwencje w jedną, typ może przyjmować jedną z wartości: list, vector, string.

`elt`

```
(elt sekwencja n)
```

Funkcja zwraca n-ty element sekwencji.

`aref`

```
(aref sekwencja n)
```

Działa tak jak elt z tym że może być używana dla tablic wielowymiarowych.

`subseq`

```
(subseq sekwencja początek koniec)
```

Funkcja zwraca sekwencję zaczynającą się od elementu początek a kończącą się na wartości koniec.

`copy-seq`

```
(copy-seq sekwencja)
```

Zwraca kopie sekwencji.

`reverse`

```
(reverse sekwencja)
```

Odwraca kolejność elementów w sekwencji.

```
nreverse
(nreverse sekwencja)
```

Tak jak reverse z tym że może modyfikować swój argument (jest destrukcyjna).

```
length
(length sekwencja)
```

Zwraca liczbę elementów w sekwencji.

```
count
(count 3 '(2 3 4 5 2 3))
```

Funkcja zwraca liczbę elementów występujących w sekwencji, w tym przypadku 2.

```
count-if
(count-if #'oddp '(1 2 3 4))
```

Funkcja zlicza ile razy funkcja przekazana jako drugi parametr zwróci wartość t czyli prawdę.

```
count-if-not
(count-if-not #'funkcja sekwencja)
```

Odwrotna wersja funkcji count-if.

```
remove
(remove element sekwencja)
```

Funkcja usuwa wszystkie wystąpienia danego elementu z sekwencji.

```
remove-if
(remove-if #'funkcja sekwencja)
```

Usuwa wszystkie elementy dla których podana funkcja zwróci wartość prawdy (t).

```
remove-if-not
(remove-if-not #'funkcja sekwencja)
```

Odwrotna wersja funkcji remove-if.

```
substitute
(substitute zastąpiony zastępowany sekwencja)
```

Funkcja zastępuje każdy element zastąpiony zastępowanym.

```
substitute-if
(substitute-if zastępowany #'funkcja sekwencja)
```

Zastępuje wszystkie wystąpienia elementu zastępowany dla których #'funkcja zwróci wartość prawdy.

```
substitute-if-not
(substitute-if-not element #'funkcja sekwencja
```

Odwrotna wersja funkcji substitute-if.

```
remove-duplicates
(remove-duplicates sekwecja)
```

Usuwa wszystkie powtarzające się elementy z sekwencji.

```
merge
(merge 'typ (1 2 3) (4 5 6) #'>)
```

Funkcja łączy dwie sekwencje o podanym typie ustalając kolejność za pomocą funkcji przekazywanej jako czwarty parametr.

```
position
(position element sekwencja)
```

Funkcja zwraca pozycje elementu w sekwencji lub wartość nil.

```
find
(find element sekwencja)
```

Funkcja zwraca znaleziony element w sekwencji lub wartość nil.

```
search
(search sekwecja1 sekwencja2)
```

Funkcja zwraca pozycje wystąpienia sekwencji1 w sekwencji2.

```
sort
(sort sekwencja #'>)
```

Funkcja sortuje sekwencję za pomocą funkcji przekazywanej jako drugi parametr.