Definieren eigener Funktionen

Bisher:

Mit den bisher bekannten elementaren Funktionen sind bereits sehr komplexe Ausdrücke formulierbar. Allerdings ist es nicht möglich, einen komplexen Ausdruck so abzuspeichern, daß man ihn auf diverse andere Ausdrücke anwenden kann.

Beispiel:

```
-> (setq liste '(a b c d e f g))
-> (setq ergebnis
          (car (cdr (cdr
                        (cdr (cdr (cdr liste)))))))
f
```

Was fehlt?

Was man sich wünscht ist (ähnlich den Variablen), komplexe funktionale Gefüge mit einem eindeutigen Namen abzukürzen und zu beliebiger Zeit wie eine vorgegebene Funktion auf beliebige Argumente anzuwenden.

FUNKTIONEN

Funktionsdefinition

Mittels **defun** kann eine Funktion neu definiert und an einen Namen gebunden werden. Genauer gesagt wird dabei die Funktionsdefinition in der Funktionszelle des durch den Namen bezeichneten Symbols abgelegt.

```
(  defun      square     (x)        (* x x)  )
```

- defun → Funktion zum Generieren von Funktionen
- square → Name der Funktion
- (x) → Parameter-liste
- (* x x) → Funktionskörper, das eigentliche Programm

Name: ein beliebiges Symbol, wobei eine möglicherweise bereits existierende Definition einer Funktion gleichen Namens einfach überschrieben wird.

Parameterliste: spezifiziert die formalen Parameter, die im Funktionskörper auftreten können.

Funktionskörper: eine beliebige Folge symbolischer Ausdrücke, die dem “Programm” entspricht.

Wichtig: Der Wert eines Funktionsaufrufs entspricht dem Wert des zuletzt ausgewerteten symbolischen Ausdrucks.

Beispiel:

```
->(defun sechstes (liste)
      (car (cdr (cdr
             (cdr (cdr (cdr liste)))))))
sechstes

-> (sechstes '(a b c d e f g))
f

-> (car (sechstes '(a b c d e (f g) h)))
f
```

Bemerkung

Argumente, die beim Aufruf einer Funktion angegeben werden, heißen **aktuelle Parameter** aP, Argumente, die bei der Definition einer Funktion angegeben werden heißen **formale Parameter** fP. Die Anzahl der formalen und aktuellen Parameter muß übereinstimmen.

Aufgabe

Schreibe eine Funktion, die eine Liste als Argument erhält und eine neue Liste als Wert zurückliefert, in der das vormals erste Element nun das letzte ist.
Z.B.: (a b c d) $\rightarrow$ (b c d a)

Lösung

```
-> (defun linksdrehen (liste)
      (append (cdr liste)
              (list (car liste))))
linksdrehen

-> (linksdrehen '(ihr kommt nach hause))
(kommt nach hause ihr)

-> (linksdrehen
      (linksdrehen '(ihr kommt nach hause)))
(nach hause ihr kommt)
```

Beispiel

Aufgabe

Schreibe eine Funktion **neues-vorletztes**, die ein beliebiges Element vor das letzte Element einer beliebigen Liste einfügt.
Z.B.: (neues-vorletztes 'd '(a b c e)) → (a b c d e)

Lösung

```
-> (defun neues-vorletztes (element liste)
      (reverse
       (append (list (car (reverse liste)))
               (list element)
               (cdr (reverse liste)))))
neues-vorletztes

-> (neues-vorletztes 'kleinen
                     '(ich sehe diesen Jungen))
(ich sehe diesen kleinen jungen)
```


Prädikate

Um auf unterschiedliche Eingaben variabel reagieren zu können, ist es notwendig, Daten zu testen und aufgrund der Ergebnisse der Tests zu verzweigen bzw. die Verarbeitung erst zuzulassen, wenn die Daten bestimmte Bedingungen erfüllen. Um Werte (oder Daten) auf bestimmte Eigenschaften hin zu überprüfen, existieren in Lisp spezielle Funktionen, die sogenannten **Prädikate**.

Prädikate sind Funktionen

Ein Prädikat ist eine Funktion, deren Wert einem der Wahrheitswerte **wahr** oder **falsch** entspricht.

Wahrheitswerte in Lisp

Falsch wird immer durch das spezielle Symbol **NIL** angezeigt (beachte: NIL ist auch das Symbol für die leere Liste),

Wahr wird häufig durch das spezielle Symbol **T** angezeigt. Es wird aber generell alles von NIL verschiedene als Anzeige für Wahr verwendet.

Typprädikate

Der Typ eines Lispobjektes kann mittels der Prädikate **atom**, **listp**, **numberp**, **stringp**, **symbolp**, **integerp** usw. überprüft werden.

Das Prädikat **null** ist erfüllt, falls es sich bei dem überprüften Objekt um die leere Liste handelt.

Prädikat:	ATOM		
Syntax:	(atom <ausdruck>)		
Arbeitsweise:	ATOM liefert T, falls <ausdruck> ein Atom, sonst NIL		
Bsp:	(atom 'a)	→	T
	(atom 8)	→	T
	(atom '(a b c))	→	NIL
	(atom NIL)	→	T

PRÄDIKATE

Prädikat:	LISTP		
Syntax:	(listp <ausdruck>)		
Arbeitsweise:	LISTP liefert T, falls <ausdruck> eine Liste sonst NIL		
Bsp:	(listp 'a)	→	NIL
	(listp 8)	→	NIL
	(listp '(a b c))	→	T
	(listp NIL)	→	T
Hinweis:	Prädikate in Commonlisp haben häufig den Suffix p.		

Prädikat:	NULL		
Syntax:	(null <ausdruck>)		
Arbeitsweise:	NULL liefert T, falls <ausdruck> = NIL, sonst NIL		
Bsp:	(null nil)	→	T
	(null '())	→	T
	(null 'a)	→	NIL
	(null '(a b c))	→	NIL
Hinweis:	NULL sollte immer dann verwendet werden, wenn überprüft werden soll, ob eine Liste leer ist.		

Weitere Typprädikate sind (A steht für <ausdruck>):

numberp	Ist A eine Zahl?
symbolp	Ist A ein Symbol?
stringp	Ist A ein String?
endp	Ist A, was eine Liste sein muß, die leere Liste?
consp	Ist A eine cons Zelle?
⋮	⋮

das allgemeine Typprädikat TYPEP

Es gibt in Commonlisp eine allgemeine Typprädikatsfunktion, die den Typ eines beliebigen Ausdrucks testet:

Prädikat: TYPEP
Syntax: (typep <ausdruck> <type>)
Arbeitsweise: TYPEP liefert T, falls <ausdruck>
vom Typ <type>, sonst NIL
Bsp: (typep 'a 'atom) → T
(typep 'a 'number) → NIL
(typep '(a b c) 'atom) → NIL
(typep '(a b c) 'list) → T

Hinweis: TYPEP ist wichtig, da Commonlisp es erlaubt,
eigene Typen zu definieren und
man automatisch eine Typfunktion besitzt

Gleichheitsprädikate

Unter der Gleichheit zweier Lispobjekte kann man verschiedene Sachen verstehen.

eq Testet, ob zwei Objekte genau dieselben Speicherzellen belegen.

eql Berücksichtigt zusätzlich, daß z.B. eine Zahl in Common Lisp in Form mehrerer physikalischer Kopien vorliegen kann, d.h. zwei Objekte sind **eql**, wenn sie konzeptuell gleich sind.

equal Testet auf strukturelle Gleichheit, d.h. die Ausgaberepräsentationen zweier Objekte müssen gleich sein.

Zusätzlich stehen für einzelne Datentypen spezielle Gleichheitsprädikate bereit (**=**, **string=**, ...).

Beispiele

(equal 'a 'b)	→	NIL
(equal 'a 'a)	→	T
(eq 'a 'a)	→	T
(eql 'a 'a)	→	T
(eql 3 3)	→	T
(eql 3 3.0)	→	NIL
(equal '(a b) '(a b))	→	T
(equal '(a b) '(b a))	→	NIL
(eq '(a b) '(a b))	→	NIL
(equal '(a b c) '(a (b) c))	→	NIL

```
-> (setq liste1 '(a b c))
-> (setq liste2 '(a b c))

-> (equal liste1 liste2)
T
-> (eq liste1 liste2)
NIL

-> (setq liste1 '(a b c))
-> (setq liste2 liste1)

-> (equal liste1 liste2)
T
-> (eq liste1 liste2)
T
```