

LISP – Liste Version 1.4

A. = Ausdruck; L. = Liste; Z. = Zahl

- Überprüfen usw.

Name und Syntax	Was passiert?	Anmerkungen / Erklärungen
(member <A.> <L.>)	Wenn A. in L. vorhanden ist, gibt LISP alles zurück, was nach dem entsprechenden Atom kommt (einschließlich dem Ausdruck), wenn nicht: NIL.	Wenn A. auch eine Liste ist, gibt LISP immer NIL zurück. Sucht nur auf dem Top Level!
(consp <A.>)	Überprüft, ob A. eine nicht-leere Liste ist. Ausgabe: T / NIL.	
(endp <A.>)	Überprüft, ob A. eine leere Liste ist. Ausgabe: T / NIL.	Arbeitet nur mit Listen, sonst Fehlermeldung.
(null <A.>)	s. endp	Arbeitet auch mit Nicht-Listen.
(zerop <A.>)	Prüft, ob die Zahl gleich 0 ist. Ausgabe: T / NIL.	Arbeitet nur mit Zahlen, sonst Fehlermeldung.
(equal <A.1> <A.2>)	Prüft ob A.1 und A.2 gleich sind. Ausgabe: T / NIL.	8 ungleich 8.0; (abc) ungleich (acb)!
(= <Z.1> <Z.2> <Z.3> ...)	Prüft ob die Zahlen gleich sind. Ausgabe: T / NIL.	Arbeitet nur mit Zahlen, sonst Fehlermeldung. 8 gleich 8.0.
(< <Z.1> <Z.2>)	Prüft, ob Z.1 kleiner als Z.2; ebenso: > Ausgabe: T / NIL.	Arbeitet nur mit Zahlen, sonst Fehlermeldung.
(not <A.1> <A.2> <A.3> ...)	Prüft, ob ein A. zu NIL evaluiert. Ausgabe: T / NIL.	Alles was nicht explizit zu NIL evaluiert, gilt als wahr .
(or <A.1> <A.2> <A.3> ...)	Gibt NIL zurück wenn alle Ausdrücke zu NIL evaluieren, sonst T.	Auswertung nur auf dem Top Level, evaluieren Elemente nicht zu einem Wahrheitswert, werden sie direkt ausgegeben.
(and <A.1> <A.2> <A.3> ...)	Gibt NIL zurück wenn einer der Ausdrücke zu NIL evaluiert, wenn alle Ausdrücke zu T evaluieren, gibt es T zurück, wenn ein Ausdruck weder zu T noch zu NIL evaluiert, gibt es diesen zurück. Wenn mehrere Ausdrücke nicht zu einem Wahrheitswert evaluieren, gibt es immer den letzten zurück.	Beispiele: (and T T) → T (and T T NIL) → NIL (and 23) → 23 (and 1 2 3) → 3 (and T 5) → 5 usw...
(atom/listp/stringp/symbolp/numberp <A.>)	Überprüft, ob A. ein Atom, eine Liste, ein String, ein Symbol oder eine Zahl ist, Ausgabe: T / NIL.	
(length <L.>)	Zählt Elemente (auf dem Top Level) einer Liste. Ausgabe: Zahl	Arbeitet nur mit einer Liste, sonst Fehlermeldung. Zählt nur auf dem Top Level.

- Manipulationsmöglichkeiten für Listen:

Name und Syntax	Was passiert?	Anmerkungen / Erklärungen
(first / second ... tenth <L.>)	Gibt das erste / zweite ... zehnte Element einer Liste aus (als Atom).	Arbeitet nur mit einer L., sonst Fehlermeldung.
(last <L.>)	Gibt das letzte Element einer Liste aus (als Liste). → (last '(a b c)) → (c)	dito
(rest <L.>)	Gibt alle Elemente einer Liste (als neue Liste) aus, außer das erste.	dito
(nth <Z.> <L.>)	Gibt das Element aus L. aus, das mit Z. gewünscht wurde (als Atom).	Achtung! Es wird bei 0 angefangen. Arbeitet nur mit Listen. → (nth 3 '(a b c d)) → d
(reverse <L.>)	Dreht L. um. Ausgabe: Neue Liste.	Arbeitet nur mit Listen.
(list <A.1> <A.2> <A.3> ...)	Erzeugt eine Liste mit den Ausdrücken.	

Name und Syntax	Was passiert?	Anmerkungen / Erklärungen
(append <L.1> <L.2> <L.3> ...)	Fügt mehrere L. zusammen.	Achtung bei leeren Listen, die fallen weg. → (append () () `(a)) → (a) oder: (append () ()) → NIL Kann benutzt werden, um schmutzige Listen zu erstellen. → (append `(a) `a) → (a . a)
(cons <A.> <L.>)	Fügt A. vorne in eine Liste ein.	Auch hier können schmutzige Listen erstellt werden. → (cons `a `b) → (a . b)
(acons <key> <datum> <alist>)	Fügt <key> . <datum> vorne in eine assoziierte L. ein.	Funktioniert auch mit „normalen“ Listen. Bsp.: > (setq liste `((age . 38)(krawatte . hässlich))) Ausg.: ((age. 38) (krawatte . hässlich)) > (acons `farbe `gelb liste) Ausg.: ((farbe . gelb) (age . 38) (krawatte . hässlich))
(pairlis <keylist> <datelist>)	Erstellt Assoziationslisten mit den Schlüsseln in <keylist> und den Daten in <datelist>.	<keylist> und <datelist> dürfen hierbei natürlich keine schmutzigen Listen sein.
(assoc <key> <alist>)	Gibt Paar mit <key> aus <alist> aus. Nicht gefunden: NIL. Funktioniert auch mit nicht-schmutzigen Listen.	Gibt immer nur den ersten gefundenen Schlüssel aus! Beispiel mit <i>acons</i> , <i>pairlis</i> , <i>assoc</i> und <i>rassoc</i> siehe weiter unten!
(rassoc <date> <alist>)	Gibt Paar mit <date> aus <alist> aus. Nicht gefunden: NIL. Funktioniert nur mit Assoziationslisten.	Beispiel mit <i>acons</i> , <i>pairlis</i> , <i>assoc</i> und <i>rassoc</i> siehe weiter unten!

▪ Zahlenspielchen:

Name und Syntax	Was passiert?	Anmerkungen / Erklärungen
(+ <Z.1> <Z.2> <Z.3> ...)	Addiert die Z., Ausgabe: Ergebnis-Zahl	
(- <Z.1> <Z.2> <Z.3> ...)	Subtrahiert Z.2 bis Z.n von Z.1, Ausgabe: Ergebnis-Zahl	
(SQRT <Z.>)	Zieht die Wurzel aus Z., Ausgabe: immer Kommazahl. → (sqrt 4) → 2.0	Nur eine Zahl!
(1+ <Z.>)	Zählt 1 zu Z. hinzu.	Nur eine Zahl!
(1- <Z.>)	Zieht 1 von Z. ab.	Nur eine Zahl!
(max <Z.1> <Z.2> <Z.3> ...)	Gibt die größte Z. zurück. → (max 3 4 5) → 5	siehe Übungsblatt 7, Aufg. 3
(* <Z.1> <Z.2> <Z.3> ...)	Multipliziert die Z.	
(/ <Z.1> <Z.2> <Z.3> ...)	Dividiert Z.1 durch Z.2, das Ergebnis durch Z.3 usw.	Gibt immer den Bruch mit dem kleinsten Nenner aus. → (/ 45 5 6) → 3/2

▪ Definieren von Funktionen:

(defun <Fkt.name> <Parameterliste> <Fkt.körper1> <Fkt.körper2> ...)

Zur Erklärung:

- Fkt.name: Der Name, unter dem die Fkt. später aufgerufen werden kann. Bereits vorhandene Namen werden überschrieben.
- Parameterliste: Eine Liste, die die Argumente enthält, die der Fkt. übergeben werden.
- Fkt.körper: Der Funktionskörper enthält das, was gemacht werden soll. Sollte sinnigerweise die Parameter enthalten, die in der Parameterliste übergeben werden. Es können mehrere Funktionskörper hintereinander geschrieben werden.

Ausgabe: Funktionsname

Beispiel mit hintereinander „geschalteten“ Fkt.körpern:

```
> (defun f (a) (setq b (+ a 3)) (+ b 4))
F
> (f 4)
11
```

▪ Fallunterscheidungen usw.:

(cond (<test1> <form1>) (<test1> <form2>) ...)

Zur Erklärung:

- test: Die Tests werden solange durchgeführt, bis einer zu T evaluiert. Dann wird das in <form> gewünschte ausgeführt. Will man also, dass eine <form> auf jeden Fall ausgeführt wird wenn alle Tests davor **nicht** zu T evaluierten dann kann man statt einem <test> auch einfach ein T schreiben.
- Ausgabe von „cond“ ist natürlich die jeweilige Ausgabe der zugehörigen <form>.

Beispiel:

```
> (cond ((> 1 3) (+ 2 2)) ((> 3 2) (+ 2 3)))
5
```

(if <test> <form1> <form2>)

Zur Erklärung:

- Binäre Verzweigung. Wenn <test> zu T evaluiert dann <form1> wenn nicht dann <form2>.

Beispiel:

```
> (if (< 1 2) (+ 2 2) (+ 2 3))
4
```

▪ set, setq, setf, let:

(set <symbol> <A.>)

Weist dem <symbol> den A. zu. Wichtig: Es tut dies immer **global**.

(setq <symbol> <A.>)

Weist dem <symbol> den A. zu. Wichtig: Es tut dies immer für die momentane *Bindungsumgebung*. Das heißt wenn setq auf der globalen BU angewendet wird, dann weist auch setq **global** zu.

(setf)

setf kann dazu verwendet werden, Teile von Liste zu manipulieren, am besten wird dies an einem **Beispiel** deutlich:

```
> (setq liste1 `(a b c))
(A B C)
> (setq liste2 '(2 3))
(2 3)
> (setf (rest liste1) liste2)
(2 3)
> liste1
(A 2 3)
```

Diese Ausgabe kommt zustande weil **setf** immer das zurück gibt, was eingesetzt wurde, also wird hier das zweite Argument (liste2) zurückgegeben.

Evaluieren wir aber liste1 direkt, sehen wir, dass – wie gewünscht – der „rest“ von liste1 (also (b c)) durch liste2 ersetzt wurde.

(let ((<var1> <value1>) (<var2> <value2>)...(<varn> <valuen>)) <body>)

Da die (nur) lokale Definition von Variablen über setq und setf trickreich sein kann (setq definiert nur lokal, wenn die entspr. Variable bereits definiert wurde) gibt es die Möglichkeit, Variablen mit let bzw. let* zu definieren. Diese Funktion bietet zwei entscheidende Vorteile gegenüber set/setq/setf: erstens kann man mit ihr mehreren Variablen gleichzeitig Werte zuweisen und zweitens erzeugt let eine eigene Bindungsumgebung, sodass die von let erzeugten/veränderten Variablen nur in dieser Bindungsumgebung funktionieren. Das hört sich komplizierter an, als es ist: auf Deutsch heißt das, dass die von let verwendeten Variablen **nur** innerhalb der Klammern nach let gelten.

Beispiel:

```
(defun beispiel-fuer-let ()
  ( (setq x 12)
    (let ( (x 14) (y 16) (z 18))
      )
    )
  )
```

Im Beispiel hätte nach Ende der Funktion die Variable x den Wert 12, und *nur innerhalb des Funktionskörpers von let* den Wert 14. Sobald sich also die letzte Klammer von let schließt, hat x wieder den alten Wert. Das ist nützlich, da man in dem Funktionskörper von let (in der Syntax oben: <body>) nun allerhand andere Funktionen aufrufen und so Berechnungen anstellen kann. Ein gutes Beispiel, bei dem sich let eher anbietet als setq, ist die Iteration.

Fehlt nur noch eines, nämlich der Unterschied zwischen let und let*.

Kurz gesagt: in let geschieht die Wertzuweisung der Variablen parallel, während sie in let* sequenziell abgearbeitet wird. Das heißt: benutzt man let, werden den Variablen 1 bis n *gleichzeitig* die Werte zugewiesen, bei let* geschieht das *nacheinander*, (als Sequenz) also erst die erste, dann die zweite usw. Beispiele:

```
(let ( (erste 1) (zweite 2) (dritte (+ erste zweite)))dritte)
(let* ( (erste 1) (zweite 2) (dritte (+ erste zweite)))dritte)
```

Im ersten Fall gibt LISP eine Fehlermeldung wieder, da gleichzeitig versucht wird, den Variablen erste, zweite und dritte einen Wert zuzuweisen; das geht aber nicht, da sich der Wert von dritte erst aus dem Wert von erste und zweite ergibt. Benutzt man allerdings let*, funktioniert es, da dann nacheinander den drei Variablen die Werte zugewiesen werden und so die Addition von erste und zweite möglich ist.

▪ Iteration:

```
(do ((<IV> <AW> <IS>)) (<ABBRUCH>) <RUMPF>))
```

Zur Erklärung:

- IV: Iterationsvariable
- AW: Anfangswert der IV
- IS: Iterationsschritt
- ABBRUCH: Die Abbruchbedingung, wenn dies eintritt dann stoppt die Iteration.
- RUMPF: Dies hier wird solange gemacht bis ABBRUCH eintritt.

Beispiel:

```
> do ((i 1 (1+ i)))((> i 10)) (print (* i i))
```

Also:

Gib mir solange das Quadrat von i (Rumpf) aus, bis die Iterationsvariable i, die einen Anfangswert von 1 hat, bei einem Erhöhungsschritt von 1 größer als 10 ist.

Ausgabe:

```
1
4
9
16
25
36
49
64
81
100
NIL
```

▪ Ausgeben und Verfolgen:

```
(print <form> <stream>)
```

<form> wird auf <stream> ausgegeben, wenn <stream> weggelassen wird, dann wird auf dem Bildschirm ausgegeben.

```
(trace <Fkt.name1> <Fkt.name2> <Fkt.name3> ... )
```

Die Zwischenergebnisse einer komplexen Funktion mit dem Namen <Fkt.name> werden ausgegeben.

Mit (untrace <Fkt.name1> <Fkt.name2> ...) kann dies rückgängig gemacht werden.

- Eingeben lassen und lesen:

`(format T „<formatierter Text>“)`

mit format lassen sich Ausgaben formatiert ausgeben (z.B. mittels „T“ (siehe Syntax oben) auf dem Bildschirm). Eine formatierte Ausgabe meint hier beispielsweise, dass sie in einer neuen Zeile beginnt¹ und dass sie nicht, wie bei einem String, mit Anführungsstrichen angezeigt wird.

Format kann dazu benutzt werden, Eingabeaufforderungen anzeigen zu lassen; die Eingabe kann dann mit

`(read)`

gelesen und in eine Variable gespeichert werden. Wird die Funktion (read) aufgerufen, wartet sie solange, bis eine Eingabe gemacht wird.

Beispiel:

```
(setq eingabe (read (format T „Bitte Eingabe machen: “)))
```

Hier bekommt die Variable „eingabe“ den Wert der Eingabe zugewiesen. ACHTUNG: (format) selber hat nichts mit einer Eingabe zu tun, es kann nur zum formatieren von *Ausgaben* benutzt werden.

- Beispiel mit „acons“, „pairlis“, „assoc“ und „rassoc“:

Zuerst definieren wir eine Assoziationsliste, die „liste1“ heißen soll, mit folgenden Einträgen:

Schlüssel – Datum; Wochentag – Montag; Uhrzeit – zwölf; Wetter – Gut

```
(setq liste1 (pairlis `(Schlüssel Wochentag Uhrzeit Wetter) `(Datum Montag zwölf Gut)))
```

Ausgabe:

```
((WETTER . GUT) (UHRZEIT . ZWÖLF) (WOCHENTAG . MONTAG) (SCHLÜSSEL . DATUM))
```

Jetzt verwenden wir „acons“ um das Schlüssel-Datum-Paar „Jahr – 2004“ einzufügen:

```
(setq liste1 (acons `jahr 2004 liste1))
```

Ausgabe:

```
((JAHR . 2004) (WETTER . GUT) (UHRZEIT . ZWÖLF) (WOCHENTAG . MONTAG) (SCHLÜSSEL . DATUM))
```

Jetzt verwenden wir statt „acons“ „cons“ um das Paar „Jahr – 2003“ einzufügen (achte auf die *schmutzige Liste*, die wir einfügen!):

```
(setq liste1 (cons `(Jahr . 2003) liste1))
```

Ausgabe:

```
((JAHR . 2003) (JAHR . 2004) (WETTER . GUT) (UHRZEIT . ZWÖLF) (WOCHENTAG . MONTAG) (SCHLÜSSEL . DATUM))
```

Jetzt benutzen wir „rassoc“, um uns das Paar ausgeben zu lassen, das das Datum „Montag“ hat:

```
(rassoc `Montag liste1)
```

Ausgabe:

```
(WOCHENTAG . MONTAG)
```

Und jetzt benutzen wir noch „assoc“, um uns das Paar ausgeben zu lassen, das den Schlüssel „wetter“ hat:

```
(assoc `wetter liste1)
```

Ausgabe:

```
(WETTER . GUT)
```

Lisp-Liste 1.4, 19. Februar 2004

¹ Das entsprechende Formatierungszeichen wird mit einer Tilde (~) eingeleitet. ~% lässt z.B. die Ausgabe in einer neuen Zeile erscheinen. Syntaxbeispiel: (format T „~%<formatierter Text>“. Achtung: die Formatierungszeichen müssen nicht gelernt werden!