Lisp – Eine Kurzeinführung

- Lisp ist eine "listen-orientierte" Sprache $\rightarrow$ Verarbeitung symbolischer Informationen
- Lisp unterstützt primär einen *funktionalen*, jedoch auch den *prozeduralen* und *objekt-orientierten* Programmierstil
- Von John McCarthy 1958 entwickelt (mit Fortran eine der ältesten Sprachen überhaupt)
- "Interpretierte" Sprache mit extrem einfacher Syntax (Klammer-Notation)
- Standardisiert nach ANSI
- Lisp-Maschinen (spezielle Rechnerarchitektur mit Lisp-Prozessor)

Ressourcen

Quellen:

- Paul Graham, *ANSI Common Lisp*, 1996, Prentice Hall
- Guy Steele, *Common Lisp, second edition*, 1990, Digital Press
- Einführung in die LISP-Programmierung, Blockkurs jew. vor dem WS

Interpreter im Linux-Pool:

- Franz Allegro CommonLisp V. 6.2 (/soft/acl62/alisp)
  `http://www.informatik.uni-ulm.de/ki/Edu/Vorlesungen/GdKI/WS0203/.emacs`
  nach `$HOME` kopieren; `M-x run-lisp`; Beenden mit `:ex`; Evaluierung mit `C-c C-c`
- CMU CommonLisp (/soft/CMU_CL/bin/c_lisp)
- CLISP (/usr/bin/clisp)

Warum nicht C oder Java?

```
(defun sum (n)
  (let ((s 0))
    (dotimes (i n s)
      (incf s i))))
```

```
int sum(int n) {
  int i, s = 0;
  for(i = 0; i < n; i++)
    s += i;
  return s;
}
```

```
(defun addn (n)
  #'(lambda (x) (+ x n)))

> (addn 3)
#<Interpreted Closure (:INTERNAL ADDN) @ #x71b06c02>
> (funcall (addn 2) 3)
5
```

$\longrightarrow$ Lisp-Programme sind selber Lisp-Strukturen

read-eval-print-Schleife

Interaktives "front-end" jedes Lisp-Interpreters heisst *top-level* (in Allegro CL durch `CL-USER(92):` gekennzeichnet). Im folgenden durch das Zeichen ">" abgekürzt:

```
> 1
1
>
```

→ Zahlen werden zu sich selbst/ihrem Wert evaluiert.

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

```
> (+ 2 3 4 5)
14
```

```
> (/ (- 7 1) (- 4 2))
3
```

→ Alle Lisp-Ausdrücke sind entweder *Atome* (wie 1) oder *Listen*, die wiederum aus einer oder mehreren Ausdrücken bestehen.

Evaluierung

Ausdrücke werden wie folgt evaluiert (evaluation rule):

1. Zunächst werden die Argumente von links nach rechts evaluiert.
2. Dann werden die Werte dieser Evaluierung an den Operator weitergereicht.

Bei Fehlern: Die Break-Loop (hier für Allegro CL)

```
CL-USER(93): (/ 1 0)
Error: Attempt to divide 1 by zero.
  [condition type: DIVISION-BY-ZERO]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this process.
[1] CL-USER(94): :res
CL-USER(95):
```

“Quote” und Daten

Ausnahme bzgl. der Evaluierung ist der quote-Operator bzw. dessen Kurzschreibweise ’:

```
> (quote (+ 3 5))
(+ 3 5)
```

```
> '(+ 3 5)
(+ 3 5)
```

Neben Zahlen (Integer) und Strings wie "Lisp" kennt Lisp noch zwei spezielle Datentypen: *Symbole* und *Listen*

```
> 'Ein-Symbol
EIN-SYMBOL
```

```
> '(künstliche intelligenz)
(KüNSTLICHE INTELLIGENZ)
```

→ Das Quote verhindert die Evaluierung von Atomen und Ausdrücken.

Listen und Listenoperatoren (list, append)

Listen können mit Hilfe der Funktion `list` erzeugt werden:

```
> (list 'LISP 'FÜR 'ALLE)
(LISP FüR ALLE)
```

Da `list` eine Funktion ist, werden alle Argumente evaluiert:

```
> (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)
```

→ Ein Lisp-Programm wird selbst als Liste dargestellt. D. h. Lisp-Programme können Lisp-Code generieren.

`append` gibt die Aneinanderreihung beliebig vieler Listen zurück:

```
> (append '(a b) '(c d) '(e f))
(A B C D E F)
```

Listen und Listenoperatoren (cons, car, cdr)

Die leere Liste kann wie folgt ausgedrückt werden:

```
> ()              > nil
NIL               NIL
```

Die Funktion `cons` erzeugt eine Liste. Ist das zweite Argument eine Liste, so wird eine neue Liste zurückgegeben mit dem ersten Argument als erstes Element:

```
> (cons 'a '(b c d))          > (cons 'a (cons 'b nil))
(A B C D)                     (A B)
```

Die primitive Funktionen zur Extraktion von Elementen einer Liste sind `car` und `cdr`. `car` bzgl. einer Liste bezieht sich auf das erste Element und `cdr` auf alles was nach dem ersten Element in der Liste steht (vgl. auch `first`, `rest`, `second`):

```
> (car '(a b c))  > (cdr '(a b c))  > (car (cdr (cdr '(a b c d))))
A                 (B C)             C
```

Die Wahrheit

Das Symbol `t` ist in Lisp die Repräsentation für "wahr". Wie `nil` wird es zu sich selbst evaluiert. Die Funktion `listp` liefert "wahr', falls ihr Argument eine Liste ist:

```
> (listp '(a b c))
T
```

Solche als Prädikate bez. Funktion haben oftmals Namen mit dem Buchst. `p` am Schluss. "Falsch" wird durch `nil` repräsentiert:

```
> (listp 27)
NIL
```

Das Prädikat `null` testet auf die leere Liste und `not` negiert den Wahrheitswert:

```
> (null nil)        > (not nil)
T                   T
```

Konditionale (if, and, or)

Der "Dann"-Teil eines if's ist optional:

```
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))
3
```

```
> (if (listp 27)
      (+ 2 3))
NIL
```

Alles außer `nil` gilt als wahr:

```
> (if 27 1 2)
1
```

Die logischen Operatoren `and` und `or` ähneln Konditionalen. Sie akzeptieren eine beliebige Anzahl von Argumenten und evaluieren soweit wie nötig. Sind alle Argumente wahr, dann gibt `and` den Wert des letzten Arguments zurück:

```
> (and t (+ 1 2))
3
```

[Ausnahme der eval-loop → Makro]

Konditionale (when, cond)

Alle Konditionale bauen auf `if` auf.

```
(when (= 2 3)          entspricht   (if (= 2 3)
   t)                                  t)
```

Mit Hilfe von `cond` können mehrere Bedingungen abgeprüft werden. Überprüfung von oben nach unten bis einr Bedingung zutrifft (`t` gilt immer).

```
(if (< 2 3)            entspricht   (cond ((< 2 3) 2)
   2                                      ((> 2 3) 3)
   (if (> 2 3)                            (t nil))
      3
      nil))
```

Funktionen (defun)

Drei oder mehr Argumente: Name, Parameter und Rumpf

```
> (defun our-third (x)
    (car (cdr (cdr x))))
OUR-THIRD
```

```
> (our-third '(a b c d))
C
```

Variable: Bezeichner mit best. Bindung/Wert.
Im Gegensatzt zu ge-quoteten Symbolen oder Listen (eine nicht ge-quotete Liste wird als Code interpretiert)

Keine Unterscheidung zw. Programm, Prozedur und Funktion.

Vorteil des interaktiven Ansatzes: Ein Programm kann Stück-für-Stück getestet werden (z. B. per Trace).

Rekursion

```
> (defun our-member (obj lst)
    (if (null lst)
        nil
        (if (eql (car lst) obj)
            lst
            (our-member obj (cdr lst)))))

> (trace our-member)

> (our-member 'b '(a b c))
 0[1]: (OUR-MEMBER B (A B C))
   1[1]: (OUR-MEMBER B (B C))
   1[1]: returned (B C)
 0[1]: returned (B C)
(B C)
```

Parameterlisten

&rest
Z. B. `(defun test1 (x &rest args) ...)`
Bindet alle Argumente (außer dem Ersten) an `args`

&optional
Z. B. `(defun test2 (x &optional opt) ...)`
Bindet ein optionales Argument an `opt` (default `nil`)

&key
Z. B. `(defun test3 (x &key y) ...)`
Flexiblere Möglichkeit bzgl. optionaler Parameter. Identifikation des Arguments nicht über die Position, sondern über dessen symbolischen Bezeichner.
Aufruf z. B.: `(test3 'a :x 'b)`
Default ist `nil`, kann aber mit einem Wert vorbelegt werden:
`(defun test4 (x &key (y 'c)) ...)`

Ausgaben (format)

Format (~% erzwingt einen Zeilenumbruch)

```
> (format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL
```

Erstes Argument: Ausgabeort
Zweites Argument: String
Rest: Objekte, die zur Darstellung des Strings benötigt werden (siehe ~)

```
> (defun pluszwei (a)
    (format t "~s plus 2 ist ~s" a (+ a 2)))

> (pluszwei 3)
3 plus 2 ist 5
```

Lokale Variablen (let)

```
> (let ((x 1) (y 2))
    (+ x y))
3
```

Variablendeklaration: ((Variable Ausdruck) ...)

Rumpf von Ausdrücken: Der Wert des letzten Ausdrucks wird zurückgegeben

```
(defun ask-number ()
  (format t "Zahl eingeben: ")
  (let ((val (read)))
    (if (numberp val)
        val
        (ask-number))))
```

```
> (ask-number)
Zahl eingeben: a
Zahl eingeben: (ho hum)
Zahl eingeben: 52
52
```

Globale Variablen (defparameter, defconstant)

Definition globaler Variablen:

```
> (defparameter *glob* 99)     > (boundp '*glob*)
*GLOB*                         T
```

Üblicherweise werden diese mit einem führenden und abschließenden "*" gekennzeichnet.

Definition globaler Konstanten:

```
> (defconstant limit (+ *glob* 1))
LIMIT
```

Jeweils überall gültig (mit der Ausnahme, dass globale Variablen lokal überdeckt werden dürfen).

Zuweisungen (setf)

Allgemeiner Zuweisungsoperator: `setf` (`setq`)

```
> (setf *glob* 98)
98
> (let ((n 10))
    (setf n 2)
    n)
2
```

Ist das erste Argument ein Symbol, welches keine lokale Variable bezeichnet, dann bezieht sich `setf` auf eine globale Variable.

```
> (setf x (list 'a 'b 'c))
(A B C)
```

$\longrightarrow$ Implizite Erzeugung globaler Variablen

Funktionales Programmieren

Funktionale Programme arbeiten nach dem Prinzip der Werte-Rückgabe anstatt Objekte zu modifizieren.
Das "interessante" an Lisp Funktionen ist das was sie zurückgeben, nicht das was – als Seiteneffekt – verändert wird.

Beispiel: `remove`

```
> (setf lst '(c a r a t))
(C A R A T)
```

```
> (remove 'a lst)
(C R T)
```

Die Ursprungsliste wurde nicht verändert:

```
> lst
(C A R A T)
```

Was, wenn aber gerade diese verändert werden soll?

```
(setf x (remove 'a x))
```

Iterieren (dolist)

Operator für Iterationen über die Elemente einer Liste:

```
(defun our-length (lst)
  (let ((len 0))
    (dolist (obj lst len)
      (setf len (+ len 1)))))
```

Die Argumentliste ist von der Art (variable expression expression). Der Rumpf wird der Reihe nach jeweils mit einer Bindung der Variable an ein Element der Liste (zweites Arg.) ausgeführt. Das dritte Argument bestimmt den Rückgabewert des Konstrukts nach der Iteration.

Funktionen als Objekte (function, apply, funcall)

In Lisp sind Funktionen reguläre Objekte wie Symbole oder Strings.
Die Funktion `function` gibt z. B. solch ein Funktionsobjekt zurück:

```
> (function +)
#<Function +>
```

Als Abkürzung für `function` kann `#'` verwendet werden.

Wenn Funktionen reguläre Objekte sind können sie auch als Argumente für andere Funktionen hergenommen werden:

```
> (apply #'+ '(1 2 3))
6
```

```
> (funcall #'+ 1 2 3)
6
```

(Unterschiede hier nur in der Art der Argumentdarstellung)

Lambda-Ausdrücke

Das `defun`-Makro gibt Funktionen einen Namen. Es gibt in Lisp aber auch Funktionen ohne Namen.
Anonyme Funktionen werden durch Lambda-Ausdrücke ausgedrückt. Ein Lambda-Ausdruck ist eine Liste die das Symbol `lambda`, eine Liste von Parametern und einem Rumpf mit einem oder mehreren Ausdrücken enthält.

```
> (lambda (x y)
    (+ x y))
#<Interpreted Function (unnamed) @ #x71c6d31a>
```

Ein Lambda-Ausdruck kann wie eine Funktion angewendet werden:

```
> ((lambda (x) (+ x 100)) 1)
101
> (funcall #'(lambda (x) (+ x 100)) 1)
101
```

Sequenzfunktionen (remove-if)

Sequenzfunktionen erwarten als erstes Argumente ein Prädikat und als zweites eine Liste. Z. B.:

```
> (remove-if #'oddp '(1 2 3 4 5 6))
(2 4 6)
```

(Das Prädikat oddp liefert T, gdw. dessen Argument ungradzahlig ist)

Interessant insb. auch im Zusammenhang mit Lambdafunktionen:

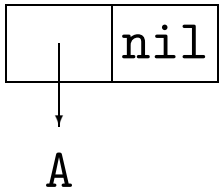
```
> (setf zweier (lambda (x) (= (mod x 2) 0)))
#<Interpreted Function (unnamed) @ #x71bf2482>

> (remove-if zweier '(1 2 3 4 5 6))
(1 3 5)
```

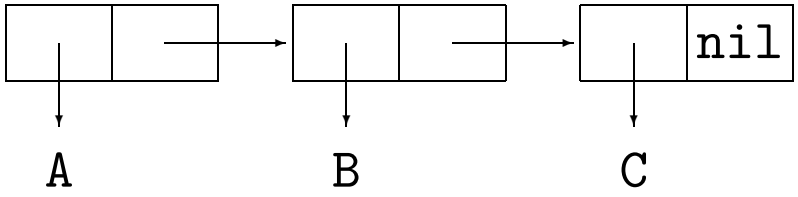
Listen im Detail

Die Anwendung von `cons` kombiniert zwei Objekte in eine Art Tupel-Objekt (genannt *cons-cell*). Konzeptuell besteht dieses Tupel aus zwei Referenzen (die Erste lässt sich durch `car`, die Zweite durch `cdr` auflösen)

```
> (setf x (cons 'a nil)
(A)
```

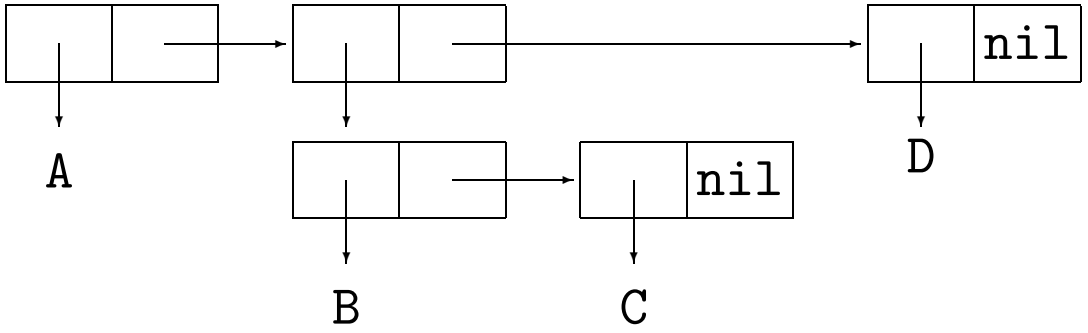



```
> (setf y (cons 'a (list 'b 'c)))
(A B C)
```

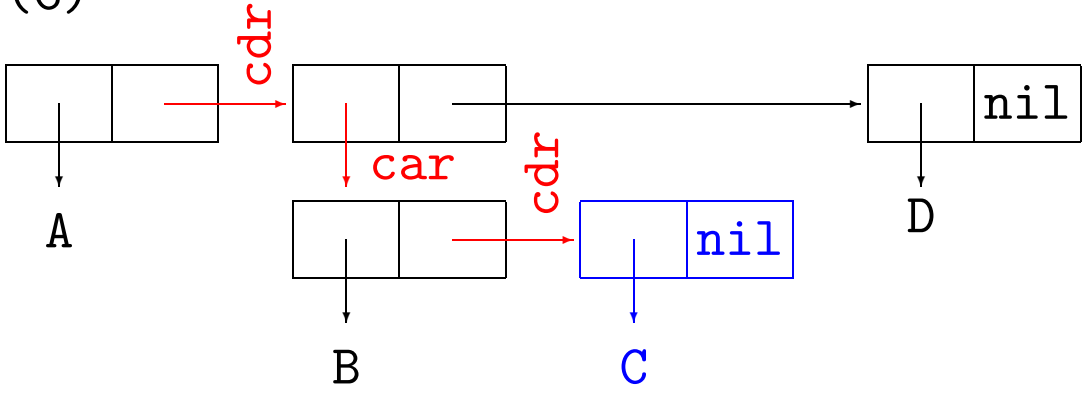


Verschachtelte Listen

```
> (setf z (list 'a (list 'b 'c) 'd))
(A (B C) D)
```




```
> (cdr (car (cdr z)))
(C)
```



Gleichheit (eql, equal)

Jeder Aufruf von `cons` reserviert neuen Speicher für zwei Referenzen. Beim zweifachen Aufruf von `cons` mit den gleichen Argumenten entstehen zwei unterschiedliche Objekte:

```
> (eql (cons 'a nil) (cons 'a nil))
nil
```

→ `eql` testet, ob zwei Objekte identisch sind.

```
> (equal (cons 'a nil) (cons 'a nil))
T
```

→ `equal` testet, ob zwei Objekte die gleichen Elemente enthält.

Punkt-Listen

Die Listen die sich mit `list` erstellen lassen, nennen sich "ordentliche Listen" (*proper lists*).
Mit `cons` lassen sich aber auch sog. Punkt-Listen (*dotted lists*) erstellen. Für jede Struktur die zwei Felder hat lassen sich cons-Zellen verwenden:

```
> (setf pair (cons 'a 'b))
(A . B)
```

Solche Listen werden mit einem Punkt zw. car und cdr dargestellt.

Bem.: Auch ordentliche Listen lassen sich in Punkt-Form darstellen bzw. mischen:

```
> '(a . (b . (c . nil)))
(A B C)
```

```
> (cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```

Blöcke (progn, return-from)

Eine (oftmals benötigte) Art einen Block von Ausdrücken zu bilden ist `progn`:

```
> (progn (format t "a")
         (format t "b")
         (+ 11 12))
ab
23
```

Um einen Block vorzeitig abzubrechen gibt es `return-from`. Hierbei wird als erstes Argument angegeben aus welchem Block (z. B. bei verschachtelten Blöcken) man aussteigen möchte.

```
(defun foo ()
  (return-from foo 27))
```

Bem.: Auch Funktionen sind Blöcke.

Strukturen (defstruct)

Strukturen sind Einheiten mit bestimmten Feldern:

```
> (defstruct point
    x y)
POINT
```

Diese Deklaration definiert implizit die Funktionen `make-point`, `point-p`, `copy-point`, `point-x`, `point-y`.

```
> (setf p (make-point :x 0 :y 0))
#S(POINT :X 0 :Y 0)
```

Die Zugriffsfunktionen für `point` erlauben sowohl lesenden als auch schreibenden Zugriff:

```
> (setf (point-y p) 2)
2
```

```
> p
#S(POINT :X 0 :Y 2)
```