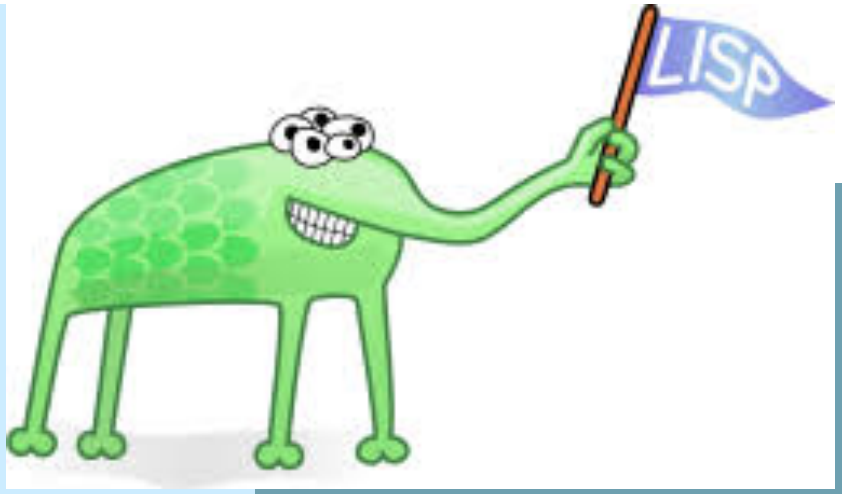
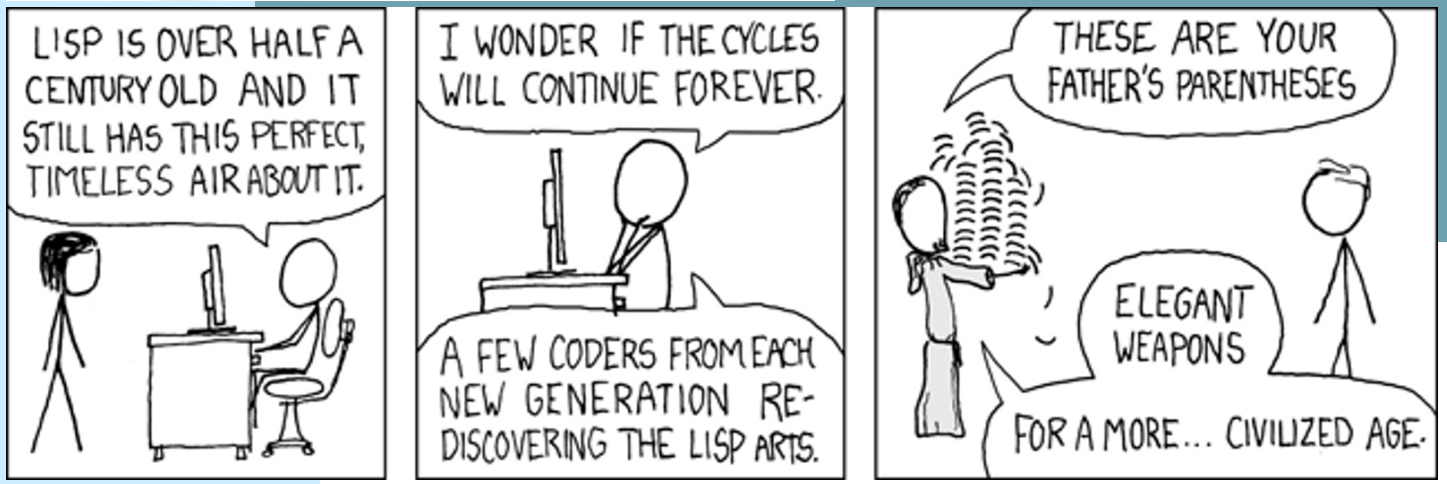
5. Kapitel



LISP



Programmieren 3

Prof. Dr. Wolfgang Schramm

Übersicht

Lernziele des Kapitels

- Grundprinzipen von Lisp kennenlernen.
- Typische Eigenschaften von Lisp verstehen und nutzen können.
- Arbeitsweise des Lisp-Interpreters verstehen.

LISP – Überblick wichtige Eigenschaften

- Entstanden Ende der 1950-er Jahre
- ListProcessor, entwickelt von McCarthy am MIT
- Theoretische Grundlage: Lambda-Kalkül
- Verschiedene Versionen → manchmal leicht unterschiedliche Funktionsnamen
- Extrem einfache Syntax (Präfix-Form) → einfaches Parsen
- Listenorientierte Sprache → Verarbeitung symbolischer Informationen

  Symbolverarbeitung = Umgang mit Begriffen und ihren Beziehungen untereinander (im Unterschied zu numerischer Verarbeitung).

LISP – Überblick wichtige Eigenschaften

- Viele wichtige Programmiersprachen-Ideen erstmalig in Lisp, z.B
  - Garbage Collection
  - Rekursion
  - Quellcode-Tracing und Debugging
- Funktionale Programmiersprache, aber nicht (mehr) völlig funktional: enthält heute z.B.
  - Zuweisungen
  - Kontrollfluss (Schleifen, ...)
- Bekanntester Standard: Common Lisp (mit hunderten von vordefinierten Funktionen), 1984 (1994)
- Erweiterungen von Lisp
  - CLOS (Common Lisp Object System)
  - CLIM (Common Lisp Interface Manager

Lisp Syntax

Lisp-Programme sind Listen!

Listen

- dienen zur Beschreibung satzähnlicher Objekte.
- bestehen aus einer öffnenden Klammer, gefolgt von beliebig vielen Listenelementen, jeweils durch ein Leerzeichen getrennt, und einer schließenden Klammer.

Listenelement

- ist entweder ein Atom, oder wiederum eine Liste

Lisp-Syntax - Beispiele


```
(Dies ist eine Liste) ((das) (auch noch) dazu)
```

```
(defun print-quadratzahlen (x)
   (when (plusp x)
     (print (* x x))
      (print-quadratzahlen (- x 1))))
```

Beide Beispiele sind syntaktisch korrekt, das erste ist semantisch falsch.

Lisp Semantik

Der Kern eines Lisp-Systems ist der **Interpreter**, dessen Aufgabe es ist, eingegebene symbolische Ausdrücke auszuwerten (zu evaluieren).

Das Ergebnis ist ebenfalls ein symbolischer Ausdruck.

Das Ergebnis wird nach der Evaluation ausgegeben.

Der Interpreter ist in einer sog. Read-Eval-Print-Loop (REPL) eingebettet.

Interpretation

- Lisp ist zunächst eine interpretative Sprache, d.h. jeder Ausdruck (jede Form) erhält durch Interpretation einen Wert.
- Die oberste Ebene des Lisp-Systems ist ein Interpreter für Eingaben. Der Interpreter ist realisiert als eine “read-eval-print-Schleife”:
  - Ein Ausdruck wird mit `read` eingelesen. `read` verwandelt die externe Repräsentation in eine interne Repräsentation .
  - mit `eval` ausgewertet.
  - mit `print` wird die interne Repräsentation des Ergebnisses ausgegeben.
  - Bei Fehler → Aktivierung eines sogen. break-level (wird i.d.R. Benutzer in der durch eine Zahl (≥ 1), die vor dem Prompt erscheint, signalisiert. Innerhalb des break-levels hat man Zugriff auf die aktuelle Bindungsumgebung und kann so die, das Programm direkt modifizieren und Fehler schnell beseitigen.
- Wir beschäftigen uns im folgenden damit, welche Ausdrücke man in das Lisp-System eingeben kann, und wie diese ausgewertet werden.

Lisp Interpreter

Der Interpreter arbeitet nach 3 Regeln:

- Identität
  - Eine Zahl, eine Zeichenfolge, oder die Symbole `t` und `nil` werden zu sich selbst evaluiert.
- Listen
  - Jede Liste stellt einen Funktionsaufruf dar.
  - Das erste Element der Liste bezeichnet die anzuwendende Funktion.
  - Die weiteren Elemente der Liste (wenn vorhanden), sind die Argumente der Funktion (in Präfix-Notation).
  - Die Argumente werden (“normalerweise“) vor der der Anwendung der bezeichneten Funktion ausgewertet.
- Symbole
  - Die Auswertung eines Symbols liefert den mit dem Symbol assoziierten Wert zurück.

Elemente von Lisp

- Atome
  - Elementare Lisp-Ausdrücke
    - Zahlen
    - Zeichen
    - Strings
  - Symbole
  - Funktionen
  - Arrays
  - Strukturen
  - Streams
- Conses (früher Cons cell) - Listen
  - Paarlisten
  - Listen
    - einfache Listen
    - Assoziationslisten
    - Property-Listen

Atom

- Elementarer Lisp-Ausdruck
  - Zahl
    - Integer: 3 oder #b1101 (= 13_{10}) oder #xD5 (= 213_{10}) oder #12rA1 (=121_{10})
    - Float: 123.45 oder 3.0E-2; Double: 3.0D2
    - Rational: Bruch zweier Integer -17/30 (Bruch bleibt erhalten)
    - Komplexe #C(2 3) = 2+3i
  - Zeichen
    - #\<Zeichen>
  - Zeichenkette (String): “<Zeichenkette>“

Symbole

- Symbol (man kann auch sagen: Variablen, denen ein Wert zugewiesen werden kann)
  - beliebige Kombination Zahlen/Zeichen, beginnend mit einem Buchstaben
  - stellen „Variablen“, Funktionsnamen oder Prädikate dar
  - haben Wert (Konstante, Funktionsergebnis, Wahrheitswert)
  - Beispiele: X11, ABS, Montag-22-10-17, VariablenSolltenImmerBedeutungsvolleNamenHaben
  - T ≙ true, NIL ≙ false (auch: () – leere Liste)

Evaluation von S-Expressions

- LISP-Datenstrukturen werden als S-Expressions (symbolic expressions) bezeichnet. Es sind nur eine geringe Zahl einfacher semantischer Regeln erforderlich um festzulegen, wie eine S-Expression zu interpretieren (evaluieren) ist.

- Zahlen, Zeichen und Zeichenketten (strings) werden als selbstevaluierende Objekte bezeichnet. Die Evaluierung von S-Expressions dieses Typs, wird durch Regel 1 bestimmt:

Regel 1: Zahlen, Zeichen und Zeichenketten evaluieren zu sich selbst.

Operationelle Semantik (Atome) 1/2

Was ist der Wert eines Atoms?

- Atome, die keine Symbole sind, evaluieren zu sich selbst.

- Beispiele:
  - Auswertung von Konstanten

```
1234567890       ⟾   1234567890
6/8              ⟾   3/4
12.23456678      ⟾   12.23456678
"hello, world”   ⟾   "hello, world"
```

Operationelle Semantik (Symbole) 1/3

- LISP-Symbole sind komplexe Objekte, die in Common LISP Implementationen aus mindestens den folgenden 5 Teilen bestehen:
  - print-name
  - value-binding
  - function-binding
  - property-list
  - package

- Symbolen können Werte zugewiesen werden, die abhängig von der Art des Wertes und der Form der Wertzuweisung in dem mit value-binding, function-binding (auch: Wertbindung bzw. Funktions-bindung) bzw. property-list bezeichneten Teil des Symbols gespeichert werden.

Operationelle Semantik (Symbole) 2/3

- Zunächst Beschränkung auf die Wert- und Funktionsbindung von Symbolen.
  - Wertbindung. Da es möglich ist, Symbolen Werte zuzuweisen, können sie als Variablen verwendet werden. Als Werte für Symbole sind beliebige LISP-Objekte zulässig. (Beispiel: der Wert eines Symbols kann ein anderes Symbol sein).
- Einige Symbole haben eine vordefinierte Bedeutung, die durch den Benutzer nicht verändert werden sollte (System-Konstanten). Die beiden wichtigsten Symbole dieses Typs sind:
  - T bezeichnet den booleschen Wert True;
  - NIL bezeichnet den booleschen Wert False (und gleichzeitig auch die leere Liste).
- Die Evaluation eines Symbols wird durch den Kontext gesteuert, in dem es vorkommt.

Regel 2: Wenn keine andere Regel anwendbar ist, dann evaluiert ein Symbol zu einer Wertbindung (falls vorher einer zugewiesen wurde).

Operationelle Semantik (Symbole) 3/3

- Auswertung von Symbolen

```
pi              ⟼   3.141592653589793D0
x               ⟼   Error: The variable X is unbound.
```

Um ein Symbol auszuwerten, muss zuerst ein Wert zugewiesen werden:

```
(setf x 4)      ⟼   4
x               ⟼   4
```

Nach der `setf` Anweisung ist das Symbol x an den Wert 4 gebunden; x ist eine globale Variable.

Listen

- Listen (Reihen von Elementen) dienen zur Beschreibung satzähnlicher Objekte.
- Liste =
  - Datenobjekte, Funktionsaufrufe, Makros, spezielle Ausdrücke, Programmcode.
  - Durch ( ) geklammert, beliebig viele Listenelemente durch Leerstelle getrennt.
- Ein Listenelement ist entweder ein Atom o der wiederum eine Liste.
- Beispiele:

```
( 3 6 2 9 )                 ; einfache Liste
( (a b) (c d))              ; geschachtelte Liste
( a (b c) ( (1 2) 3) )      ; beliebige Schachtelung möglich
() oder NIL                 ; leere Liste
```

Funktionen

- Programmteile werden durch Listen (“Formen”) repräsentiert.
- Einheitliche Syntax in Präfix-Notation und Funktionskomposition durch Verschachtelung von Listen.

- Funktionen = Operation mit Parametern
  - Große Menge an vordefinierten Funktionen,
  - haben immer Präfix-Notation,
  - Aufbau: `(funktions_name par_1 ... par_n).`

Operationelle Semantik (Listen) 1/2

- Was ist der Wert einer Liste (Funktion)?
- Das erste Element einer Liste wird i.allg. als Funktionsname interpretiert.
- Gibt es keine Funktion dieses Namens, so signalisiert der Interpreter einen Fehler.
- Die restlichen Elemente der Liste werden von links nach rechts evaluiert, um als Argumente der Funktion übergeben zu werden.

Regel 3: Wenn ein Symbol als erstes Element einer zu evaluierenden Liste vorkommt, wird bei der Evaluation der Liste die Funktionsbindung des Symbols verwendet. Die dort gespeicherte Funktion wird auf das Resultat der Evaluierung aller übrigen Ausdrücke der Liste angewendet.

- Bemerkung: In CommonLisp gibt es neben normalen Funktionsanwendungen auch sogenannte Spezialformen und Makros, bei denen die Argumente nicht unbedingt von links nach rechts ausgewertet werden – mehr dazu später.

Operationelle Semantik (Listen) 2/2

- Beispiele:

```
(+ 3 5 9)            ⟼   17
(+ 3 (* 4 5))        ⟼   23
(/ 2 3)              ⟼   2/3
(/ 6 2.0)            ⟼   3.0
(- 50 3 5 10)        ⟼   32
```

Soll Parameter nicht evaluiert werden → ‘ / (quote) voranstellen

```
(quote x)            ⟼   X
‘x                   ⟼   X
‘(+ 3 (* 4 5))       ⟼   (+ 3 (* 4 5))
(first  ‘(a b c))    ⟼   A
```

Listen erzeugen

`nil`	⟼	Erzeugt leere Liste.
`(cons a l)`	⟼	Fügt Element a an den Anfang der Liste.
`(list a1 a2)`	⟼	Konstruiert eine Liste aus den Elementen a1 und a2 (a2 darf fehlen)
`(append l1 l2)`	⟼	Hängt die Listen l1 und l2 zusammen (l2 darf fehlen).

- Beispiele:

```
nil =  '()                              ⟼  NIL
(cons 'a '(b c d))                      ⟼  (A B C D)
(list 1 (+ 1 1) (+ 1 (+ 1 1)))          ⟼  (1 2 3)
(append '(1 2 3) '(4 5 6))              ⟼  (1 2 3 4 5 6)
```

Selektoren auf Listen

- ... trennen Listen auf.
- Die Lisp-typischen Trenner sind CAR und CDR.
  - CAR = Contents of the Address part of Register number
- CDR = Contents of the Decrement part of Register number
- Syntax: `(car liste), (cdr liste)`
- Die Funktion CAR liefert das erste Element von liste; d.h. das Objekt, auf das der CAR-Pointer der ersten CONS-Zelle von Liste weist.

- Die Funktion CDR liefert eine Liste, die alle Elemente von Liste außer dem ersten enthält; d.h. das Objekt, auf das der CDR-Pointer der ersten CONS-Zelle von Liste weist.

Selektoren auf Listen – Beispiele


```
(car nil) ≡ (first nil)                  ⟼    NIL
(car '(a b c)) ≡ (first '(a b c))        ⟼    A
(cdr nil) ≡ (rest nil)                   ⟼    NIL
(cdr '(a b c)) ≡ (rest'(a b c))          ⟼    (B C)
(third '(a b c)) ≡ (caddr'(a b c))       ⟼    C
(nth 2 '(a b c))                         ⟼    C
(last '(a b c)) ≡ (cddr'(a b c))         ⟼    (C)
```

Bemerkung:

Statt `car` kann auch `first` und statt `cdr` auch `rest` verwendet werden.

`car` und `cdr` sind konkatenierbar – z.B. `cadr, cdar, cddddr`

```
(car (cdr liste)) = (cadr liste)
(car (cdr (cdr (cdr liste)))) = (cadddr liste)
```

Die Grenzen dieses kompositionellen Verfahrens sind implementationsspezifisch.

Listen und Funktionsaufrufe

- Die Unterscheidung zwischen Listenkonstruktion und Funktionsaufruf bereitet anfänglich immer Schwierigkeiten.
- Durch **`(f a1 a2)`** wird die **Funktion** f auf die Argumente a1, a2 angewendet. Es wird dadurch nicht die Liste mit Elementen f, a1, a2 konstruiert!
- Durch **`(list expr1 expr2)`** wird eine **Liste** konstruiert, deren Elemente die Werte von expr1, expr2 sind.
- Durch **`'(e1 e2 e3)`** wird eine Liste mit Elementen e1, e2, e3 konstruiert.

Listen und Funktionsaufrufe - Beispiele


```
>(defun f (n1 n2) (* n1 (+ n2 2)))
F
>  (f 3 4)
18
>(f (+ 2 3) 4)
30
>(list 'f (+ 2 3) 4)
(F 5 4)
>'(f (+ 2 3) 4)
(F (+ 2 3) 4)
>'(e1 e2 e3)
(E1 E2 E3)
> (e1 e2 e3)
Undefined operator E1 in form (E1 E2 E3).
```

Prädikate

- = Funktionen, die Wahrheitswert T oder NIL liefern
- vordefinierte Prädikate (eine Auswahl):

```
(eq x y)          ; identisches Objekt (gleiche Speicheradresse)?
(eql x y)         ; gleiche Zahl / identisches Objekt?
(equal x y)       ; Sturkturgleichheit / gleicher Lisp-Ausdruck?
(= x y)           ; gleiche Zahl?
(not x)           ; Negation von x
(atom x)          ; x Atom?
(numberp x)       ;ist x vom Typ number? analog: integerp, floatp,;
                  ; realp, rationalp, complexp, characterp, ; stringp,...
(evenp x)         ; x gerade? analog: oddp
(consp l)         ; ist l eine zusammengesetzte Liste
(null l)          ; ist l die leere Liste
(listp l)         ; ist l eine List e(vom Typ list)?
(member a l)      ; ist a ein Element der Liste l?
```

Prädikate - Beispiele

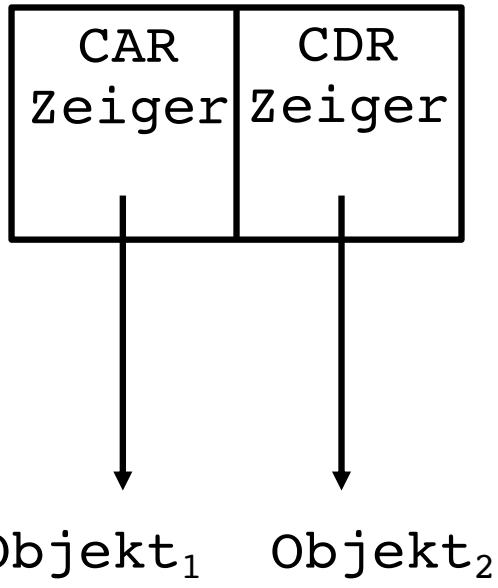

```
(numberp 3)                  ⟹  T
(symbolp 3)                  ⟹  NIL
(evenp 3)                    ⟹  NIL
(evenp t)                    ⟹  error
(eq  ‘(a) ‘(a))              ⟹  NIL
(eq ‘a ‘a)                   ⟹  T
(eql 3 3.0)                  ⟹  NIL
(equal  ‘(a) ‘(a))           ⟹  T
(= 3 3.0)                    ⟹  T
(and (> 3 2) (< 10 9))       ⟹  NIL
(or (> 3 2) (< 10 9))        ⟹  T
(null nil)                   ⟹  T
(null ’(1))                  ⟹  NIL
(member ’a ’(a b c))         ⟹  (A B C)
```

CONSes

Ein CONS ist ein komplexes Objekt:

- wird durch die geordnete Verbindung zweier LISP-Objekte gebildet,
- jedes dieser Objekte kann selbst wieder ein Objekt beliebigen Typs (z.B. wieder ein CONS) sein
- → CONSes bilden eine rekursive Datenstruktur.
- Objekte dieses Typs werden häufig durch sogen. CONS-Zellen repräsentiert, die zwei Zeiger enthalten, wobei der erste auf das erste Objekt und der zweite auf das zweite Objekt weist.
- Zur Erzeugung von CONSes gibt es in LISP die Funktion `CONS`.

CONS-Zellen



Eine cons-Zelle ist ein Paar von Zeigern.

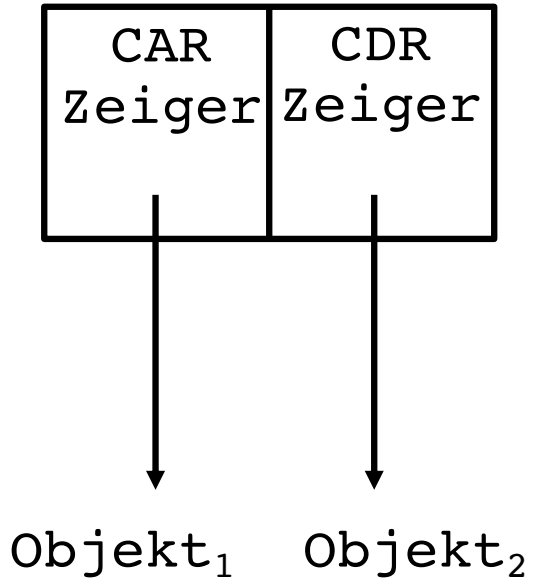
Der erste Zeiger heißt CAR-Zeiger.

Der zweite heißt CDR-Zeiger.

Mit der Funktion `cons` wird eine cons-Zelle allokiert.

Mit car, cdr wird auf die jeweiligen Bestandteile zugegriffen.

CONS-Zellen und Listentypen



- Es gibt zwei Typen von CONSes: **Listen** und **Paarlisten** (dotted pairs).
- Bei Listen: CDR-Zeiger der letzten CONS-Zelle weist immer auf das die leere Liste repräsentierende Symbol NIL.
- Bei den Paarlisten CDR-Zeiger der letzten CONS-Zelle weist auf ein beliebiges LISP-Objekt.
- Listen bilden also einen speziellen Subtyp von Paarlisten - es sind Paarlisten, bei denen der CDR-Zeiger der letzten CONS-Zelle auf NIL weist.

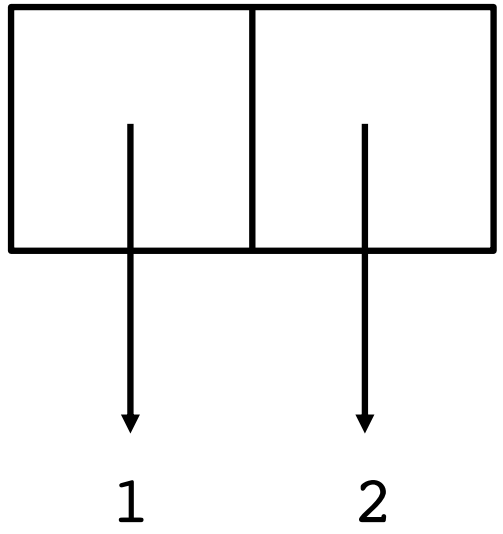
Darstellung von CONSes

- Unterscheidung von Listen und Paarlisten: Paarlisten werden vom System auf dem Bildschirm anders dargestellt als “echte“ Listen.
- Bei der Repräsentation von Paarlisten wird der Punkt "." verwendet, um die beiden Objekte der Paarliste voneinander zu trennen; d.h. der Punkt "." trennt den CAR-Teil vom CDR-Teil einer CONS-Zelle.

CONS-Zelle: Beispiel


```
(setf *c* (cons 1 2)) ⟼ (1 . 2)
(car *c*)             ⟼ 1
(cdr *c*)             ⟼ 2
(consp *c*)           ⟼ T
```

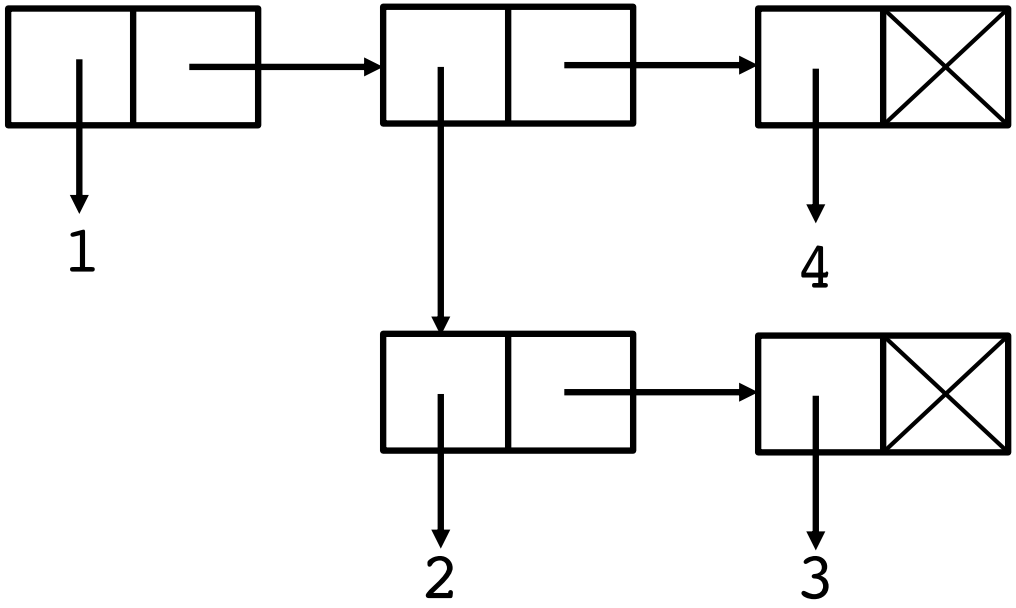
Dotted Pair



Interne Darstellung von Listen als cons-Zellen


```
‘(1 (2 3) 4) ≡
(cons 1 (cons (cons 2 (cons 3 nil)) (cons 4 nil))) ≡
‘(1 . ((2 . (3 . nil)) . (4 . nil)))
```

Alle 3 Darstellungen sind äquivalent.

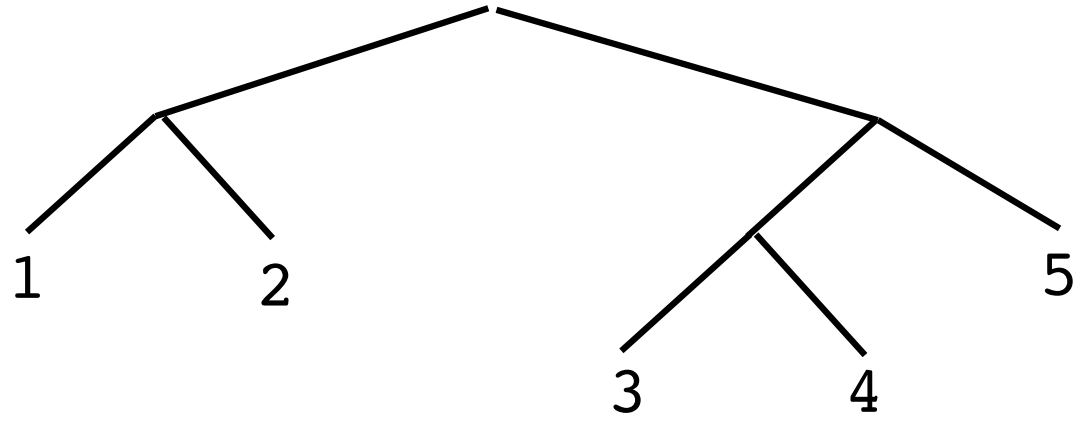


Hinweis:

`‘(a . (b . (c . nil)))` ist eine Liste , an die noch etwas angehängt werden kann.

`‘(a . (b . (c . d)))` ist eine Liste , an die nichts mehr angehängt werden kann.

Darstellung von Bäumen durch Listen



Binärbaum, Werte nur in den Blättern.

```
(cons (cons 1 2) (cons (cons 3 4) 5)) ≡
'((1 . 2) (3 . 4) . 5)
```

Read-Eval-Print Loop (REPL)

- Alle CL-Umgebungen starten automatisch in der sog. Read-Eval-Print Loop.
- Interaktives “front-end” jedes Lisp-Interpreters heißt *top-level* (im Folgenden durch das Zeichen “>” abgekürzt).

  > 1

  1

  >

  →

  Zahlen werden zu sich selbst/ihrem Wert evaluiert.

> (+ 2 3)	> (+ 2 3 4 5)	> (/ (- 7 1) (- 4 2))
5	14	3

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

Ein erstes Lisp-Programm


```
(defparameter *klein* 1)
(defparameter *gross* 100)


(defun errate-meine-zahl ()
   (ash (+ *klein* *gross*) -1))

(defun kleiner ()
   (setf *gross* (1- (errate-meine-zahl)))
   (errate-meine-zahl))

(defun groesser ()
   (setf *klein* (1+ (errate-meine-zahl)))
   (errate-meine-zahl))
```

Evaluierung (Auswertung)

- 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 (LispWorks):

```
CL-USER 38 : 5 >(/ 5 0)
Error: Division-by-zero caused by / of (5 0).
   1 Return a value to use.
   2 Supply new arguments to use.
   3 (abort) Return to level 5.
   .....
   10 Return to debug level 2.
   11 Return to level 1.
   12 Return to debug level 1.
   13 Return to level 0.
   14 Return to top loop level 0

.Type :b for backtrace or :c <option number> to proceed.Type :bug-form
"<subject>" for a bug report template or :? for other options.

CL-USER 39 : 6 >
```

Zusammenfassung

Evaluierung

- Eingaben werden in einer read-eval-print-Schleife interpretiert.
- Zahlen, Zeichen, Zeichenreihen evaluieren zu sich selbst.
- Symbole evaluieren zu dem an sie gebundenen Wert.
- Die Evaluierung eines Ausdrucks wird durch QUOTE verhindert.
- Evaluation komplexer Ausdrücke
  - Zuerst werden der Funktionsname und danach sämtliche Argumente evaluiert,
  - dann wird die Funktion auf die Argumente angewendet.

Listen

- Listen werden durch die Konstruktoren nil und cons gebildet.
- Die zugehörigen Testfunktionen sind null und consp Zugriffsfunktionen sind car (first) und cdr (rest).

Darstellung von Listen

- Listen werden intern durch “Cons-Zellen” dargestellt.
- Gleichheitsfunktionen unterschiedlicher Stärke sind eq, eql und equal