

Einführung in LISP

Simon Lutz, Frank Preiswerk

“Real Programmers don't write in LISP. Only idiots' programs contain more parenthesis than actual code.”

“Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.”

Zeitlicher Überblick



1958: Lisp wurde von John McCarthy am MIT speziell für nicht-numerische Probleme designed und von Steve Russell auf einer IBM 704 Maschine implementiert. Der Name bedeutet “List Processing Language”.

1968: Erster Lisp Compiler geschrieben in Lisp von Tim Hart und Mike Levin.

1968-70: Implementation von SHRDLU mittels Lisp, berühmtes AI-System.

1970er Jahre: Performanz von bestehenden Lisp-Systemen wird zum Thema, es entstehen optimierte Lisp-Maschinen. Heute wieder verschwunden.

1980er – 1990er Jahre: Bemühungen für eine Vereinheitlichung der vielen bestehenden Lisp-Dialekte: Common Lisp entsteht.

1994: ANSI veröffentlicht Common Lisp Standard. Dies zu einem Zeitpunkt, als der Weltmarkt für Lisp viel kleiner war als in den früheren Zeiten.

Lisp heute

Nach einem Rückgang in den 90er Jahren ist das Interesse seit 2000 wieder gestiegen, massgeblich im Rahmen von Open Source Implementationen von Common Lisp.

2004: Peter Seibel's "Practical Common Lisp" ist für kurze Zeit an zweiter Stelle von Amazon's populärsten Programmierbüchern.

April 2006: Tiobe Software rankt Lisp auf Platz 14 der populärsten Programmiersprachen.

Neue "Lisper" beschreiben die Sprache gerne als "eye-opening experience" und behaupten, in Lisp deutlich produktiver zu sein als in anderen Sprachen.

Eigenschaften von Lisp

- Funktionale Programmiersprache
- Expression oriented: Keine Unterscheidung zwischen Expressions und Statements, alles wird in Form von Expressions formuliert.
- Metaprogramming: Lisp-Funktionen sind selbst wieder Listen und können so wie Daten verarbeitet werden. Kein Unterschied zwischen Daten und Programmen. Strukturveränderung des Programms durch Selbstoptimierung, Selbstlernen möglich. Deshalb hat sich Lisp v.a. im Bereich der Künstlichen Intelligenz durchgesetzt.
- Typfreie Sprache (es gibt Typen, diese sind aber optional und erhöhen die Effizienz)

LISP Dialekte

Heute verwendete Dialekte

- Common LISP: Beruht auf ZetaLISP, Franz Lisp und einigen Einflüssen von InterLISP. Ist heute der Industriestandard.
- Scheme: (eine akademische "reine, minimale" Variante, lexical variable scoping und continuations.)
- Emacs Lisp: Die Skript-Sprache des Emacs-Editors

Andere Lisp-Varianten

- MacLisp
- InterLisp
- AutoLISP
- muLISP
- ISLisp
- EuLisp

Wir behandeln im Folgenden ausschliesslich Common LISP.

Darstellung von Listenstrukturen

Syntaktische Darstellung (1)

Daten und Programme in LISP werden als “*S-Expression*” (symbolischer Ausdruck) geschrieben. Eine S-Expression ist z.B. Ein “*Atom*” oder eine “*Liste*”.

Ein Atom ist entweder ein “*literales Atom*” oder eine “*Zahl*”:

A	= literales Atom
FOO	= literales Atom
1FOOISNOTBAR	= literales Atom, da keine Zahl
2	= ganze Zahl (Integer)
3.1416	= dezimale Zahl (Real)

Darstellung von Listenstrukturen

Syntaktische Darstellung (2)

Als Liste kann beispielsweise der Inhalt einer Schüssel geschrieben werden:

(APFEL BANANE CITRONE) oder abstrakt: (A B C)

Eine Liste ist entweder leer oder enthält Atome und/oder Sublisten, z.B.:

() \equiv NIL = leere Liste, ist mit dem Atom NIL
identisch

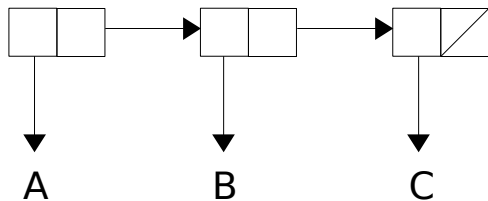
(D (E F) G) = Liste mit Atom D, der Subliste (E F) und
dem Atom G

Darstellung von Listenstrukturen

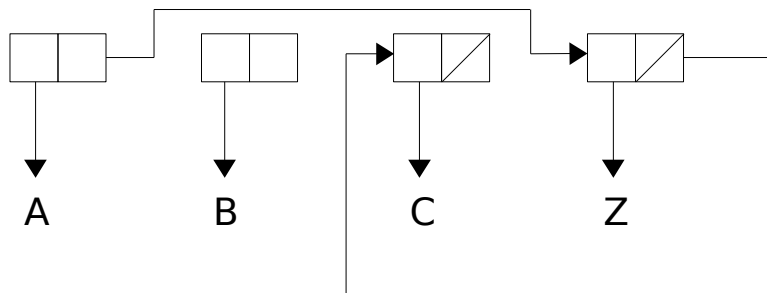
Interne Darstellung (1)

Man kann sich ein Listenelement als aus einer Doppelzelle (cond) bestehend vorstellen, die zwei Zeiger enthält. Der linke Zeiger (*car*) verweist auf den Inhalt, der rechte Zeiger (*cdr*) verweist auf das nächste Listenelement.

Beispiel einer lineare Liste:



Änderungen der Listenstruktur können durch Änderung der Zeiger vollzogen werden:

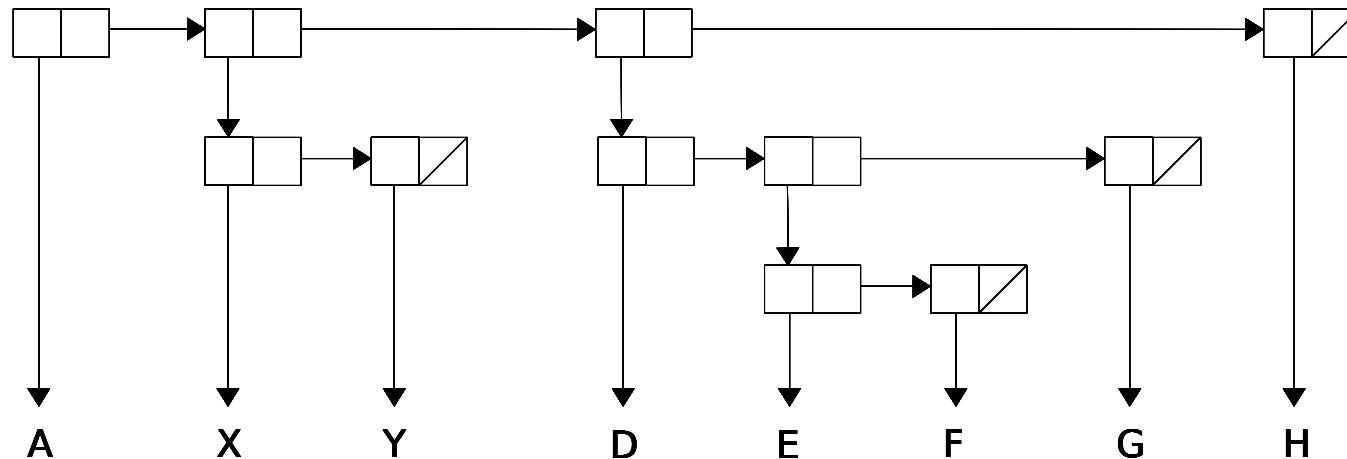


Darstellung von Listenstrukturen

Interne Darstellung (2)

Beispiel einer hierarchischen Liste:

(A (X Y) (D (E F) G) H)



Forms

Forms bilden die Grundeinheit für die Interaktion mit einem LISP-System.
Alle Forms können evaluiert werden und liefern einen oder mehrere Werte.

Forms können in folgende Kategorien aufgeteilt werden:

- Selbstevaluierende Forms z.B. Zahlen
- Symbole stehen für Variablen
- Listen

Listen können weiter unterteilt werden in:

- Function Calls z.B. Arithmetische Operationen
- Special Forms grundlegende Kontrollstrukturen wie `if`
- Macro Calls erweiterte Kontrollstrukturen

Funktionen

Funktionen werden in Listen geschrieben, dabei ist das erste Element der Funktionsname, alle weiteren Elemente sind die Argumente.

Funktionen am Beispiel von Arithmetische Operationen

Arithmetische Operationen werden in Präfix-Notation geschrieben:

```
(+ 1 2 3 4) // gibt 10 zurück
```

Das Äquivalent in Infix-Notation ist $1 + 2 + 3 + 4$

Arithmetische Operatoren in Lisp sind variadisch, d.h. ihre Anzahl an Argumenten ist unbeschränkt.

Schachtelungen werden von innen nach aussen evaluiert, z.B.:

```
(* (+ 5 (+ 1 3)) (- 0 2)) // gibt -18 zurück
```

Special forms

Special forms sehen aus wie Funktionsaufrufe, sind aber in Wirklichkeit Bestandteile der Sprachsyntax. Beispiele sind Kontrollstrukturen wie `if`-Statements und `do`-Loops; Zuweisungen wie `setq`, `setf`, `push` und `pop`, Definitionen wie `defun` und `defstruct`.

```
(if nil
    (list 1 2 "foo")
    (list 3 4 "bar"))           // gibt (3 4 "bar") zurück
```

Der Special Operator `if` akzeptiert 3 Argumente. Wenn das erste Argument nicht `nil` ist, wird das zweite Argument evaluiert, ansonsten das dritte.

Weiteres Beispiel: Loop

```
(setf x 0)
(loop
  (setf x (+ x 1))
  (when (> x 7) (return x))) // gibt 8 zurück
```

Macros

Beispiel: `cond`

```
(cond
  (präd1 expr11 expr12 ...) // Klausel 1
  (präd2 expr21 expr22 ...) // Klausel 2
  ...
  (prädn exprn1 exprn2 ...) ) // Klausel n
```

Es wird nacheinander jedes Prädikat getestet, bis ein Prädikat zu ungleich `nil` evaluiert (z.B. `präd2`). Dann werden nacheinander die zu dieser Klausel gehörenden S-Expressions (`expr21 ... expr2m`) abgearbeitet und anschliessend die `cond` Funktion mit dem Wert der letzten Expression (`expr2m`) verlassen.

Evaluiert kein Prädikat ungleich `nil`, so “fällt” `cond` durch und hat den Wert `nil`.

Wertbindung

Mit der Funktion `setf` kann eine Wertbindung an eine Variable bewirkt werden (obsolete `setq`):

```
(setf A 5)           // A = 5  
(setf B 3)          // B = 3
```

Man kann in weiteren Anweisungen darauf zurückgreifen, z.B.:

```
A                // gibt 5 zurück  
(+ A B)          // gibt 8 zurück
```

Zuweisung einer Liste:

```
(setf LISTE1 '(A B C))
```

Wichtig: Das Hochkomma verhindert dabei die Evaluierung der Liste.
Äquivalent wäre:

```
(setf LISTE1 (quote ( A B C))) // andere Schreibweise
```

Wertbindung

Als Beispiel der Ambivalenz von Daten und Programmcode in LISP soll folgendes Beispiel dienen:

```
(setf A 5)  
(setf B 3)
```

```
(setf SUM (+ A B))           // SUM = 8
```

```
(setf SUM `(+ A B))         // SUM = (+ A B)
```

Bemerkenswerte Eigenschaft, dass die hier als Daten an `SUM` gebundene Liste auch als “Programm” interpretierbar ist.

Mit der Funktion `eval` kann `SUM` ausgeführt werden:

```
(eval SUM)                   // gibt 8 zurück
```

Elementarfunktionen zur Listenverarbeitung

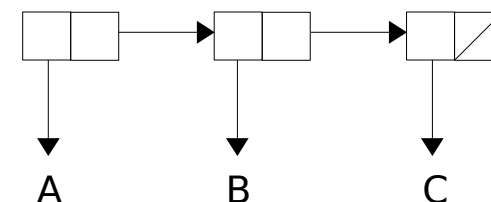
Ein `cons` (construct) ist die Datenstruktur, aus denen Listen aufgebaut werden. Ein `cons` enthält zwei Komponenten, `car` (zeigt auf den Inhalt des `cons`) und `cdr` (wird als “kuder” ausgesprochen, zeigt auf das nächste `cons`).

Mit der Funktion `cons` kann eine Liste gebildet oder erweitert werden, z.B.:

```
(cons 'A nil)           // (A)
(cons 'A '(B C))       // (A B C)
(cons '* '(A B C))    // (* A B C)
```

Die Funktion `car` liefert das erste Element einer Liste, die Funktion `cdr` liefert die Restliste:

```
(car '(A B C))         // A
(cdr '(A B C))        // (B C)
```



Kurzformen

```
(caddr x) = (car (cdr (cdr x)))
(caadar x) = (car (car (cdr (car x))))
```

Höhere Funktionen zur Listenverarbeitung

Es gibt auch höhere Funktionen zur Listenverarbeitung wie z.B.:

```
(reverse '(A C))           // (C B A)
(sort '(5 2 4 1 3) #'<)   // ( 1 2 3 4 5)
(remove 'B '(A B C))      // (A C)
(subset 'A 'B '(A B C))   // (A A C)
(union '(A B C) '(B C D)) // (A B C D)
(intersection '(A B C) '(B C D)) // (B C)
(nth 1 '(A B C))          // (B)
(last '(A B C))           // (C)
```

Property-Listen

Es können bestimmte Eigenschaften eines Atoms auf einer Property-Liste gespeichert und bei Bedarf abgerufen werden:

```
(setf (get `apfel `farbe) `(gruen))  
(setf `quader `dimension `(2 6 4))
```

Die Eigenschaften können mit der Funktion `get` abgerufen werden:

```
(get `apfel `farbe)           // GRÜN  
(get `quader `dimension)     // (2 4 6)
```

Mit der Funktion `remprop` können Eigenschaften gelöscht werden:

```
(remprop `apfel `farbe)
```

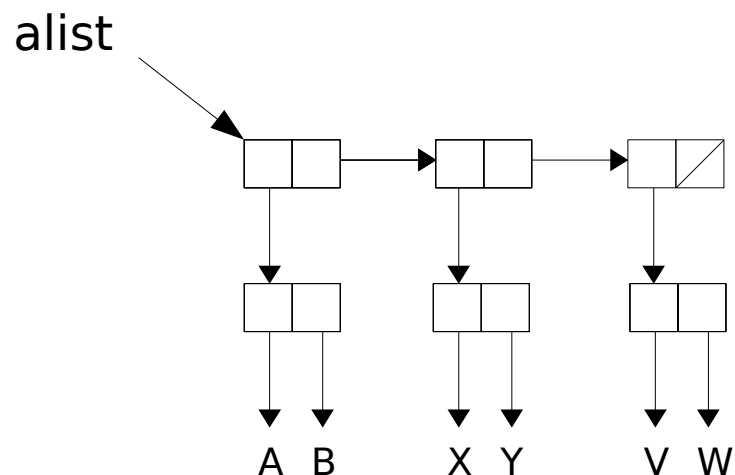
Assoziations-Listen

Assoziations-Listen sind Punkt-Paare folgender Form:

```
( (key1 . wert1)(key2 . wert2) ... (keyn . wertn) )
```

Wobei `key1 ... keyn` (z.B. atomare) Schlüssel, `wert1 .. wertn` beliebige S-Expressions sind.

```
(setf alist `( (A . B) (X . Y) (V . W) ) )
```



Zugriff:

```
(car alist)           // (A . B)  
(car (car (alist)))  // A  
(cdr (car (alist)))  // B
```

Prädikatfunktionen

Prädikatfunktionen sind Funktionen, die einen Wahrheitswert \mathbb{T} (für true) oder `nil` (für false) zurückliefern. Einige Beispiele mit unserer `listel = (A B C)`

```
(null () )           // T
(null listel)        // nil
(zerop 0)           // T
(atom (car listel)) // T
(equal `A (car LISTE1)) // T
(member `D LISTE1)   // nil
(member `D `(A B C D E) ) // (D E)
```

Die Funktion `member` im letzten Beispiel liefert statt \mathbb{T} die Restliste, deren erstes Element das gesuchte Element ist.

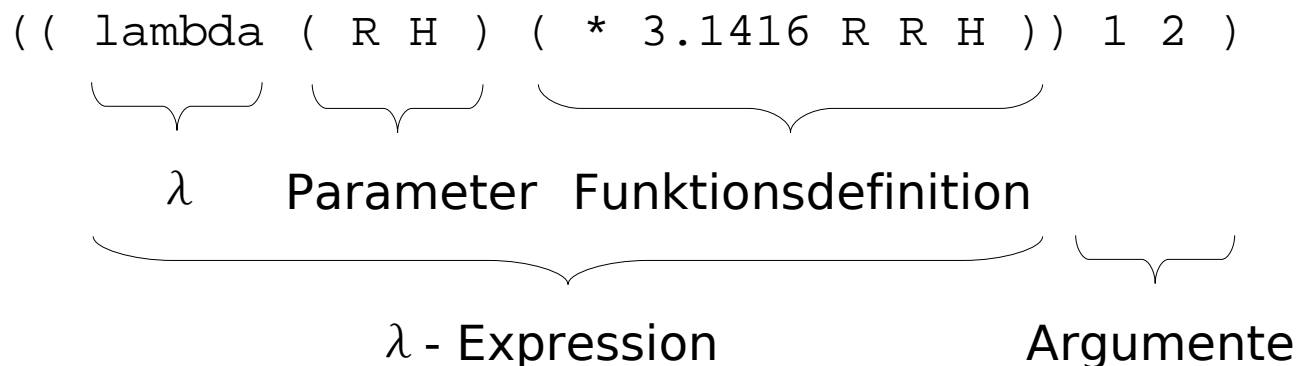
Die Lambda-Funktion

Zur Bearbeitung einer Formel, z.B. x^y mit den Argumenten (2, 3) ist es notwendig, eine Konvention bezüglich der Parameter-Bindung zu treffen.

In LISP wird in Anlehnung an die Schreibweise des λ -Kalküls von Alonso Church (1941):

$$\lambda ((x, y); y^x) (2, 3) = 9$$

die Funktion `lambda` benutzt, um eine (nicht explizit benannte) Funktionsdefinition zu formulieren:



Die gebundenen Parameter haben nur lokale Bedeutung, gleichnamige Atome ausserhalb der Lambda-Funktion werden dadurch nicht angesprochen.

Funktionen sind selbst Daten

Sortieren einer Liste anhand des `>`-Vergleichsoperators.

```
(sort (list 5 2 6 3 1 4) #'>)
```

Der Vergleichsoperator ist selbst eine Funktion.

Folgendes Beispiel mit einer Lambda-Expression soll dies verdeutlichen:

```
(sort (list '(9 a) '(3 b) '(4 c)) #'(lambda (x y) (< (car x)
  (car y))))
```

Sortiert die Liste anhand des ersten Elements jeder der drei Sublisten.

Um eine Funktion als Argument an eine andere Funktion zu übergeben wird der `function` Operator, abgekürzt mit `#'`, verwendet.

Funktionen und Daten liegen also nicht im gleichen Namespace, dies ist einer der Unterschiede von Common Lisp und Scheme.

Tail recursion

Ein weiterer wesentlicher Unterschied zwischen Common LISP und Scheme ist *tail recursion*, welche in funktionalen Programmiersprachen oft vorkommt und in Scheme implementiert ist.

```
(define (factorial n)
  (define (iterate n acc)
    (if (= n 0)
        acc
        (iterate (- n 1) (* acc n))))
  (if (< n 0)
      (display "Wrong argument!")
      (iterate n 1)))
```

- Der rekursive Aufruf ist das letzte Unterziel des Rumpfes der Klausel.
- Es gibt keine unversuchten alternativen Klauseln.

- Dies erlaubt einem Interpreter oder Compiler die Reorganisation der Ausführung zur Effizienz-Steigerung (Zeit, Speicherbedarf).

Tail recursion

Gewöhnliche Abarbeitung des vorherigen Codes:

```
call factorial (3)
  call iterate (3 1)
    call iterate (2 3)
      call iterate (1 6)
        return 6
      return 6
    return 6
  return 6
```

Reorganisierte Variante:

```
call factorial (3)
replace arguments with (3 1), jump into "iterate"
replace arguments with (2 3), re-iterate
replace arguments with (1 6), re-iterate
return 6
```

Definition benannter Funktionen

Angenommen, wir wollen unsere Funktionsdefinition von oben unter dem Namen ZYLINDER aufrufen:

```
(defun zylinder(r h) (* 3.1416 r r h))
```

Man beachte den analogen Aufbau der Lambda-Expression.

Anwendung:

```
(zylinder 1 2) // 6.283199
```

Erstes Programm: myMAX

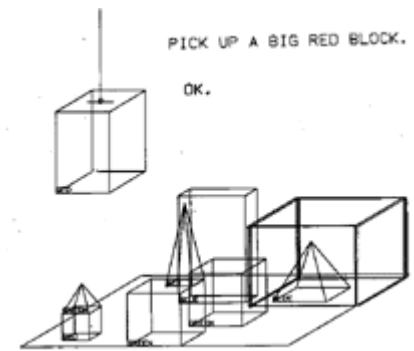
Die Funktion myMax gibt das grösste Element einer Liste zurück:

```
(defun myMax(L)
  (prog (X)
    (cond ( (null L) (return nil) ) )
    (setf X (car L) )
    (loop
      (setf L (cdr L) )
      (cond ( (null L) (return) )
            ( (< X (car L)) (setf X (car L) ) )
            )
      )
    )
  (return X)
)
```

```
>> (myMax '(3 6 8 1))           // gibt 8 zurück
```

```
>> (+ (myMax '(3 2)) (myMax '(5 3)) ) // gibt 8 zurück
```

Berühmte LISP-Anwendungen



SHRDLU

Wurde von 1968 bis 1970 von Terry Winograd am MIT entwickelt. SHRDLU ist in der Lage, mit dem Benutzer einen Dialog über eine kleine Block-Welt von zu führen.

Person: PICK UP A BIG RED BLOCK.

Computer: OK. (does it)

Person: GRASP THE PYRAMID.

Computer: I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN.

Person: FIND A BLOCK WHICH IS TALLER THAN THE ONE YOU ARE HOLDING AND PUT IT INTO THE BOX.

Computer: BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I AM HOLDING.

Computer: OK. (does it)

Person: WHAT DOES THE BOX CONTAIN?

Computer: THE BLUE PYRAMID AND THE BLUE BLOCK.

Person: WHAT IS THE PYRAMID SUPPORTED BY?

Computer: THE BOX.

Computer: FOUR OF THEM.

...

...

Berühmte LISP-Anwendungen

EMACS

Bietet LISP-Dialekt zur Erweiterung, Multics Emacs ist sogar komplett in LISP geschrieben

FESTIVAL

Festival: Geschrieben in C++, bietet aber ein Scheme-basierten Interpreter zur Befehlseingabe.

AUTOCAD

Bietet mit AutoLISP einen Dialekt zur Programmerweiterung

Software

GNU CLISP - an ANSI Common Lisp Implementation
<http://clisp.cons.org>

Pakete verfügbar für die gängigsten Linux-Distributionen

Quellenverzeichnis

Common Lisp The Language, 2nd Edition, Guy L. Steele
<http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html>

Common Lisp Tutorial, Geoffrey J. Gordon
<http://www.notam02.no/internt/cm-sys/cm-2.2/doc/clt.html>

Einführung in das Programmieren in Lisp, Christian-M. Hamann
2. Auflage, 1985, De Gruyter Lehrbuch

Wikipedia

http://en.wikipedia.org/wiki/Lisp_programming_language

http://en.wikipedia.org/wiki/Scheme_programming_language