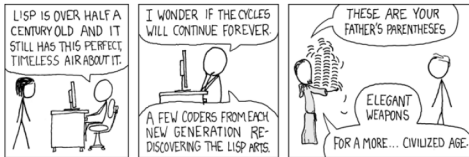


lisp



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.

— Eric S. Raymond, "How to Become a Hacker".

lisp

part 1: basics

basics

eine liste:

(foo bla fael)

ein stück code:

(foo bla fael)

;this is a comment

basics: evaluation

```
(+ 1 2 3) » 6
```

```
(string-upcase "bla") » "BLA"
```

```
(+ 23 (* 1 2 3)) » 29
```

```
(string-upcase (string-downcase "bLa"))  
» "BLA"
```

basics: t and nil

- falsch bzw. leere liste:

nil

()

- wahr:

t



basics: evaluation

dinge, die zu sich selbst
evaluieren:

"hello world"

23

t

nil

basics: t and nil

```
(if t
```

```
  2
```

```
  3)
```

```
» 2
```

```
(if nil
```

```
  2
```

```
  3)
```

```
» 3
```

basics: evaluation

```
(+ 2  
  3  
  (if (is-full-moon-p)  
       3  
       4))
```

» 9 [oder 8 falls vollmond ist]

basics: evaluation

alles was in () steht, evaluiert auch zu etwas

basics: functions

defun = "define function"

```
(defun add (number1 number2)  
  (+ number1 number2))
```

```
(add 1 2)
```

```
» 3
```

basics: format

```
(defun report-club-condition (place state)
  (format t "hello. the ~a is ~a."
          place
          state))
```

```
(report-club-condition "kueche" "dreckig")
```

```
hello. the kueche is dreckig.
```

```
» nil
```

basics: quoting

foo

» "i am the value of foo"

'foo

» foo

''foo

» 'foo

...

basics: quoting

```
(format t "the list is ~a."
        (eins 2 drei 4))
```

» The function EINS is undefined.

```
(format t "the list is ~a."
        '(eins 2 drei 4))
```

the list is (EINS 2 DREI 4).

» nil

basics: loops

```
(dolist (element '(eins zwei viele))  
  (format t "~a, " element))
```

```
EINS, ZWEI, VIELE,
```

```
» nil
```

basics: loops

```
(dotimes (index 4)
  (format t "quadrat von ~a ist ~a.~%"
           index
           (* index index)))
```

quadrat von 0 ist 0.

quadrat von 1 ist 1.

quadrat von 2 ist 4.

quadrat von 3 ist 9.

» nil

basics: REPL

the "read eval print loop":

```
(loop (print (eval (read))))
```

- lese eingabe ein
 - evaluiere sie
 - gib ergebnis aus
 - loop
-
- praktisch zum testen und spielen

skillz

- listen machen
- funktionen anwenden
- funktionen definieren: **defun**
- format: text ausgeben
- einfache loops: **dotimes**, **dolist**
- verzweigung: **if**

work work!

- bekomme den REPL zum laufen
- gib quadratwurzeln der zahlen von 0 bis 42 aus
- gib alle geraden zahlen von 23 bis 42 aus.
hint:
`(evenp 4) » t`
- mache funktion, die das für alle zahlen von 0 bis a tut
- play!

[cheat sheet available]

lisp

**part 2:
more stuff**

funktional

```
(mapcar #'sqrt '(1 4 9 16 25))  
» (1.0 2.0 3.0 4.0 5.0)
```

```
(every #'evenp '(2 4 6 8))  
» t
```

```
(reduce #'list '(1 2 3 4))  
» ((1 2) 3) 4)
```

funktional: lambda

- **problem:**
überprüfe, ob alle zahlen einer liste größer als 10 sind
- **erster ansatz:**

```
(defun groesser-als-zehn (zahl)  
  (> zahl 10))
```

```
(every #'groesser-als-zehn '(23 42 1337))
```

```
» t
```

funktional: lambda

- **nachteil: funktion fliegt so rum, wird aber eigentlich nur im every verwendet. daher:**

```
(every (lambda (zahl)
        (> zahl 10))
      '(23 42 1337))
```

» t

- **(lambda (var1 var2 ...) body) evaluiert also zu einer anonymen funktion**

imperative elemente

```
(defun do-lots-of-stuff ()  
  (do-something)  
  (do-more)  
  (do-even-more))
```

- **defun** erzeugt einen block. elemente eines blocks werden nacheinander evaluiert.
- rückgabewert des blocks ist der rückgabewert des letzten elements

imperativ: blocks

eine direkte möglichkeit, blöcke zu machen, ist **progn**:

```
(if t
  (progn
    (format t "foo")
    (format t "bar"))
  (format t "das passiert nicht"))
```

- ein block entspricht also ungefähr dem {} in c/java
- blöcke werden oft implizit erzeugt. **progn** braucht man selten

blocks with let

let erzeugt einen block, in dem symbole an werte gebunden sind [**defun** tut das ja auch]

```
(let ((zahl1 23)
      (zahl2 5)
      (zahl3 (+ 21 21)))
  (format t "die zahlen sind ~a, ~a und ~a."
          zahl1 zahl2 zahl3)
  (+ zahl1 zahl2 zahl3))
```

die zahlen sind 23, 5 und 42.

» 70

let

```
(let ((foo 1))  
  (format t "foo ist aussen ~a.~%" foo)  
  (let ((foo 2))  
    (format t "foo ist innen ~a.~%" foo))  
  (format t "foo ist aussen ~a.~%" foo))
```

```
foo ist aussen 1.  
foo ist innen 2.  
foo ist aussen 1.
```

```
» nil
```

imperativ: setf

setf ändert den wert, der an ein symbol gebunden ist.

```
(let ((zahl 23))  
  (format t "zahl ist ~a~%" zahl)  
  (setf zahl 42)  
  (format t "zahl ist ~a~%" zahl))
```

zahl ist 23

zahl ist 42

» nil

work work!

- verbessere den code von vorhin mit mapcar, **lambda**, ...
- schreibe funktionen, die
 - das minimum einer liste von zahlen zurueckgeben
 - eine liste von zahlen normieren (alle durch ihr maximum teilen)
- ...

[cheat sheet available]

lisp

part 3: magic

macro magic

```
(dotimes (index 23)  
  (do-some-stuff))
```

warum wird nicht versucht, (index 23)
auszuwerten?

lösung: **dotimes** ist ein macro. es wird zur
compile-zeit expandiert zu:



```
(do ((index 0 (1+ index)))  
  ((>= index 23) nil)  
  (declare (type unsigned-byte index))  
  (do-some-stuff))
```

macro magic

dies wiederum wird expandiert zu... igitt:



```
(let ((index 0))
  (declare (type unsigned-byte index))
  (tagbody
    (go #:G2113)
    #:G2112
    (tagbody (do-some-stuff))
    (let* ()
      (multiple-value-bind
        ( #:G2114
          (1+ index)
          (progn (setq index #:G2114) NIL)))
      #:G2113
      (if (not (>= index 23)) (progn nil (go #:G2112)) nil)
      (return-from nil (progn nil))))))
```

macro magic

- ein macro sieht aus wie eine funktion [steht am anfang einer liste]
- wird aber zur compilezeit evaluiert, bekommt den code als argument
- gibt neuen code zurueck, der dann anstatt des macros eingesetzt wird
= es wird expandiert

macro magic

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

```
(when something-is-true
  (do-stuff)
  (do-more-stuff))
```



```
(if something-is-true
  (progn
    (do-stuff)
    (do-more-stuff)))
```

macro magic

macros, die wir schon kennen:

`dotimes`, `dolist`, `loop`, `defun`, `setf`, `defmacro`

the mighty loop

`dotimes` und `dolist` sind nützlich, aber nicht sehr flexibel. `loop` kann alles!

```
(loop for i from 23 to 42 by 2
      do (format t "~2d")
      collecting (* i 2))
```

.....

```
» (46 50 54 58 62 66 70 74 78 82)
```

the mighty loop

ein **loop** kann aus den elementen bestehen:

- iteration control: **for..from..to**, **for..in..**, etc
- evaluation: **do..**
- combination: **collecting**, **counting**, **summing**, **minimizing**, etc

```
(loop for element in '(foo bla fasel)
      for i from 1 to 2
      do (format t "~a. ~a~%" i element))
```

```
1. F00
2. BLA
```

```
>> nil
```

funcall and apply

```
(funcall #' + 1 2 3 4)
```

```
» 10
```

```
(apply #' + '(1 2 3 4))
```

```
» 10
```

funcall

```
(defun repeat-function (fn iterations data)
  (dotimes (i iterations)
    (setf data
          (funcall fn data)))
  data)
```

```
(repeat-function (lambda (number)
                  (* number number))
                 3
                 2)
```

» 256

funcall

```
(defun mapcar (fn list)
  (loop for element in list
        collecting (funcall fn element)))
```

keyword symbols

- normalfall: evaluieren ergibt wert eines symbols
- symbole, die mit **:** beginnen, heissen keyword symbols und evaluieren zu sich selbst:

:foo

» :foo

- keyword symbols lassen sich z.b. wie enums verwenden.

keyword symbols

```
(defun muss-ich-aufstehen (wochentag)
  (if (or (eql wochentag :samstag)
          (eql wochentag :sonntag))
      nil
      t))
```

```
(muss-ich-aufstehen :montag)
» t
```

```
(muss-ich-aufstehen :samstag)
» nil
```

defparameter

```
(defparameter *wochenende*  
              '(:samstag :sonntag))
```

defun reloaded

```
(defun foo (a &optional b c d)  
  (list a b c d))
```

```
(foo 1)
```

```
» (1 nil nil nil)
```

```
(foo 1 2)
```

```
» (1 2 nil nil)
```

```
(foo 1 2 3)
```

```
» (1 2 3 nil)
```

```
(foo 1 2 3 4)
```

```
» (1 2 3 4)
```

defun reloaded

```
(defun foo (&key a b c)
  (list a b c))
```

```
(foo)                » (nil nil nil)
```

```
(foo :a 0)           » (0 nil nil)
```

```
(foo :b 0)           » (nil 0 nil)
```

```
(foo :c 0)           » (nil nil 0 )
```

```
(foo :a 1 :c 3)       » (1 nil 3 )
```

```
(foo :a 1 :b 2 :c 3) » (1 2 3 )
```

```
(foo :a 1 :c 3 :b 2) » (1 2 3 )
```

defun reloaded

```
(defun bar (&key (a 10) (b 20) (c 30))  
  (list a b c))
```

```
(bar)                » (10 20 30)
```

```
(bar :a 0)           » ( 0 20 30)
```

```
(bar :b 0)           » (10  0 30)
```

```
(bar :c 0)           » (10 20  0)
```

```
(bar :a 1 :c 3)       » ( 1 20  3)
```

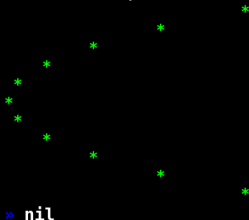
```
(bar :a 1 :b 2 :c 3) » ( 1  2  3)
```

```
(bar :a 1 :c 3 :b 2) » ( 1  2  3)
```

work work!

- verbessere den code von vorhin mit **loop**
- schreibe einen fun ktionsplotter:

```
(plot (lambda (x)
      (* x x))
      :start -5
      :end 5)
```



» nil

optional mit macro:

```
(plot (* x x)
      :start -5
      :end 5)
```

work done

eine lösung:

```
(defun plot (function &key (start 0) (end 10))  
  (loop for x from start to end  
    do (progn  
      (dotimes (y (round (funcall function x)))  
        (format t " "))  
        (format t "~*~%" y))))
```

lisp

part 4: CLOS and more on macros

CLOS

- the **common lisp object system**
- teil von ANSI Common Lisp
- alles ist ein objekt (auch funktionen, strings, listen, ...)
- klassenhierarchie: t ist root-klasse
- reflektiv (MOP)
- multiple dispatch, multiple inheritance

defclass

```
(defclass point ()  
  (x y))
```

```
(make-instance 'point)  
» #<POINT {B105EA1}>
```

defclass

```
(defclass point ()  
  ((x :initarg :init-x  
       :accessor x)  
   (y :initarg :init-y  
       :accessor y)))
```

```
(make-instance 'point :init-x 23 :init-y 42)  
» #<POINT {B13A3E1}>
```

```
(setf (x *some-point*) 17)
```

```
(x *some-point*)  
» 17  
(y *some-point*)  
» 42
```

defclass

```
(defclass line (drawable-thing)  
  ...)
```

```
(defclass glowing-thing (drawable-thing)  
  ...)
```

- mehrfachvererbung ist möglich:

```
(defclass glowing-line (line glowing-thing)  
  ...)
```

methods

```
(defgeneric distance (thing1 thing2))

(defmethod distance ((p1 point) (p2 point))
  (let ((x-distance (- (x p1)
                        (x p2)))
        (y-distance (- (y p1)
                        (y p2))))
    (sqrt (+ (* x-distance x-distance)
              (* y-distance y-distance)))))

(distance (make-instance 'point :x 2 :y 2)
          (make-instance 'point :x 3 :y 3))
» 1.4142135
```

multiple dispatch

- methoden "gehören" nicht einer klasse:

```
(defmethod distance ((p1 point) (l1 line))  
  ...)
```

- die spezialisierteste methode wird angewendet

methods

- mit `call-next-method` kann die nächst weniger spezialisierte methode aufgerufen werden:

```
(defmethod draw ((thing drawable-thing))  
  ...)
```

```
(defmethod draw ((my-line line))  
  ...  
  (call-next-method))
```

before, after, around

```
(defmethod draw :before ((thing animated-thing))  
  (step-animation thing))
```

```
(defmethod draw :around ((thing thing-in-db))  
  (with-db-connection *my-db*  
    (call-next-method)))
```

**datenbankanbindung einrichten/abbauen,
bookkeeping, logging, checks, ...**

macros sind wichtig

- du bemerkst, dass du muster im code wiederholst -> macro draus machen -> aaah :)
- die sprache entwickelt sich auf das problem zu (siehe **loop**)
- das problem entwickelt sich zu einem zunehmend high-level formulierten <- TODO

macros sind wichtig

- code wird lesbarer und kompakter
- macros haben lisp alle programmiertrends assimilieren lassen
- "syntactic sugar" ist immer nur mit mitteln der sprache implementiert

lisp

part 5: why use lisp?

bad things

- generische funktionen dürfen nicht wie bestehende "normale" funktionen heissen
- inkonsistenzen durch lange geschichte
- executables erzeugen stinkt

more features

i did not tell you about:

- **garbage collector**
- **interaktiv: zur laufzeit zeug nachladen/aendern etc.**
- **closures**
- **lisp ist kompiliert: (disassemble #'my-function)**
- **libraries**
- **builtin: hashtabellen, arrays, big nums, komplexe zahlen, ...**
- **multiple values**
- **conditions**
- **reader macros**

readability

- **lisp-code ist lesbar und kompakt -> spass**

Q: How can you tell when you've reached Lisp Enlightenment?

A: The parentheses disappear.

— Anonymous

SQL, Lisp, and Haskell are the only programming languages that I've seen where one spends more time thinking than typing.

— Philip Greenspun

leet

- **lisp ist leet**

Just because we Lisp programmers are better than everyone else is no excuse for us to be arrogant.

— Erann Gat

There are no average Lisp programmers. We are the Priesthood. Offerings of incense or cash will do.

— Kenny Tilton at comp.lang.lisp

macros

- **macros machen lisp einzigartig**

Lisp is a programmable programming language.

— John Foderaro, CACM, September 1991

Lisp isn't a language, it's a building material.

— Alan Kay

now what?

- read "Practical Common Lisp":
<http://gigamonkeys.com/book>
- [use (x)emacs+slime as your ide]
- surf cliki.net
- hack more

ask me :)