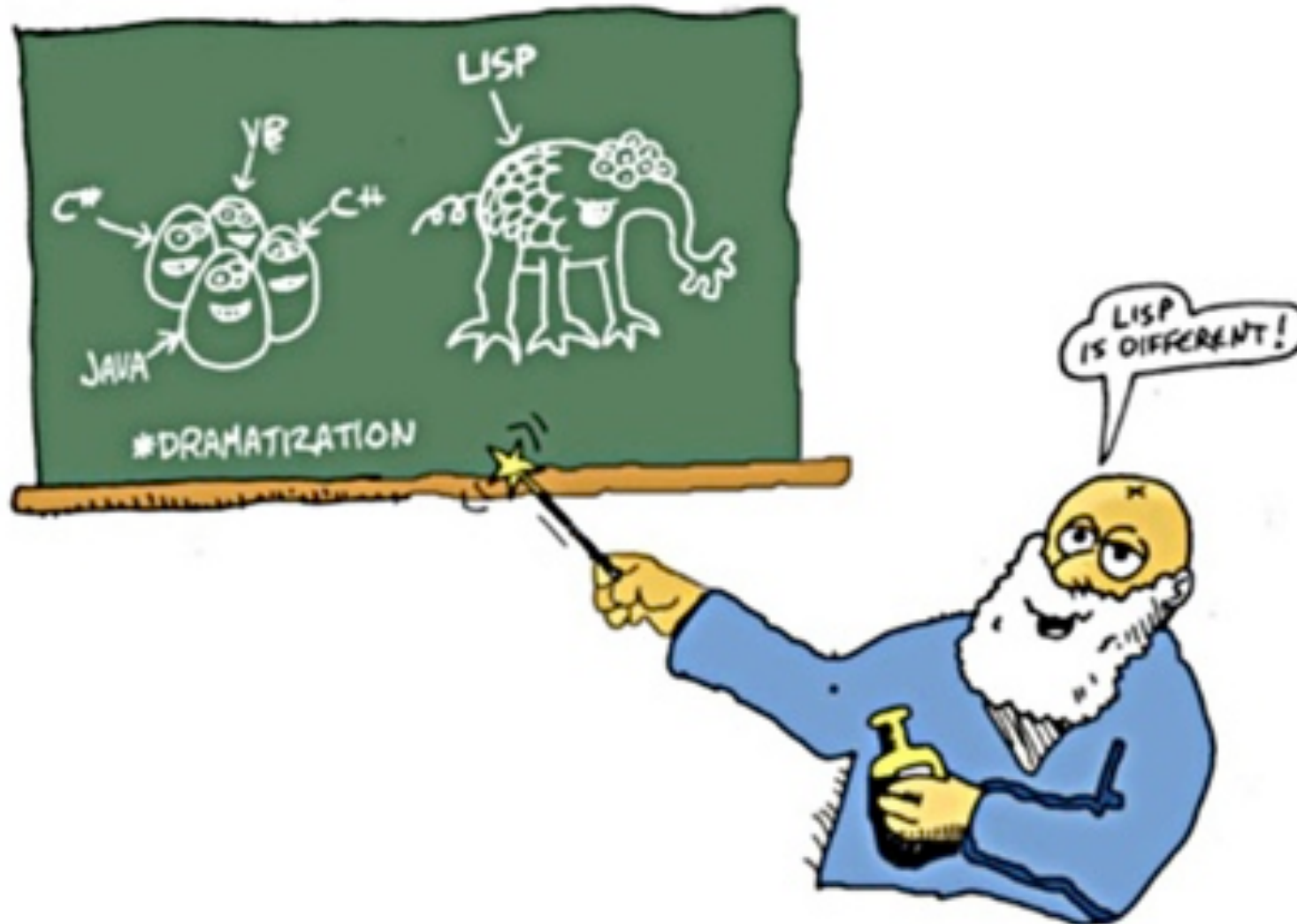


Lisp Teil 1

Einführung in Common Lisp

Programmiersprachen und -paradigmen (PSPP)

A Warning: Lisp is different!



Lisp in PSPP?

- Nicht rein funktional: Multiparadigmensprache
- Bessere Sprachen für funktionale Programmierung, z.B. Haskell
- Common Lisp ist schon einige Jahre alt
- Schneller zu erlernen als Haskell
- Erlaubt funktionale Programmierung (u.a.)
- Mutter aller Programmiersprachen genannt
- Ermöglicht neue Perspektive aufs Programmieren
- Grundlage für moderne Dialekte, z.B. Clojure

<https://exploringdata.github.io/vis/programming-languages-influence-network/>

Übersicht

- Lisp – Was soll das sein ??
- Ausdrücke und Datentypen
- Kleine Programme
- Auftrag, Quellen, weitere Informationen
- Anhang: Common Lisp Implementierungen


Lisp – was soll das sein ??

- Lisp ist eine der ältesten Programmiersprachen
- 1956: Dartmouth Summer Research Project on Artificial Intelligence
- 1956...1958: Konzept der Sprache, John McCarthy am MIT
- Ab 1958: erste Implementierung, Steve Russell, IBM 704
- Aber es ist immer noch irgendwie aktuell...

??

Read - Eval - Print Loop

- Kommunikation mit dem Lisp-Interpreter über eine interaktive Kommandozeile
- Eingebener Ausdruck wird vom Interpreter *evaluiert* und das Ergebnis ausgegeben
- *Read-Eval-Print Loop, REPL*



```
Listener
Welcome to Clozure Common Lisp Version 1.8-store-r15418 (DarwinX8664)!
? (+ 3/4 1/2 1/4)
3/2
? (cons '+ (list 3/4 1/2 1/4))
(+ 3/4 1/2 1/4)
? (string-upcase "hello")
"HELLO"
? |
CL-USER: (Lisp Listener) Listener(8) [Toplevel Read]
```

Demo: Erste Versuche

```
> (quote (was ist denn hier los ?))    ;; Kommentar  
(WAS IST DENN HIER LOS ?)             ;; das ist eine Liste  
  
> '(was ist denn hier los ?)           ;; ' statt quote  
(WAS IST DENN HIER LOS ?)             ;; Listen sind wichtig in Lisp  
  
> '(was ist (denn hier) los ?)         ;; verschachtelte Liste  
(WAS IST (DENN HIER) LOS ?)  
  
> '(die "Antwort" ist 42)              ;; Symbol, String, Zahl  
(DIE "Antwort" IST 42)  
  
> '(+ 1 2 3 4 5)                      ;; noch eine Liste  
??
```

Demo: Erste Versuche

> (+ 1 2 3 4 5) 15	:: aha, so wird die Addition ausgeführt
> 3.14 3.14	:: Zahlen evaluieren zu sich selbst
> "Hallo Lisp" "Hallo Lisp"	:: Strings auch
> 3/8 3/8	:: Bruchzahlen
> (+ 3/8 1/6) ??	:: Präfix-Notation

Demo: Erste Versuche

```
> (defun foo (n) (+ (* 2 n) 1))      ;; Funktion definieren
FOO

> (foo 4)                           ;; Funktion aufrufen
9

> (foo (- 7 2))
11

> (funcall 'foo 7)                   ;; aufzurufende Funktion
15                                   ;; als Argument

> (funcall (first '(foo bar baz)) 7) ;; Auswahl der Funktion
15
```

Demo: Erste Versuche

```
;; Listen, verschachtelt und verschiedene Typen enthaltend  
;; damit: beliebig komplexe Datenstrukturen
```

```
> (set 'ding '(auto (marke "VW") (baujahr 1981) (gewicht (894 kg))))  
(AUTO (MARKE "VW") (BAUJAHR 1981) (GEWICHT (894 KG)))
```

```
;; Mapping über eine Liste  
> (mapcar 'second (rest ding))  
??
```

```
> (mapcar 'foo '(23 5 12 32))  
??
```

```
;; mapcar und weitere Funktionen können Schleifen ersetzen  
;; diese Möglichkeiten werden wir später noch genauer ansehen...
```

Funktionale Programmierung

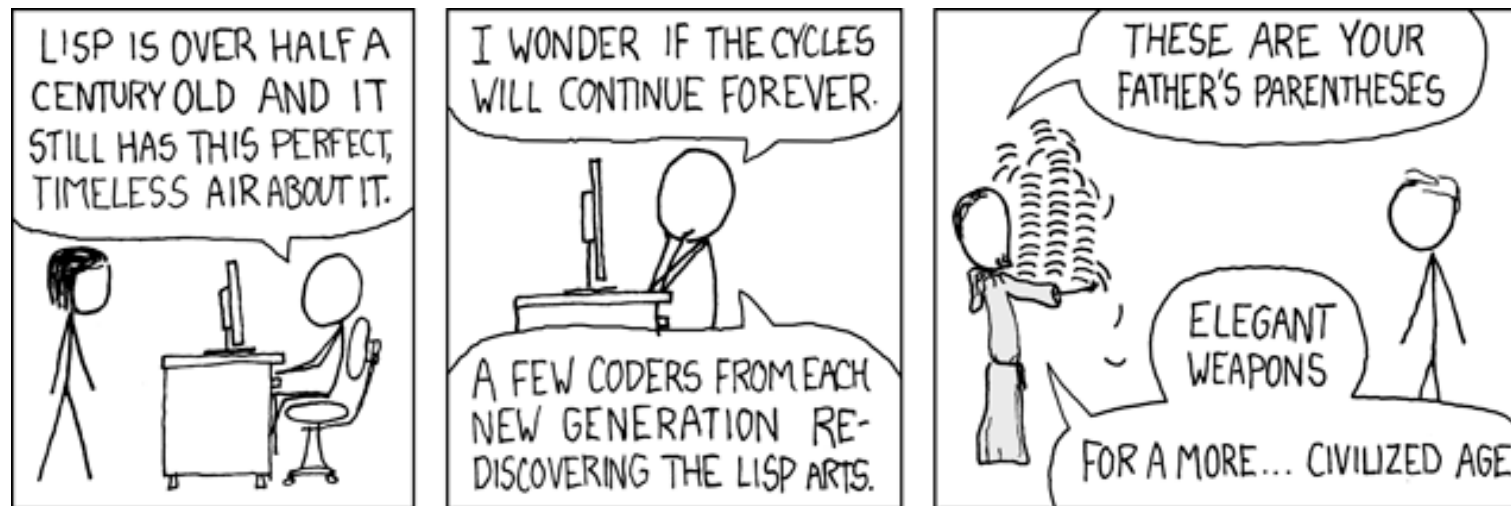
Wir notieren nebenbei ein paar erste Merkmale funktionaler Programmierung:

- Funktionen spielen eine zentrale Rolle
- Einfache, möglichst universelle Datentypen (wie Listen)
- Funktionen, die auf diesen Datentypen arbeiten (wie mapcar)
- Funktionen als Parameter anderer Funktionen

Lisp

Und ausserdem ein paar Erkenntnisse zu Lisp:

- Ziemlich viele Klammern
- Listen als zentrale Datenstruktur
- Lisp steht für *List Processing*



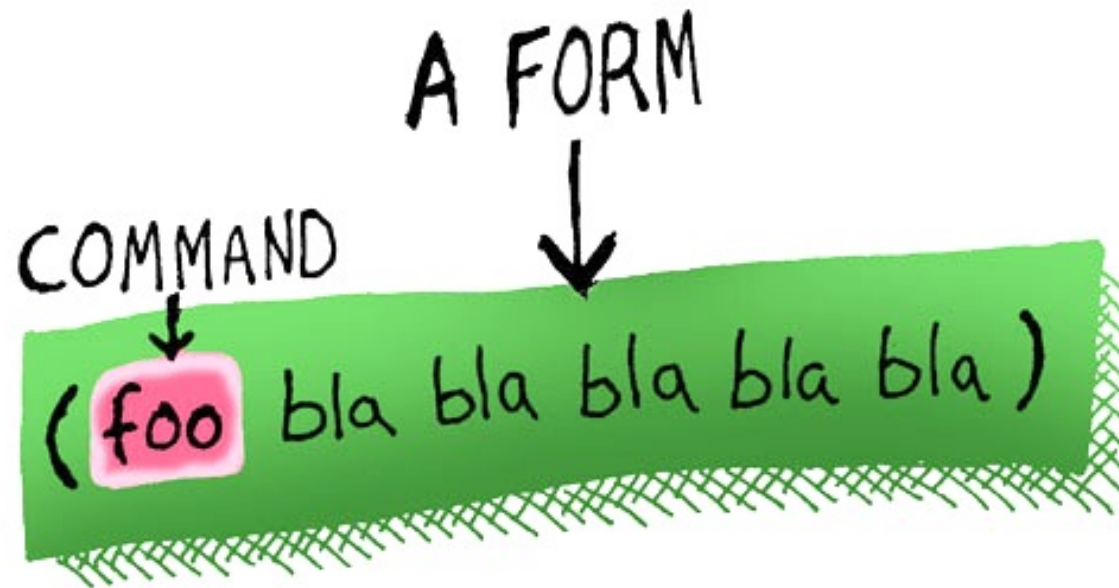
Einfache Syntax

- Daten wie Programmcode bestehen aus Listen
- Listen werden in runden Klammern eingeschlossen und können beliebig verschachtelt werden

A LIST IN LISP
↓
(Bla bla bla bla bla)

Einfache Syntax

- Das erste Element der Liste kann ein „Kommando“ sein
- Der Interpreter (oder Compiler) kann die Liste dann als Funktionsaufruf (o. ä.) interpretieren



Geschichte

- Lisp war neben Prolog die vorherrschende Sprache im Bereich der *Künstlichen Intelligenz*
- Ein Problem war lange Zeit der knappe Speicher und geringe Prozessorleistung der Hardware
- *Lisp -Maschinen*: Für Lisp spezialisierte Hardware entstand vor allem aus diesem Grund
- Heute ist eine solche Spezialisierung nicht mehr nötig



Geschichte

- Zwei Assembler-Makros der IBM 704 waren *car* (Contents of Address Register) and *cdr* (Contents of Decrement Register)
- Diese beiden Befehle wurde in Lisp zu Operationen:
car liefert das erste Element einer Liste,
cdr den Rest der Liste ohne erstes Element
- Ausserdem
cons um Listen zu konstruieren (Element in Liste einfügen)

```
> (car '(1 2 3 4 5))
```

```
1
```

```
> (cdr '(1 2 3 4 5))
```

```
(2 3 4 5)
```

```
> (cons 1 '(2 3 4 5))
```

```
(1 2 3 4 5)
```


Beispiel: Liste verarbeiten

```
(defun list-to-strings (list)
  (if (null list) '()
      ;; else
      (cons (princ-to-string (car list))      ;; car oder first
            (list-to-strings (cdr list)))))    ;; cdr oder rest
```

```
> (list-to-strings '(1 (2 3) three "4"))
("1" "(2 3)" "THREE" "4")
```

```
> (mapcar 'princ-to-string '(1 (2 3) three "4"))
("1" "(2 3)" "THREE" "4")
```

Lisp: weitere Eigenschaften

- *Dynamisches Typenkonzept*
- Unterstützung *funktionaler Programmierung*
- Unterstützung weiterer Programmierparadigmen:
CLOS – Common Lisp Object System
- Programmcode kann selbst wie Datenstrukturen verarbeitet werden

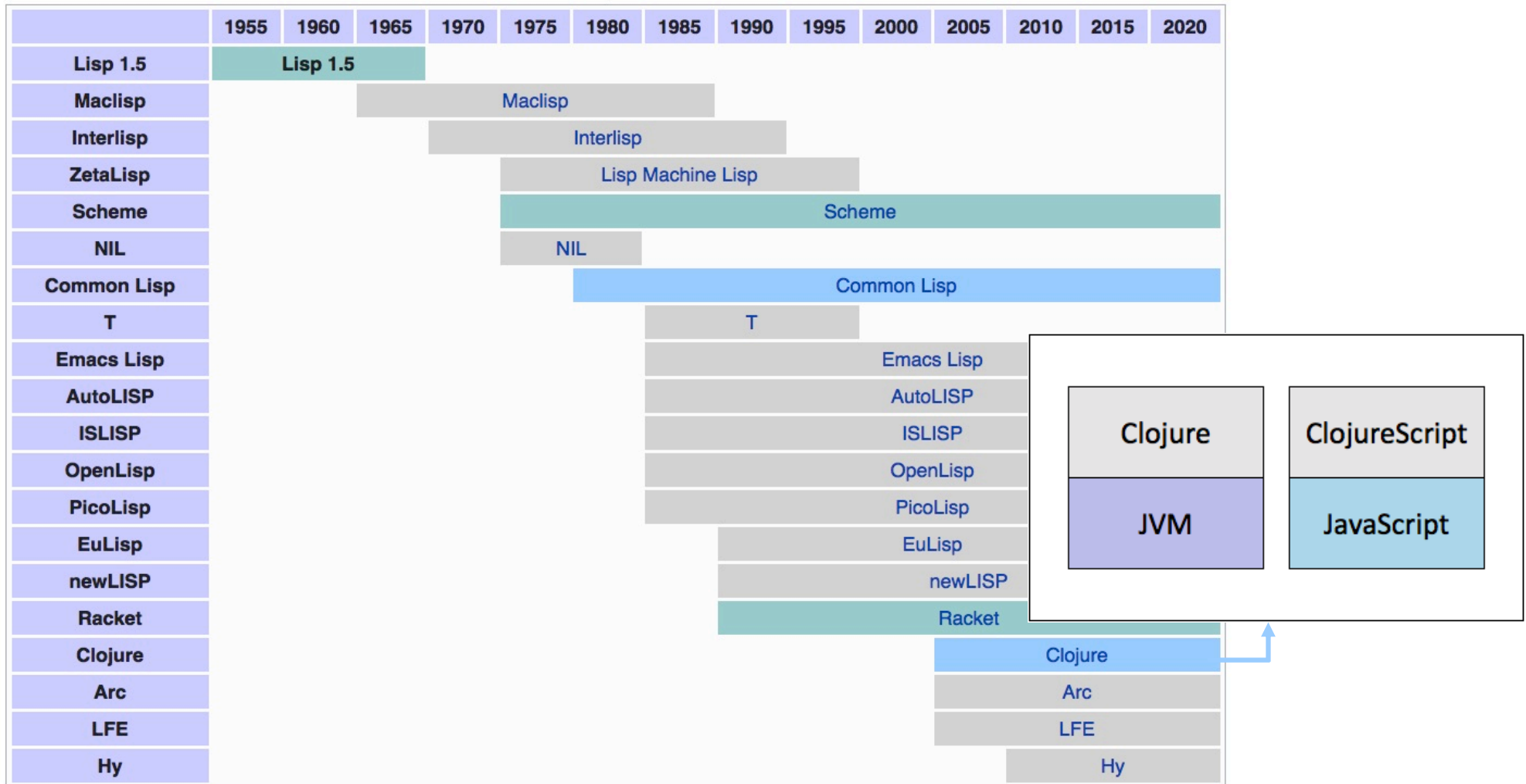
Wir werden Lisp vor allem im Hinblick auf die Unterstützung funktionaler Programmierung ansehen

Common Lisp

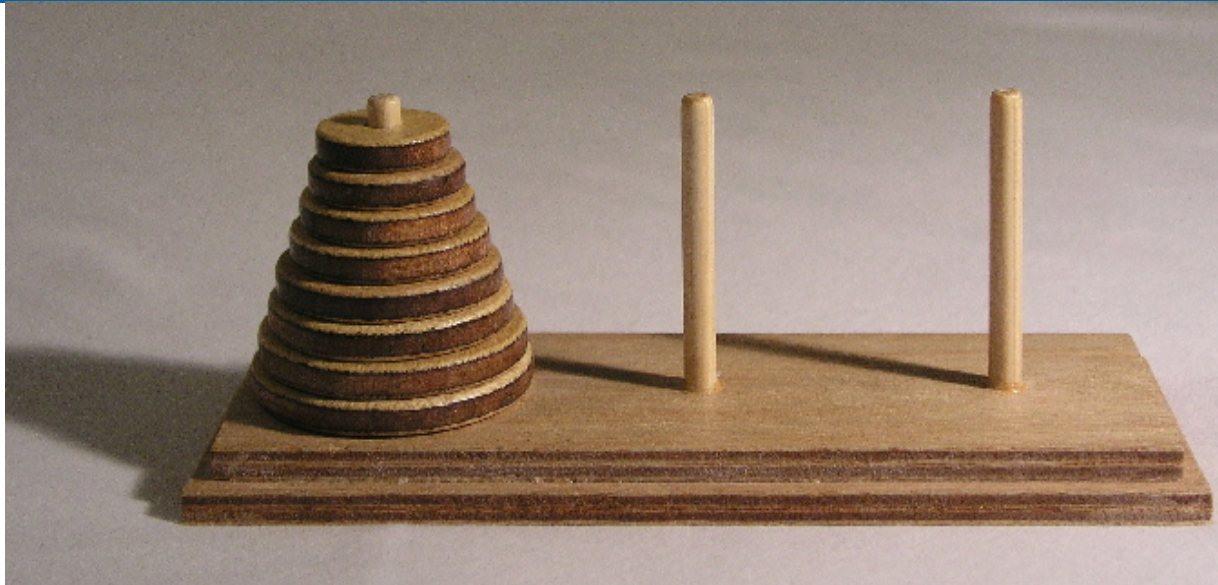
- ANSI-Standard 1994
- Umfangreich, häufig eingesetzt
- Beschrieben in:
Common Lisp the Language, 2nd Edition (s. Links)
- Zahlreiche Implementierungen
Steel Bank Common Lisp (SBCL), CLISP, ...
- Heute zwar nicht mehr topaktuell ... aber eine gute (die beste?)
Basis, andere Dialekte schnell zu erlernen

Andere Lisp-Dialekte

Timeline of Lisp dialects^(edit)



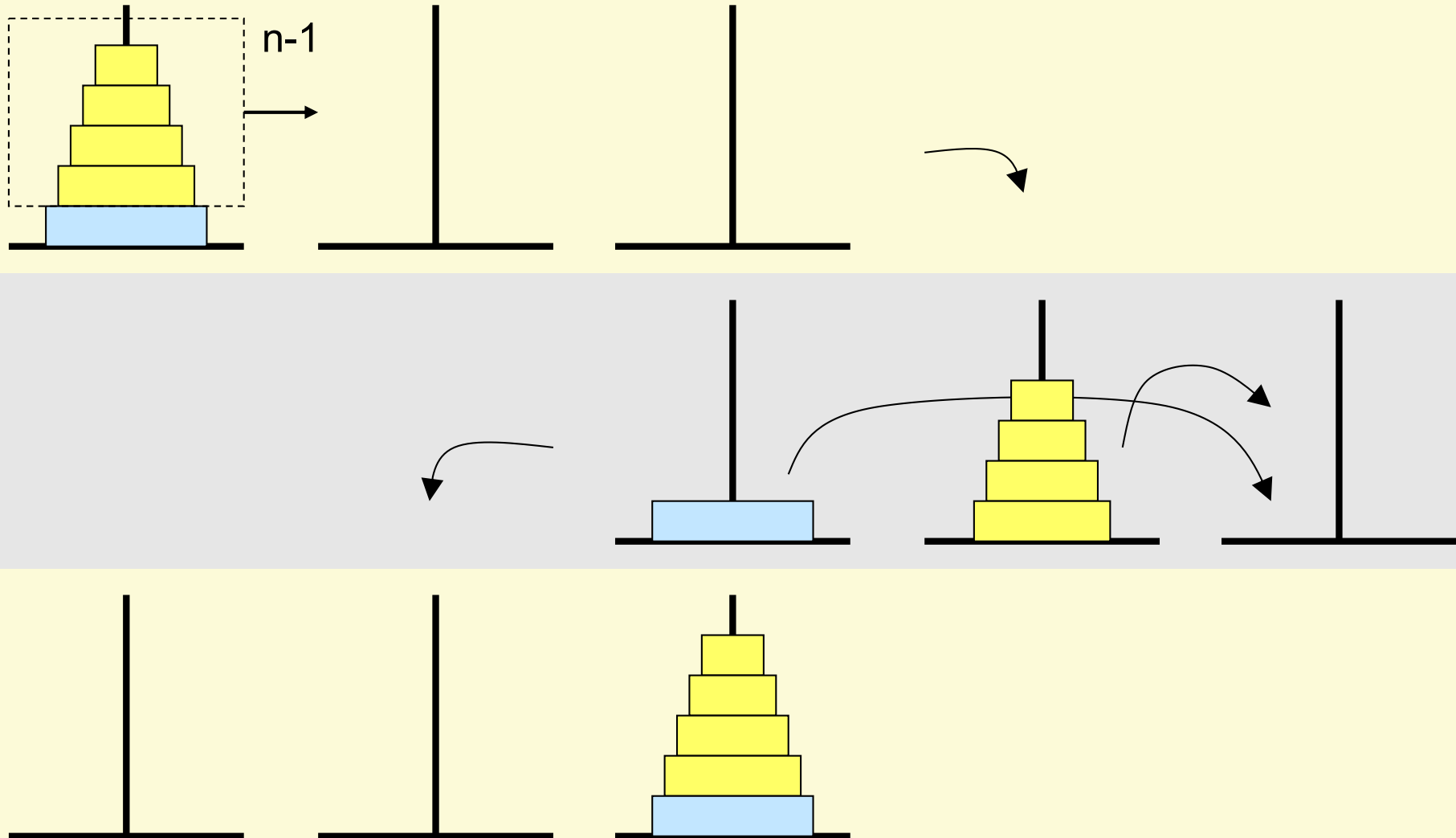
Beispiel: Towers of Hanoi



Die Scheiben sollen von Stapel 1 zu Stapel 3 bewegt werden:

- Es darf immer nur eine Scheibe bewegt werden
- Scheiben dürfen nur auf einem der drei Stapel abgelegt werden
- Es darf niemals eine grössere auf einer kleineren Scheibe liegen

Lösung rekursiv



Lösung in Common Lisp

```
(defun dohanoi (n from temp to)
  (if (> n 0)
      (append
        (dohanoi (- n 1) from to temp)
        (list (list from to)))
      (dohanoi (- n 1) temp from to))))
```

```
> (dohanoi 2 "links" "mitte" "rechts")
(("links" "mitte") ("links" "rechts")
 ("mitte" "rechts"))

> (dohanoi 4 'a 'b 'c)
((A B) (A C) (B C) (A B) (C A) (C B) (A B)
 (A C) (B C) (B A) (C A) (B C) (A B) (A C)
 (B C))
```

- Das ist schon ganz gut
- Aber es geht noch besser...

Lösung in Common Lisp

```
(defun dohanoi (n from temp to &optional (out 'list))  
  (if (> n 0)  
      (append  
        (dohanoi (- n 1) from to temp)  
        (list (funcall out from to))  
        (dohanoi (- n 1) temp from to))))
```

```
> (dohanoi 3 'a 'b 'c)  
((A C) (A B) (C B) (A C) (B A) (B C) (A C))
```

```
> (dohanoi 3 'a 'b 'c (lambda (x y) (list 'from x 'to y)))  
((FROM A TO C) (FROM A TO B) (FROM C TO B) (FROM A TO C) (FROM B TO  
A) (FROM B TO C) (FROM A TO C))
```


Lösung in Common Lisp

```
(defun print-hanoi (a b)
  (format t "move from ~S to ~S~%" a b)
  'ok)
```

```
> (dohanoi 3 'a 'b 'c 'print-hanoi)
move from A to C
move from A to B
move from C to B
move from A to C
move from B to A
move from B to C
move from A to C
(OK OK OK OK OK OK OK)
```

Zitate

"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."

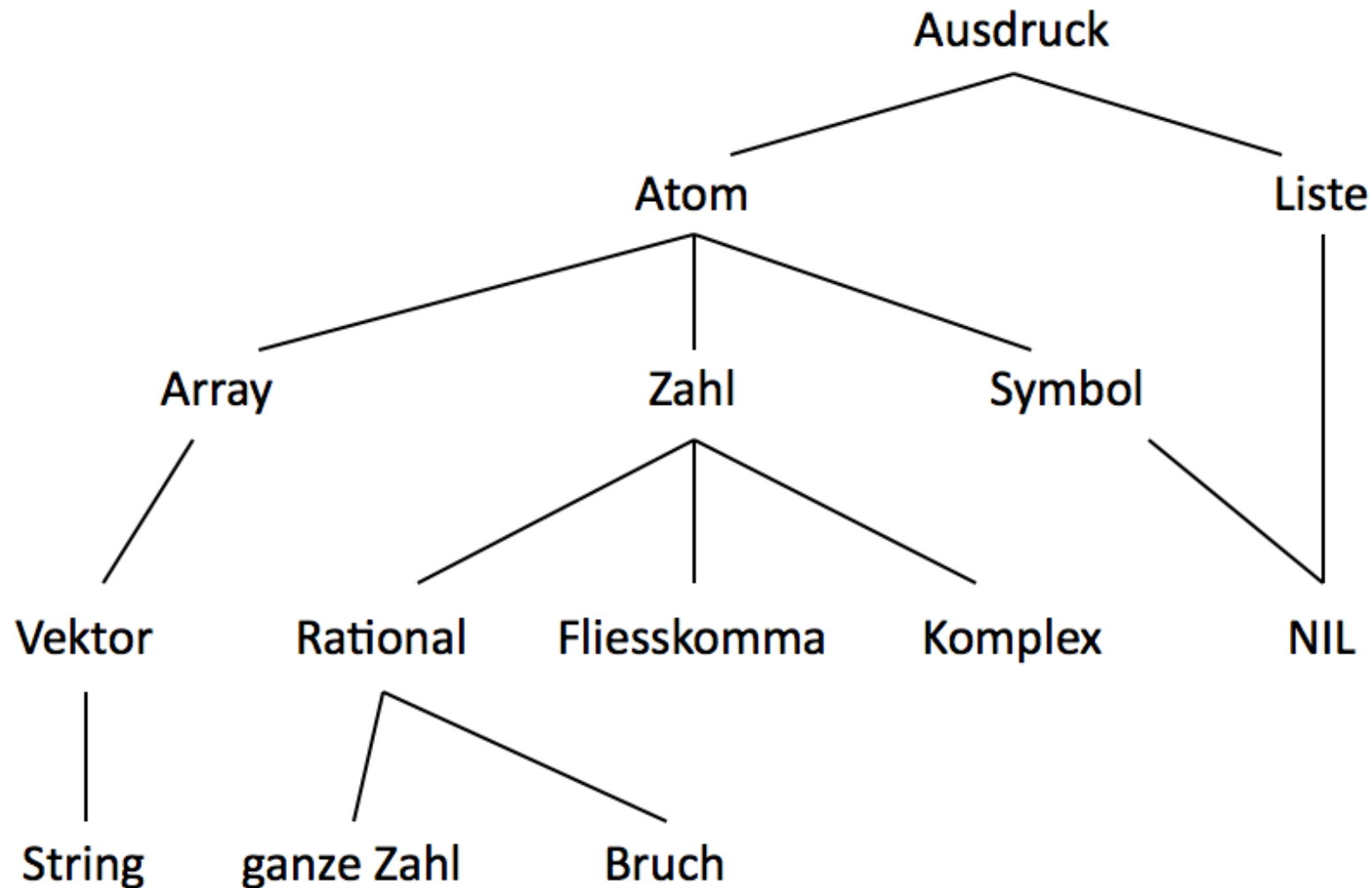
Eric S. Raymond, "How to Become a Hacker"

<http://www.catb.org/~esr/faqs/hacker-howto.html#skills1>

Übersicht

- Lisp – Was soll das sein ??
- Ausdrücke und Datentypen
- Kleine Programme
- Auftrag, Quellen, weitere Informationen
- Anhang: Common Lisp Implementierungen

Grundlegende Datentypen



Symbol und Wert

- An Symbole können (u.a.) *Werte gebunden* werden
- Symbole werden durch ihre Werte ersetzt (*evaluiert*), wenn sie nicht in einem Quote (z.B. mit Hochkomma) vorkommen

```
> figuren  
*** - EVAL: variable FIGUREN has no value  
> (set 'figuren '(quadrat kreis dreieck))  
(QUADRAT KREIS DREIECK)  
> figuren  
(QUADRAT KREIS DREIECK)
```

Für unsere Versuche auf der Kommandozeile kann *set* praktisch sein

In Funktionen verwenden wir *set* nicht (typische Anweisung imperativer Programmierung)

Symbol und Wert

```
> 'figuren
```

```
FIGUREN
```

```
> figuren
```

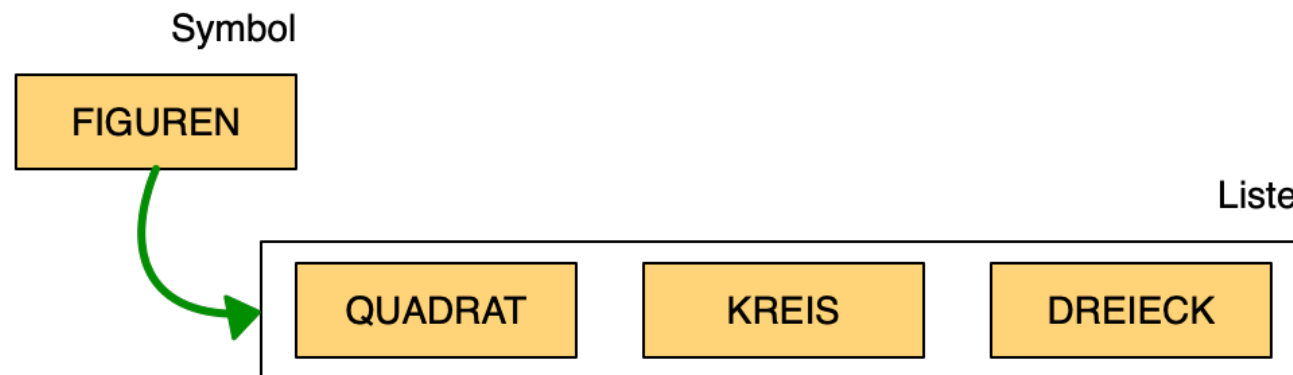
```
(QUADRAT KREIS DREIECK)
```

```
> (set (car '(a b)) 3)
```

```
3
```

```
> a
```

```
3
```



- Hochkomma verhindert die Evaluation (Abkürzung für *quote*)
- Argumente von *set*: auch Ergebnis einer Evaluation möglich

Symbol und Funktion

- Neben Werten können auch *Funktionen* an Symbole gebunden werden (in Common Lisp beides gleichzeitig möglich)

```
> (defun some-number (from to)
  (+ (random (+ (- to from) 1)) from))
```

SOME-NUMBER

```
> (set 'some-number 22)
```

22

```
> some-number
```

22

```
> (some-number 1 6)
```

5

```
> (symbol-value 'some-number)
```

22

```
> (symbol-function 'some-number)
```

#<FUNCTION SOME-NUMBER (FROM TO) (DE..

```
> #'some-number
```

#<FUNCTION SOME-NUMBER (FROM TO) (DE..

Listen

- Listen können weitere Listen oder *Atome* (Zahlen, *Symbole*) enthalten
- Listen und Atome werden auch als *S-Expressions* (S steht für *symbolic*) bezeichnet
- Einfacher Aufbau, leicht zu parsen

```
(obj-a (typ wuerfel)
      (farbe rot)
      (laenge (5 5 5))
      (oberfläche gepunktet)
      (liegt-auf tisch)
      (liegt-unter obj-b))
```

Datenstruktur oder Programm?

Atome und Listen

```
> 17
```

```
17
```

```
> 'table
```

```
TABLE
```

```
> (list 17 'table)
```

```
(17 TABLE)
```

```
> '(17 table)
```

```
(17 TABLE)
```

```
> (17 table)
```

```
*** - EVAL: 17 is not a function  
name; try using a symbol instead
```

- *17* und *table* werden als *Atome* bezeichnet
- Das Hochkomma vor *table* ist nötig, damit der Interpreter nicht versucht, das Symbol zu evaluieren

Ausdrücke

```
> (+ 3 4)
```

```
7
```

```
> '(+ 3 4)
```

```
(+ 3 4)
```

```
> (car '(1 2 3 4 5))
```

```
1
```

```
> (cdr '(1 2 3 4 5))
```

```
(2 3 4 5)
```

- *Präfix-Notation*
- Erstes Element:
Funktion, Operator, Makro
- Restliche Elemente:
Argumente
- *Evaluation, Bewertung*
eines Ausdrucks
- *Wert* eines Ausdrucks

Übung: Listen...

```
> (car '(quadrat kreis dreieck))
```

```
??
```

```
> (cdr '(quadrat kreis dreieck))
```

```
??
```

```
> (cdr (cdr (cdr '(1 2 3 4 5 6))))
```

```
??
```

```
> (car (cdr (cdr '(1 2 3 4 5 6))))
```

```
??
```

```
> (caddr '(1 2 3 4 5 6))
```

```
??
```

```
> (cadadr '(1 (2 (3 4) 5) 6))
```

```
??
```

Listen

```
> (list 1 2 (list 3 4))  
(1 2 (3 4))  
> (first '(quadrat kreis dreieck))  
QUADRAT  
> (rest '(quadrat kreis dreieck))  
(KREIS DREIECK)  
> (third '(quadrat kreis dreieck))  
DREIECK  
> (nth 2 '(quadrat kreis dreieck))  
DREIECK
```

- *list* ist eine *Funktion*
- Argumente werden ebenfalls ausgewertet
- Statt *car* und *cdr* kann auch *first* und *rest* verwendet werden
- Es gibt noch weitere Funktionen für Listen

One of the Lisp books has been promoted as „Lisp for the CDR of us“

Leere Liste

```
> (cdr '(1 2 3))
```

```
(2 3)
```

```
> (cddr '(1 2 3))
```

```
(3)
```

```
> (cdddr '(1 2 3))
```

```
NIL
```

```
> '()
```

```
NIL
```

```
> (if nil
```

```
      (list 1 2)
```

```
      (list 3 4))
```

```
(3 4)
```

- *nil* steht für die *leere Liste*
- In logischen Ausdrücken gilt *nil* bzw. die leere Liste als *false*

Weitere Listenoperationen

```
> (set 'a '(1 2 3 4))  
(1 2 3 4)
```

```
> (set 'b '(11 12 13))  
(11 12 13)
```

```
> (cons 99 a)  
(99 1 2 3 4)
```

```
> (append a b)  
(1 2 3 4 11 12 13)
```

```
> (last a)  
(4)
```

```
> (butlast a)  
(1 2 3)
```

```
> (length (append a b))  
7
```

```
> (reverse (append a b))  
(13 12 11 4 3 2 1)
```

- Diese Funktionen sind *nicht destruktiv*, d.h. sie verändern die ursprünglichen Listen nicht

Zahlen

```
> (abs -6)
```

```
6
```

```
> (min 3 8 1 5)
```

```
1
```

```
> (sqrt 2)
```

```
1.4142135
```

```
> (expt 6 3)
```

```
216
```

```
> (+ 1.25 1/2)
```

```
1.75
```

```
> (* 3.1415 2)
```

```
6.283
```

```
> (/ (* 85 (+ 1 0.15)) 2)
```

```
48.875
```

```
> pi
```

```
3.1415926535897932385L0
```

```
> (sin (/ pi 2))
```

```
1.0L0
```

```
> (coerce (sin (/ pi 2)) 'single-float)
```

```
1.0
```

Characters und Strings

- **Characters**
Einzelne Zeichen, z.B. `#\a`:
- **Strings**
Eindimensionale Arrays (Vektoren) von Zeichen

```
> (string-upcase "hello")  
"HELLO"  
> (string-capitalize "hello")  
"Hello"  
> (string-trim " " "hello")  
"hello"  
> (string-trim " (*)" " ( *three (silly) words* ) ")  
"three (silly) words"
```


Weitere Datentypen

- Sequences
Verallgemeinerung von Listen und Vektoren
- Arrays
Werden mit *make-array* erzeugt, Zugriff über Index (*aref*)
- Hashtables
Werden mit *make-hash-table* erzeugt, Zugriff über Schlüssel
- Structures
Werden mit *defstruct* erzeugt, Attribute und Default-Werte
- Streams
Für die Ein-/Ausgabe

Übersicht

- Lisp – Was soll das sein ??
- Ausdrücke und Datentypen
- Kleine Programme
- Auftrag, Quellen, weitere Informationen
- Anhang: Common Lisp Implementierungen

Einfache Funktionen

```
> (defun flaeche-quadrat (laenge)
    (list 'flaeche (* laenge laenge)))
```

FLAECHE-QUADRAT

```
> (flaeche-quadrat 10)
```

??

Liste:
Leerzeichen
erforderlich

- Funktionen werden mit *defun* definiert
- Anschliessend folgt der Funktionsname, eine Liste von Parametern und ein oder mehrere Ausdrücke
- Aufruf: der Wert des letzten Ausdrucks wird als Funktionswert zurückgegeben

Einfache Funktionen

```
> (defun flaeche-quadrat (laenge)  
    (list 'flaeche (* laenge laenge)))
```

FLAECHE-QUADRAT

- Die **defun**-Spezialform evaluiert zum Funktionsnamen
- Als Seiteneffekt wird die Funktion an den Namen gebunden

Bedingte Evaluation

```
(if nil ;; if ist eine Spezialform
    (list 1 2 "foo") ;; Bedingung
    (list 3 4 "bar")) ;; Ergebnis, falls Bedingung wahr
                        ;; Ergebnis, falls Bedingung falsch
```

- Evaluiert zu Argument 2 oder 3 je nach Argument 1
- Common Lisp kennt keinen Datentyp *boolean*
- In Bedingungen gilt die leere Liste (*nil*) als *false* und alles andere, zum Beispiel *t* als *true*
- *t* ist wie *nil* eine Konstante, der Wert von *t* ist *t*

Beispiel: Fakultät

```
(defun fakultaet (n)
  (if (> n 1)
      (* n (fakultaet (- n 1)))
      1))
```

```
> (fakultaet 50)
```

```
3041409320171337804361260816606476884437764156896051200000000000
```

- Es ist ein grosser Vorteil, sich nicht um die maximale Grösse von Zahlentypen kümmern zu müssen :)

Prädikate und Vergleiche

```
> (atom 'farben)
```

```
T
```

```
> (atom '(rot gruen blau))
```

```
NIL
```

```
> (<= 3/8 2/7)
```

```
NIL
```

```
> (zerop 0)
```

```
T
```

```
> (not nil)
```

```
T
```

```
> (or nil 'wuerfel nil nil)
```

```
WUERFEL
```

- Typ prüfen: *atom*, *listp*, *numberp*, *symbolp*, *null*, ...
- Vergleichsoperatoren: *>*, *>=*, *<*, *<=*
- Prädikate für Zahlen: *zerop*, *plusp*, *minusp*, *evenp*, *oddp*, ...
- Logische Verknüpfungen: *not*, *and*, *or*

Gleichheit

```
> (equal '(rot gruen blau) '(rot gruen blau))
```

```
T
```

```
> (eq '(rot gruen blau) '(rot gruen blau))
```

```
NIL
```

```
> (eq 'farbe 'farbe)
```

```
T
```

```
> (= 4 4)
```

```
T
```

```
> (= 4 4.0)
```

```
T
```

```
> (eq1 4 4)
```

```
T
```

```
> (eq1 4 4.0)
```

```
NIL
```

- Es gibt verschiedene Möglichkeiten, die Gleichheit von Ausdrücken zu überprüfen

- Details in der Lisp Doku, oder:

<https://anticrisis.github.io/2017/09/08/equality-in-common-lisp.html>

Verzweigungen mit cond

```
(cond ((> a 7) 'groesser7)  
      ((= a 7) 'gleich7)  
      (t      'kleiner7))
```

- Mehrfachverzweigung: Ausdrücke nach erster erfüllter Bedingung werden ausgewertet
- t als letzte Bedingung: Default-Zweig
- Ist keine Bedingung erfüllt, ist *nil* der Wert des *cond*

Beispiel

```
> (defun my-reverse (seq)
  (cond ((null seq) seq)
        (t (append (my-reverse (cdr seq))
                     (list (car seq))))))

> (my-reverse '(1 2 3 4 5))
(5 4 3 2 1)
```

- Rekursive Funktion
- Keine Variablenzuweisung

Funktionale Programmierung

- Einfache, möglichst universelle Datentypen (wie Listen)
- Funktionen, die auf diesen Datentypen arbeiten (wie mapcar)
- Funktionen als Parameter anderer Funktionen
- Zustand (set...) und Seiteneffekte werden möglichst vermieden oder an wenigen Stellen konzentriert
- Konsequenz: „Variablen“ eher als Konstanten betrachtet
- Funktionen häufig rekursiv programmiert

Übung: Lisp-Funktion

- Schreiben Sie eine Funktion, die alle Zahlen einer Liste quadriert, also eine neue Liste mit den Quadratzahlen erzeugt
- Beispiel (und zwei Hinweise):

```
> (list-sqr '(5 2 8 7))  
(25 4 64 49)
```

```
> (cons 99 '(1 2 3 4))  
(99 1 2 3 4)
```

```
> (expt 7 2)  
49
```

Lisp Programm

- Ein Lisp Programm kann interaktiv im Interpreter eingegeben und getestet werden
- Der Normalfall ist aber, das Programm in einer Datei abzulegen und zur Ausführung zu laden
- Datei *test.lisp*:

```
;; Beispieldatei test.lisp  
(print "Starte Ladevorgang...")  
(defun doppelt (a)  
  (* 2 a))  
(print "Fertig.")
```

Lisp Programm

```
> (load "~/test.lisp")  
;; Loading file /Users/burkert/test.lisp ...  
"Starte Ladevorgang..."  
"Fertig."  
;; Loaded file /Users/burkert/test.lisp  
T  
  
> (doppelt 4)  
8  
  
> (load "~/test.lisp" :verbose nil)  
"Starte Ladevorgang..."  
"Fertig."  
T
```

Kommentare

```
;; Kommentare beginnen mit einem Semikolon und  
;; gelten bis zum Zeilenende
```

```
;; Fakultatsfunktion  
;; Achtung: Kein Test, ob korrektes Argument übergeben
```

```
(defun fakultaet (n)  
  (if (= n 0)  
      1 ; Fakultat von 0 ist 1  
      (* n (fakultaet (- n 1)))))
```

Ziel erreicht?

- Sie haben einen ersten Eindruck von Lisp und seiner Syntax
- Sie kennen die wichtigsten Datentypen von Common Lisp
- Sie wissen, wie Ausdrücke aufgebaut sind und was das *Evaluieren* von Ausdrücken bedeutet
- Mit Hilfe von Funktionen können Sie einfache Lisp-Programme schreiben

Übersicht

- Lisp – Was soll das sein ??
- Ausdrücke und Datentypen
- Kleine Programme
- Auftrag, Quellen, weitere Informationen
- Anhang: Common Lisp Implementierungen

Aufträge zum Selbststudium

1. Für das Praktikum benötigen Sie eine Common-Lisp-Installation, verschiedene Varianten dazu im Anhang.
2. Das Standardwerk zu Common Lisp ist „*Common Lisp the Language, 2nd Edition*“ (s. nächste Seite). Verschaffen Sie sich einen Überblick über dessen Aufbau. Suchen Sie die Beschreibung zu einigen der behandelten Funktionen und Spezialformen, mindestens *cdr*, *append*, *cond*, *cons*.
3. Empfohlen: Lesen Sie die Lektionen 1 bis 5 von „*Chapter 3 - Essential Lisp in Twelve Lessons*“ (David B. Lamkins):
<http://successful-lisp.blogspot.ch>
(Zip-File bei den PSPP-Unterlagen)

Online Ressourcen: Bücher

- Guy L. Steele: Common Lisp the Language, 2nd Edition (1990)

<https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>

Wichtige Quelle: Am besten
Lesezeichen setzen

- Successful Lisp:
How to Understand and Use Common Lisp

<http://successful-lisp.blogspot.ch>

- Practical Common Lisp

<http://www.gigamonkeys.com/book/>

- The Common Lisp Cookbook

<http://cl-cookbook.sourceforge.net/index.html>

Online Ressourcen: Sonstiges

- Common Lisp Hints
<http://www.n-a-n-o.com/lisp/cmuccl-tutorials/LISP-tutorial.html>
- Beating The Averages
<http://www.paulgraham.com/avg.html>
- Common Lisp - Myths and Legends
http://www.lispworks.com/products/myths_and_legends.html
- Chaosradio Express Podcast CRE084 (9.4.2008)
Lisp – Die Mutter aller Programmiersprachen
<http://cre.fm/cre084>

Myths About Lisp

Myth #1: Lisp is slow

Myth #2: Lisp is big

Myth #3: Lisp has no arrays

Myth #4: Lisp has no compiler

Myth #5: Lisp is not standard

Myth #6: Lisp doesn't talk to other programs

Myth #7: Lisp syntax is painful

Myth #8: Lisp GC is slow

Myth #9: Lisp needs special hardware

Myth #10: Lisp is expensive



Quelle: http://www.lispworks.com/products/myths_and_legends.html

Übersicht

- Lisp – Was soll das sein ??
- Ausdrücke und Datentypen
- Kleine Programme
- Auftrag, Quellen, weitere Informationen
- Anhang: Common Lisp Implementierungen

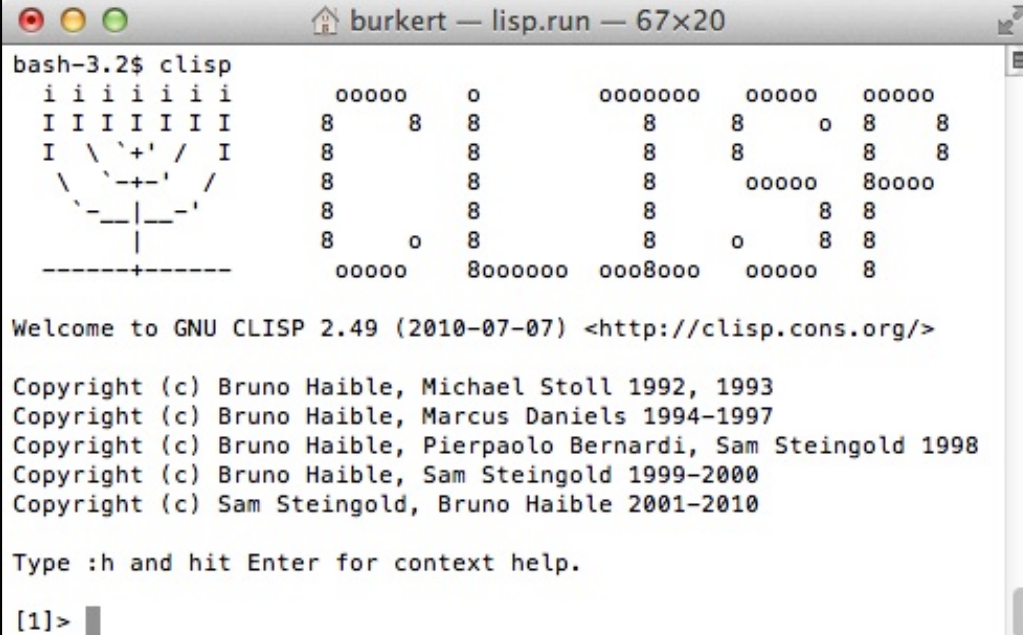
Aufgabe: Infrastruktur

- Für das Praktikum benötigen Sie eine Common-Lisp-Installation
- Eine Liste der Alternativen finden Sie bei den Praktikumsunterlagen
- Einfachster Start mit REPL in der Kommandozeile und Code-Editor mit Lisp-Unterstützung
- Interaktiv im Web:
<https://jscl-project.github.io>

CLISP

- Linux, Windows, Mac
- Open Source

- Quellen
<http://clisp.org>



```
bash-3.2$ clisp
i i i i i i i      ooooo  o      oooooooo  ooooo  ooooo
I I I I I I I      8      8  8      8      8  o  8      8
I \ \ '+ ' / I      8      8      8      8      8  8      8
 \ \ \ \ / / \      8      8      8      ooooo  8oooo
  \ \ \ \ / / \      8      8      8      8      8  8
   \ \ \ \ / / \      8      o  8      8      o  8      8
  -----+-----      ooooo  8ooooooo  ooo8ooo  ooooo  8

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

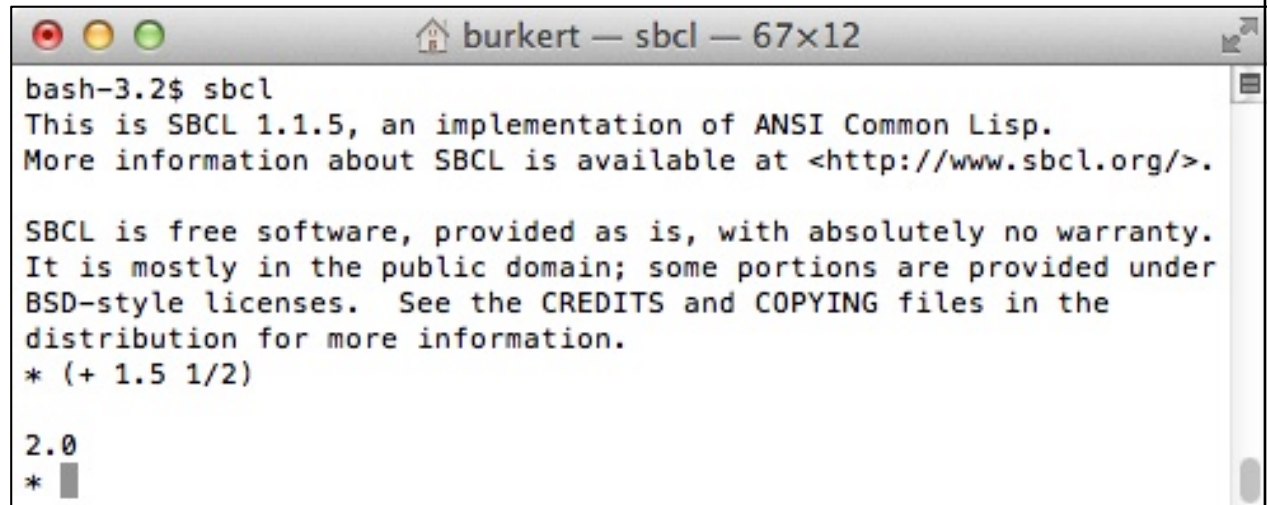
[1]>
```

- Auf dem Mac am besten mit einem Package Manager wie *Macports* oder *Homebrew* installieren
- Windows: Version mit integriertem *MinGW* von der Projektseite laden: <http://sourceforge.net/projects/clisp/>

SBCL

- Steel Bank Common Lisp
- Open Source
- Verschiedene Plattformen
- Download

<http://www.sbcl.org>

A screenshot of a terminal window titled 'burkert — sbcl — 67x12'. The terminal shows the command 'bash-3.2\$ sbcl' and the output of the SBCL 1.1.5 startup message. The message includes information about the software being free and in the public domain, and provides a link to the SBCL website. The prompt '2.0' and a cursor are visible at the bottom.

```
bash-3.2$ sbcl
This is SBCL 1.1.5, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* (+ 1.5 1/2)

2.0
* █
```

Clozure Common Lisp

- Common Lisp mit einer langen Geschichte

1984 Coral Common Lisp, CCL (Mac)

... Macintosh Allegro Common Lisp, MACL

1988 Macintosh Common Lisp, MCL

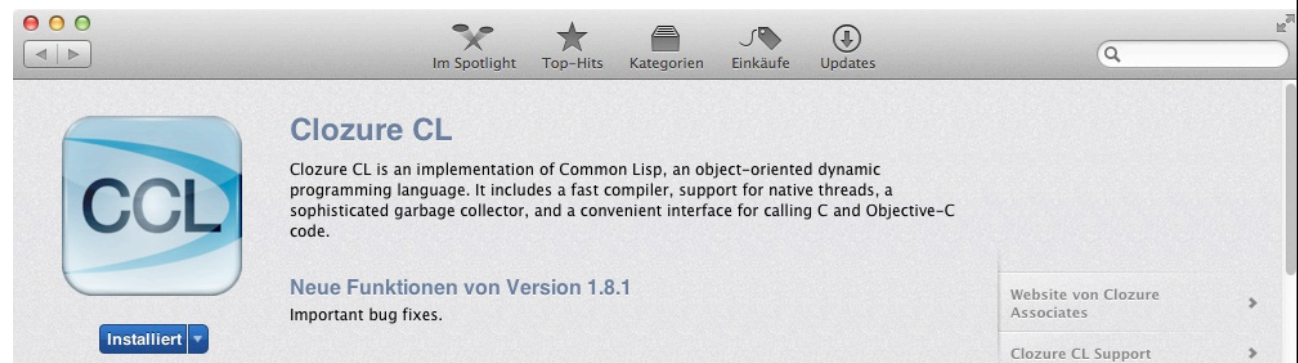
2001 OpenMCL

2007 Clozure CL

- Heute: Freies Common Lisp für Mac, Linux, Windows

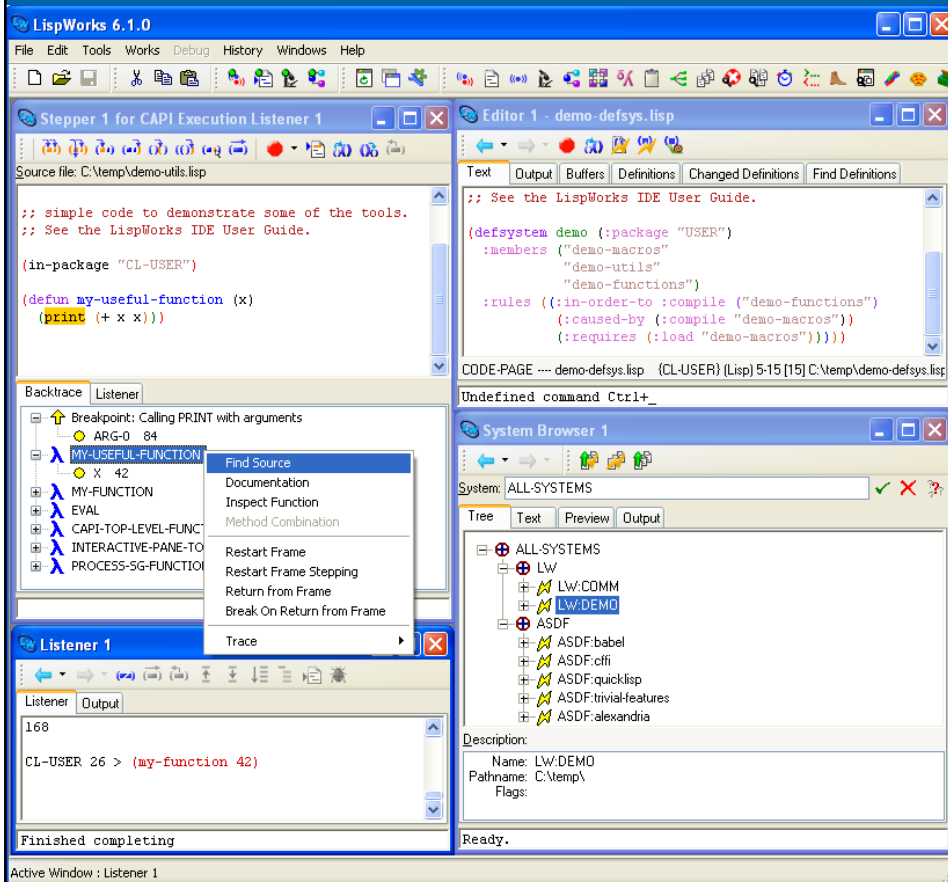
<http://ccl.clozure.com>

- Mac: auch via
App Store



LispWorks

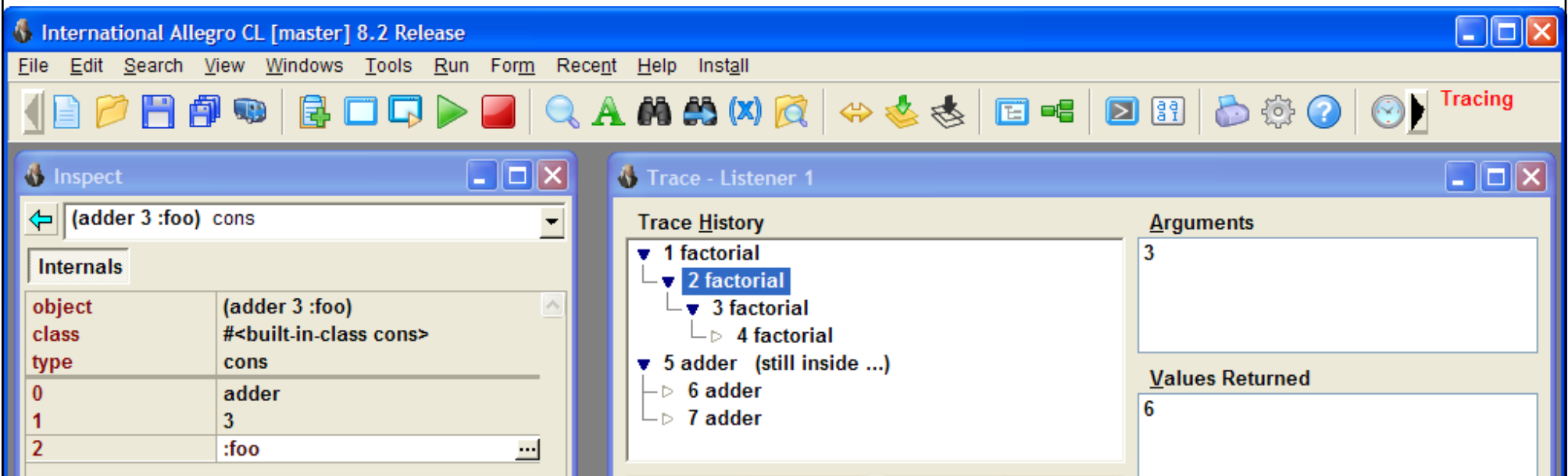
- Entwicklungsumgebung für verschiedene Plattformen
- Kommerzielles Produkt, kostenlose, eingeschränkte *Personal Edition*



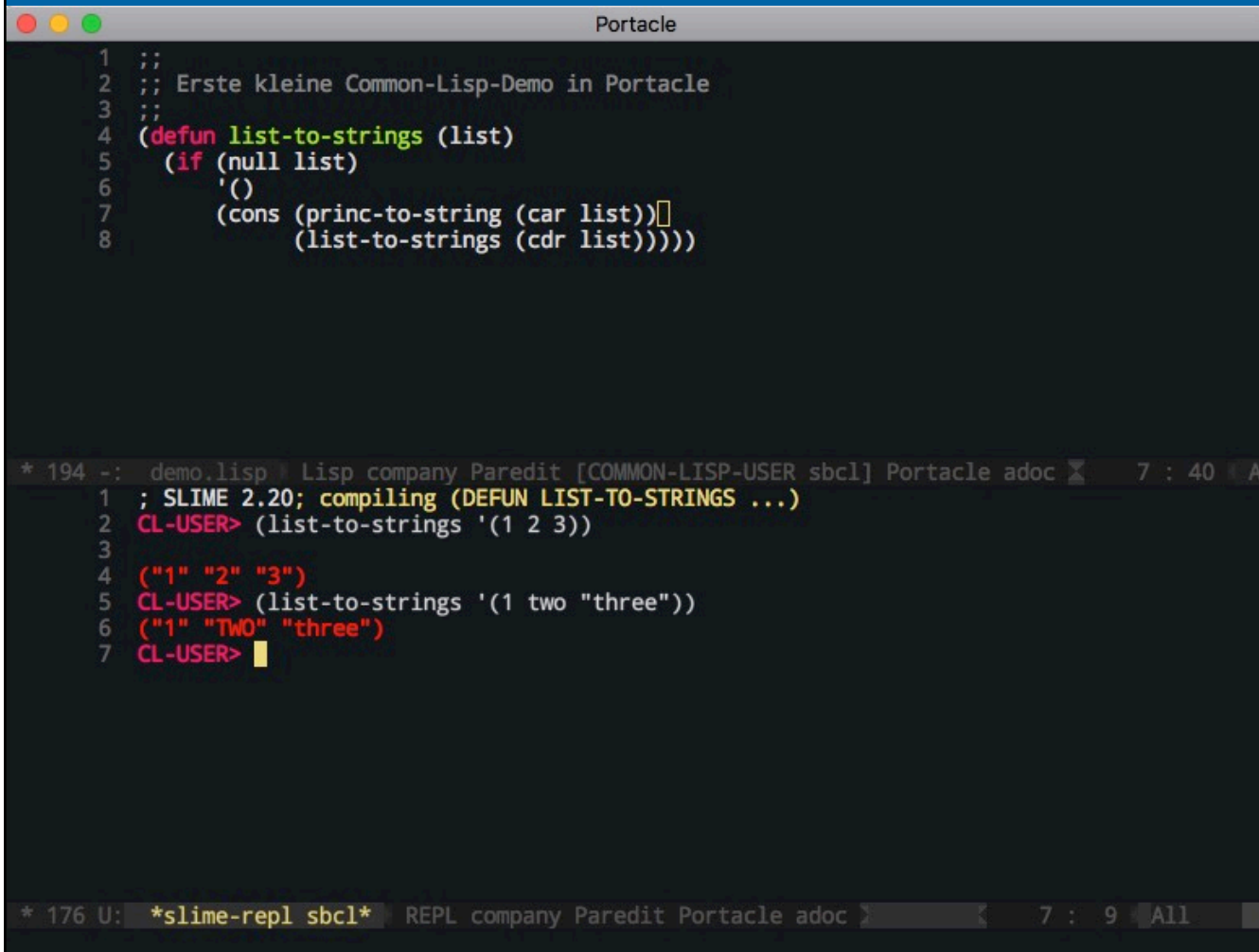
<http://www.lispworks.com/products/lispworks.html>

Allegro Common Lisp

- Kommerzielles Produkt
- Es gibt aber eine „Free Express Edition“ für bestimmte Zwecke (u.a. Ausbildung)
- <http://www.franz.com/products/allegro-common-lisp/>



Portacle



```
Portacle

1 ;;
2 ;; Erste kleine Common-Lisp-Demo in Portacle
3 ;;
4 (defun list-to-strings (list)
5   (if (null list)
6       '()
7       (cons (princ-to-string (car list))
8             (list-to-strings (cdr list)))))

* 194 -: demo.lisp Lisp company Paredit [COMMON-LISP-USER sbcl] Portacle adoc 7 : 40 A
1 ; SLIME 2.20; compiling (DEFUN LIST-TO-STRINGS ...)
2 CL-USER> (list-to-strings '(1 2 3))
3
4 ("1" "2" "3")
5 CL-USER> (list-to-strings '(1 two "three"))
6 ("1" "TWO" "three")
7 CL-USER>

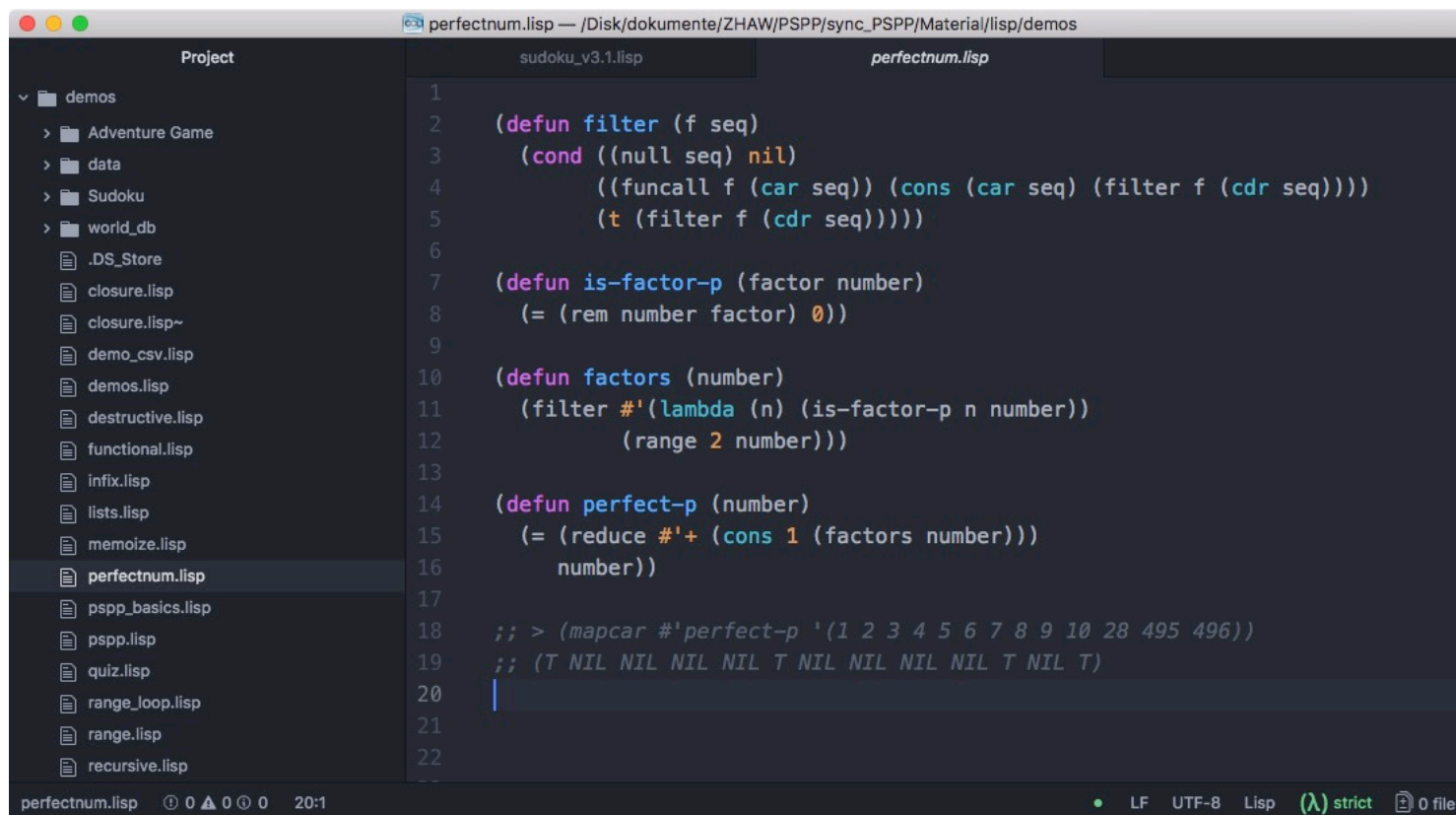
* 176 U: *slime-repl sbcl* REPL company Paredit Portacle adoc 7 : 9 All
```

- Common Lisp IDE
- Multi platform
- Emacs, SBCL

<https://portacle.github.io>

Lisp Mode im Atom-Editor

- Pakete *language-lisp* und *lisp-paredit*
- Zahlreiche weitere Optionen



```
1
2 (defun filter (f seq)
3   (cond ((null seq) nil)
4         ((funcall f (car seq)) (cons (car seq) (filter f (cdr seq))))
5         (t (filter f (cdr seq)))))
6
7 (defun is-factor-p (factor number)
8   (= (rem number factor) 0))
9
10 (defun factors (number)
11   (filter #'(lambda (n) (is-factor-p n number))
12          (range 2 number)))
13
14 (defun perfect-p (number)
15   (= (reduce #'+ (cons 1 (factors number)))
16      number))
17
18 ;; > (mapcar #'perfect-p '(1 2 3 4 5 6 7 8 9 10 28 495 496))
19 ;; (T NIL NIL NIL NIL T NIL NIL NIL NIL T NIL T)
20
21
22
```