Lisp

by

Dr. Günter Kolousek

Einführung

- ursprünglich LISP
  - ”LISt Processor”
  - Erfinder: John McCarthy
  - Erste Version LISP1 (1959!)
- Interpretersprache
  - REPL: read-eval-print loop
  - auch Compiler!
- Verschiedene Varianten und Implementierungen
  - Common Lisp (meist verwendet, umfangreich und komplex)
  - Scheme (”klein”, meist in Lehre und Forschung)
  - Closure (basiert auf der JVM)
  - AutoLISP (verwendet in AutoCAD)
  - Emacs 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 (Autor, SW-Entwickler)

Charakteristik

- hauptsächliche Datenstruktur sind Listen
  - Auch der Programmcode!
    - d.h. Programmcode kann sich selbst modifizieren
- in der Regel dynamische, aber strenge Typisierung
- sowohl dynamisches als auch lexikalisches Scoping
  - d.h. wo Identifier sichtbar sind
- Einsatz: ”General Purpose”
  - aber (früher) hauptsächlich für AI

s-expression

- für symbolic expression
  - auch: *sexpr* oder *sexp*, Plural: *sexprs* bzw. *sexps*
- sind eine Notation für verschachtelte Listen
- Eine sexp ist
  - ein Atom
  - ein Ausdruck der Form `(x . y)`
    - wobei x und y wieder sexps sind
- Meist abgekürzte Form
  - `(x y)` ≡ `(x . (y . nil))`
    - damit ist `(x y z)` ≡ `(x . (y . (z . nil)))`
- Programm ist eine Folge von sexps
  - die ausgewertet werden

Atome

- Arten
  - Zahlen
  - Zeichen
  - Strings
  - Symbole
    - ”Variablen”!
    - spezielle Sympbole: `t` und `nil`

Auswertung

- Wert eines Atoms
  - Atome, die keine Symbole sind, evaluieren zu sich selbst
  - Symbole evaluieren zu dem zugewiesenen Wert
    - `t` und `nil` evaluieren zu sich selbst
- Wert einer sexp der Form `(x y)`
  - `x` wird als Funktion betrachtet
  - `y` wird ausgewertet und an Funktion `x` übergeben und diese ausgewertet

Beispiele

```
1 ; -> 1
(+ 1 2) ; -> 3
(+ 1 2 3 4 5) ; -> 15
(/ 60 2 3 5) ; -> 2
(* (+ 1 2) 3) ; -> 9
(< 2 3) ; -> t
(+ 1 a) ; -> Symbols's value as variable is void: a
(setf a 1) ; -> 1
(+ 1 a) ; -> 2
a ; -> 1
(quote a) ; keine Auswetung -> a
'a ; -> a (kürzer)
(setf mylist (list 1 2 3 4)) ; -> (1 2 3 4)
(setf mylist2 (1 2 3 4)) ; -> invalid function: 1
(setf mylist2 '(1 2 3 4)) ; -> (1 2 3 4)
```

Beispiele – 2

```
(first '(1 2 3 4)) ; -> 1
(rest '(1 2 3 4)) ; -> (2 3 4)
(last '(1 2 3 4)) ; -> (4)
(nth 1 '(1 2 3)) ; -> 2
(first (rest '(1 2 3 4))) ; -> 2
(cons 1 '(2 3 4)) ; -> (1 2 3 4)
(cons '(1 2) '(3 4)) ; -> ((1 2) 3 4)
(append '(1 2) '(3 4)) ; -> (1 2 3 4)
(append '(1) '(2 3) '(4)) ; -> (1 2 3 4)
(reverse '(1 2 3)) ; -> (3 2 1)
(remove '2 '(1 2 3)) ; -> (1 3)
(setf foo '(0 2 3)) ; -> (0 2 3)
(setf bar (cons 1 (rest foo))) ; -> (1 2 3)
(setf (nth 2 foo) 4) ; -> 4
bar ; -> (1 2 4)
```

Beispiele – 3

```
(stringp ”abc”) ; -> t
(numberp ”abc”) ; -> nil
(atom ?a) ; -> t (character a)
(booleanp nil) ; -> t
(booleanp 0) ; -> nil
() ; -> nil
(numberp 123) ; -> t
(symbolp 1) ; -> nil
(symbolp 'abc) ; -> t
(symbolp abc) ; -> t
(progn (setf abc 1) (symbolp abc)) ; -> nil
(setf abc 'def) ; -> def
```

Beispiele – 4

```
(null '()) ; -> t
(null nil) ; -> t
(null 1) ; -> nil
(null 0) ; -> nil
(null '(1 2 3)) ; -> nil
(zerop 0) ; -> t
(zerop 1) ; -> nil
(atom (first '(1 2 3))) ; -> t
(atom (first '((1 2) 2 3))) ; -> nil
(member 2 '(1 2 3)) ; -> (2 3)
(member 4 '(1 2 3)) ; -> nil
(equal nil ()) ; -> t
(equal '(1 . nil) '(1)) ; - > t
(equal '(1 2) '(1 . (2 . nil))) ; -> t
```

Beispiele – 5

```
(defun x2(x) (* x x)) ; -> x2
(x2 2) ; -> 4
(defun factorial (n)
   (if (= n 0) 1
       (* n (factorial (- n 1)))))
(factorial 6) ; -> 720
```

Beispiele – 6

```
(if t
    (list 1 2 3)
    (list 4 5 6)
) ; -> (1 2 3)
(if (< 3 2)
    (list 1 2 3)
) ; -> nil
(setf income 35000)
(cond ((< income 10000) 0.0)
      ((< income 30000) 0.2)
      ((< income 50000) 0.25)
      ((< income 70000) 0.3)) ; -> 0.25
```

Beispiele – 7

```
(setf x 0)
(loop
    (setf x (+ x 1))
    (when (> x 7) (return x))
) ; -> 8
(setf x 0)
(loop
    (when (>= x 7) (return x))
    (setf x (+ x 1))
) ; -> 7
(setq x 3)
(while (> x 0) (setq x (- x 1))) ; -> nil
x ; -> 0
(progn (setq x 1) (setq x (+ x 1))) ; -> 2
```

set vs. setq vs. setf

```
(set foo 1) ; -> error!!!
(set (quote foo) 1) ; -> 1
foo ; -> 1
(setq foo 2) ; -> 2
(setq foo 'bar) ; -> bar
(set foo 3) ; -> 3
bar ; ->3
(setq foo '(1 2 3)) ; -> (1 2 3)
(setf (car foo) 0) ; -> 0  weder set noch setq!!!
foo ; -> (0 2 3)
```

’ vs. `list` vs. ‘

```
(+ 1 2)  ; -> 3
'(+ 1 2)  ; -> (+ 1 2)
(list 1 (+ 1 2))  ; -> (1 2)
; backquote
`(+ 1 (+ 1 1))  ; -> (+ 1 (+ 1 1))
; evaluate marked argument
`(+ 1 ,(+ 1 1))  ; -> (+ 1 2)
(setq part '(2 3))  ; -> (2 3)
; evalueate and splice it into result list
`(1 ,@part 4 5)  ; -> (1 2 3 4 5)
```