

## Proseminar Programmiersprachen

# LISP

Peter Kiechle

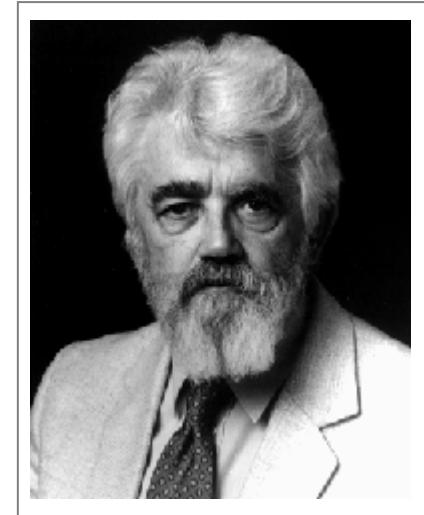
TU München

WS2006

# Inhalt

- ▶ Einführung
- ▶ Beispielprogramm
- ▶ Grundlegende Konstrukte
- ▶ Makros
- ▶ Objektorientierung

- ▶ Dartmouth Conference 1956
- ▶ 1958: Projektstart unter John McCarthy
- ▶ Zersplitterung in verschiedene Dialekte
- ▶ Erweiterung um Objektorientierung
- ▶ 1994: Standardisierung von Common Lisp  
(ANSI X3.266-1994)



John McCarthy

## Lisp ist...

### **funktional:**

- ▶ Funktionsaufrufe
- ▶ Rückgabewerte
- ▶ Rekursion

### **prozedural:**

- ▶ Seiteneffekte
- ▶ Kontrollstrukturen
- ▶ Schleifen

### **objektorientiert:**

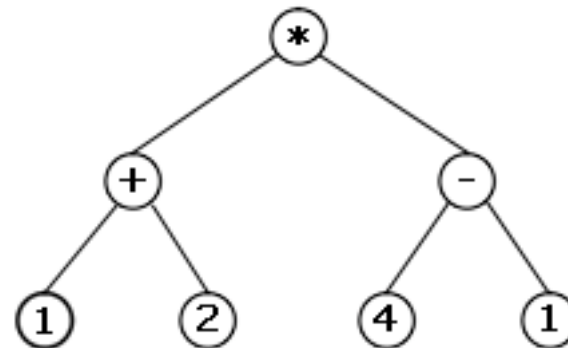
- ▶ Klassen und Instanzen
- ▶ Attribute und Methoden
- ▶ Vererbung

# Syntax

- ▶ **LIST Processing**

- ▶ **S-Expression**

`(* (+ 1 2) (- 4 1))`



- ▶ **Kein Unterschied zwischen Programmcode und Daten**

# Interpreter vs. Compiler

- ▶ read-eval-print-loop (REPL)
- ▶ Dynamisch kompilierende Interpretersprache
- ▶ Optional: vollständig kompilierbar

➔ Ideal für Rapid Prototyping

# Typisierung

## Strikte Typisierung:

- ▶ Effizienter Maschinencode
- ▶ Typfehler werden vom Compiler erkannt

## Schwache Typisierung:

- ▶ Freiheit des Programmierers

## Kompromiss in Lisp

- ▶ Standarddatentyp: Symbol
- ▶ Optional: Zahlen, Strings, Booleans usw.

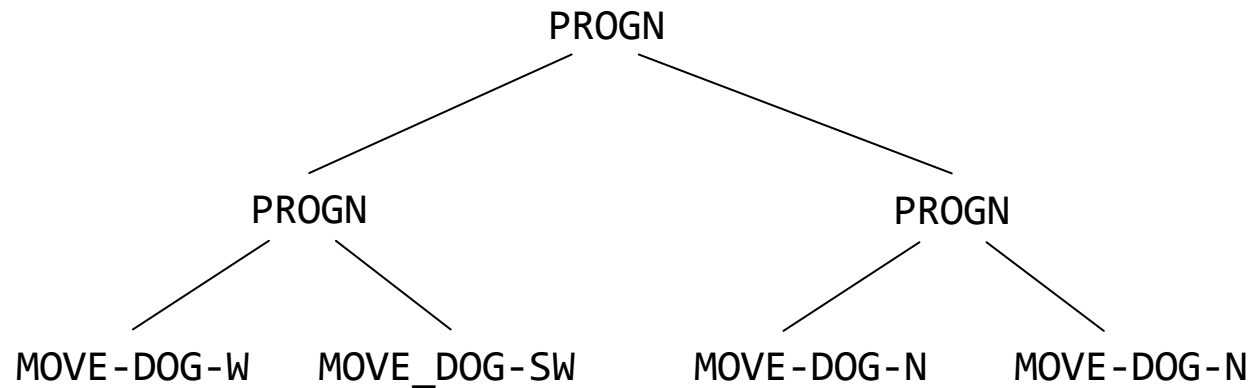
```
(defun plus-untyped (x y)
  (+ x y))
```

```
(defun plus-typed (x y)
  (declare (integer x y))
  (the integer (+ x y)))
```

# Anwendungsbereich

- ▶ General purpose
- ▶ Hauptanwendungsbereich: Künstliche Intelligenz

Beispielprogramm: Genetische Programmierung



# Forms

- ▶ S-Expressions != ausführbarer Code
- ▶ Forms sind evaluierbare S-Expressions

S-Expression

(1 2 3)

Lisp-Form

(+ 1 2 3)

Evaluationsreihenfolge:

„von innen nach aussen, von links nach rechts“

(\* (+ 1 2) (- 4 1))

1. (+ 1 2)

2. (- 4 1)

3. (\* 3 3)

# Atome

- ▶ S-Expressions bestehen aus Atomen und Listen

- ▶ Atome

- ▶ Selbstevaluierende Atome

1

„Hallo Welt!“

- ▶ Symbole

Name	← →	"X"
Package	← →	USER
Wert	← →	5
Funktion	← →	#<function>
Propertylist	← →	(farbe rot)

# Symbole

- ▶ Wertzuweisung

```
> (setf x 5)  
5
```

- ▶ Evaluation

```
> x  
5
```

- ▶ Evaluation verhindern

```
> (quote x)  
x
```

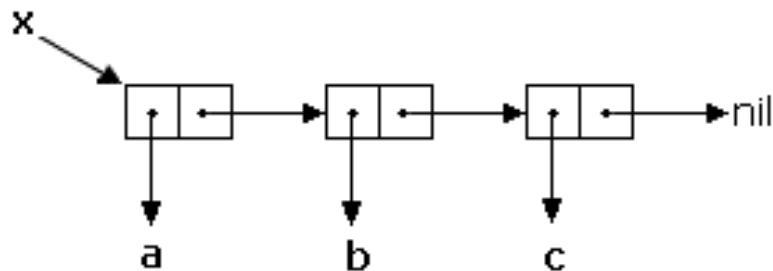
```
> 'x  
x
```

# Listen

- ▶ Listen bestehen aus Atomen und Listen

```
> (cons 'a (cons 'b (cons 'c nil)))  
(A B C)
```

```
> (setf x (list 'a 'b 'c))  
(A B C)
```



# Listen

- ▶ Zugriff auf die Elemente einer Liste:

- ▶ Erstes Element: 

```
> (car '(a b c))  
A
```

- ▶ Rest: 

```
> (cdr '(a b c))  
(B C)
```

- ▶ Letztes Element: 

```
> (car (cdr (cdr '(a b c))))  
C
```

- ▶ Alternativ: 

```
> (third '(a b c))  
C
```

# Funktionen

- ▶ First Class Objects
- ▶ Kein Unterschied zwischen eingebauten und selbst definierten Funktionen
- ▶ Keine Trennlinie zwischen Programmiersprache und Programm
- ▶ Bottom-Up Programmierung

# Funktionen

- ▶ Lambda-Kalkül

$\lambda x. x + 2$

- ▶ Anonyme Lisp-Funktion

```
(lambda (x) (+ x 2))
```

- ▶ Funktionsdefinition

```
(defun addiere-zwei (x)  
  (+ x 2))
```

```
> (addiere-zwei 5)  
7
```

# Funktionen

## ► Funktionen in Datenstrukturen speichern

```
> (setf myarray (make-array 5))  
#(NIL NIL NIL NIL NIL)
```

```
> (setf (aref myarray 0) (lambda (x) (+ x 2)))  
#(#<Interpreted Closure (unnamed) @ #x20c69d02>  
NIL NIL NIL NIL)
```

```
> (funcall (aref myarray 0) 5)  
7
```

## ► Funktionen höherer Ordnung

```
> (mapcar #'list '(A B C) '(1 2 3))  
((A 1) (B 2) (C 3))
```

# Rekursion

- ▶ Ideal für Listen als Datenstruktur

```
(defun print-elements (list)
  (if (null list) ;Abbruch wenn Liste leer
      (return-from print-elements nil))
  (print (car list))
  (print-elements (cdr list))) ;rekursiver Aufruf
```

```
> (print-elements '(a b c))
```

```
A
```

```
B
```

```
C
```

```
NIL
```

- ▶ Umwandlung von Endrekursion in Iteration durch den Compiler

# Iteration

► Mehrere Abstraktionsstufen

```
> (do ((i 1 (+ i 1)))  
      ((> i 3) 'ende)  
      (print i))
```

```
1  
2  
3  
ENDE
```

```
> (dolist (elem '(a b c))  
      (print elem))
```

```
A  
B  
C  
NIL
```

# Kontrollstrukturen

- ▶ Erste Sprache mit if-then-else Konstrukt
- ▶ if ist Special Form
- ▶ Kann aber auch durch Lambda-Funktion implementiert werden
- ▶ Die meisten Kontrollstrukturen sind Makros

```
(if objective-achieved  
    (incf hits)  
    (incf raw-fitness 999))
```

## Lisp is a programmable programming language

- ▶ Funktionen liefern Ergebnisse
- ▶ Makros generieren Funktionen, die Ergebnisse liefern
- ▶ Keine simple Textersetzung à la C und VBA
- ▶ Veränderung des Programmcodes zur Kompilierzeit
- ▶ Direkter Einfluss auf Evaluationsreihenfolge

# Makros

- ▶ Makrodefinition:

```
(defmacro set-nil (var)  
  (list 'setf var nil))
```

- ▶ Aufruf:

```
(set-nil bla)
```

- ▶ Expansion:

```
(list 'setf var nil)  ➡  (setf bla nil)
```

- ▶ Evaluation:

```
(setf bla nil)
```

# Makros

- ▶ Backquote

```
(list 'a 'b 'c)  ≡  '(a b c)  ≡  `(a b c)
```

- ▶ Evaluation an-/abschaltbar

```
(defmacro set-nil (var)  
  `(setf ,var nil)) ;Komma gibt Variablen frei
```

```
> (setf lst '(a b c))  
(A B C)
```

```
> `(Die Elemente von ,lst sind ,@lst)  
(DIE ELEMENTE VON (A B C) SIND A B C)
```

# Makros

## For-Makro

```
(defmacro for ((var start stop ) &rest body)
  `(do ((,var ,start (incf ,var))
        (limit ,stop))
        ((> ,var limit))
        ,@body))
```

```
> (for (x 1 5) (format t "~A " x))
1 2 3 4 5
NIL
```

# Makros

## Problem:

```
(for (limit 1 5) (format t "~A " limit))
```

```
> (macroexpand-1 '(for (limit 1 5) (format t "~A " limit)))
```

```
(DO ((LIMIT 1 (INCF LIMIT))  
    (LIMIT 5))  
    ((> LIMIT LIMIT))  
    (FORMAT T "~A " LIMIT))
```

**Lösung:** Eindeutiges Symbol in seperatem Namensraum

```
> (defmacro for ((var start stop ) &rest body)
  (let ((uniquelimit (gensym)))
    `(do ((,var ,start (incf ,var))
          (,uniquelimit ,stop))
        ((> ,var ,uniquelimit))
        ,@body)))
```

# Makros

```
(defmacro while-sheep (a b)
  `(let ((stepcount 0))
    (while (test-direction ,a sheep dog)
      (progn
        (when (already-there sheep) (return 'there))
        (eval ',b)
        (incf stepcount)
        (when (>= stepcount field-length) (return 'timeout))
      )
    )
  )
)
```

## Common Lisp Object System

- ▶ ANSI-Standard
- ▶ Erweiterung von Lisp um Objektorientierung
- ▶ Philosophiekonflikt mit C++/Java etc.
- ▶ Klassen und Methoden zur Laufzeit änderbar

## Klassendefinition

```
(defclass kreis ()  
  (radius))
```

## Instanziierung

```
(setf x (make-instance 'kreis))  
  
(setf (slot-value x 'radius) 3)
```

## Zugriff auf Slots

```
(defclass kreis ()  
  (radius :accessor get-radius  
          :initarg :radius  
          :initform 1))
```

```
> (setf x (make-instance 'kreis))
```

```
> (get-radius x)
```

```
1
```

```
> (setf x (make-instance 'kreis :radius 3))
```

```
> (get-radius x)
```

```
3
```

## Nachrichtenmodell vs. Generische Funktionen

`x.getRadius();`                      `(get-radius x)`

```
(defmethod area ((x circle)) ;Methode für Kreise
  (* pi (expt (get-radius x) 2)))
```

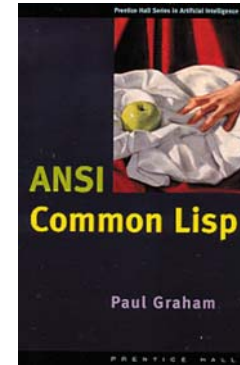
```
(defmethod area ((y rectangle)) ;Methode für Rechtecke
  (* (get-height y)(get-width y)))
```

```
(defmethod area ((x circle)(y rectangle))
  (+ (* pi (expt (get-radius x) 2))
     (* (get-width y) (get-height y))))
```

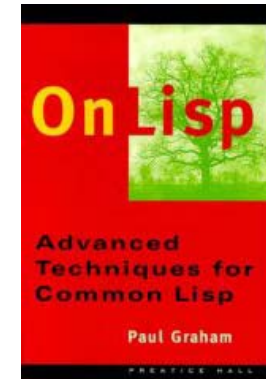
# Fragen?



ANSI Common LISP, Paul Graham (1996)



On LISP: Advanced Techniques for Common LISP,  
Paul Graham (1994)



Practical Common Lisp, Peter Seibel (2005)

