Kompaktkurs: Common Lisp

Mathis Löthe, Bernd Raichle
und weitere Mitarbeiter
der Abteilung Intelligente Systeme
am Institut für Informatik
der Universität Stuttgart

Wintersemester 1996/97

aktualisiert und korrigiert WS98/99, WS1999/2000

Inhaltsverzeichnis

3 Lisp-Spezialitäten

4 Datentypen

5 Programmiertechniken

6 Ergänzungen

A Readermakros (Addendum)

Um 1990 entwarfen einige Mitarbeiter der damals noch jungen Abteilung „Intelligente Systeme“ des Instituts für Informatik die erste Version des vor Ihnen liegenden Umdrukkes eines Kompaktkurses „Common Lisp“. Seit dieser ersten Version gingen die Folien durch die Hände fast jedes neuen Mitarbeiters der Abteilung, wobei jeder an dieser und jener Stelle etwas änderte, Teile hinzufügte, verschob und entfernte, Fehler korrigierte und auch neue Fehler einfügte.

Für das Wintersemester 1996/97 haben wir, Mathis Löthe und Bernd Raichle, nun begonnen, den Stoff und die Folien zu vereinheitlichen und dabei gleichzeitig von MS-Word auf LaTeX umzustellen. In diesem Schritt wurden zu den Folien auch zusätzliche, wenn auch bisher noch wenige Notizseiten erstellt, die den Stoff, der auf den Folien behandelt wird, durch Erklärungen, Merkregeln, weitere Beispiele, Hinweise auf Literatur u. ä. vertiefen sollen.

Anregungen und Verbesserungen, Fehlerkorrekturen, zusätzliche Anmerkungen und Hinweise sind jederzeit willkommen!

Stuttgart, im Herbst/Winter 1996/1997
Bernd Raichle
Mathis Löthe

Literatur

1. Steele, Guy L. Jr.: Common Lisp – the Language (2. Auflage, Digital Press, 1990)

   Referenzhandbuch und Nachschlagewerk

2. Winston, Patrick Henry; Horn, Berthold Klaus: Lisp (3. Auflage, Addison-Wesley, 1989)

   Lehrbuch

3. Keene, Sonya E.: Object-oriented Programming in Common Lisp: a Programmer's Guide to CLOS (Addison-Wesley, 1989)

   Einführendes Lehrbuch in das Common Lisp Object System CLOS

4. Steele, David J.: Golden Common Lisp – a hands-on Approach (Addison-Wesley, 1989)

   Einführendes Lehrbuch für die Sprache, behandelt auch fortgeschrittene Konstrukte.

5. Yuasa, Taiichi; Hagiya, Masami: Introduction to Common Lisp (Academic Press, 1987)

   Einführendes Lehrbuch

6. Norvig, Peter: Paradigms of Artificial Intelligence Programming – Case studies in Common Lisp (Morgan Kaufmann Publ., 1992)

   Lehrbuch über Programmiertechniken

7. Lisp-Seite der Abteilung IS, Institut für Informatik, Uni Stuttgart mit Dokumenten und Verweisen in alle Welt:

   `http://www.informatik.uni-stuttgart.de/ifi/is/Lisp/lisp.html`

8. Novig, Peter; Pitman, Kent: Tutorial on Good Lisp Programming Style

   Satz von 116 Folien, der auf obiger WWW-Seite als Postscriptdatei erhältlich ist.

Teil 1

Einführung

- Lisp – Sprache und Geschichte
- Grundbegriffe und Evaluierung
- Definition einer Funktion
- Einfache Listenverarbeitung

Lisp als Programmiersprache

Prinzipien:

- Grundparadigma: applikativ funktional
- Paradigmizität: verschiedene Paradigmen möglich
  vorhanden: prozedural, echt funktional, objektorientiert
  deskriptive Paradigmen (z.B. Logik) benötigen
  Erweiterungen
- alle Sprachkomponenten sind zur Laufzeit verfügbar
- (fast) alle Konstrukte können vom Benutzer erstellt werden

Aufbau:

- kompakter Kern: Evaluator
- viele standardisierte vordefinierte Funktionen
- dynamische Typisierung
- Code wird durch Symbole und Listen repräsentiert
- weitere Datentypen für den Anwender
- bequeme Funktionen für Ein- und Ausgabe

Geschichte von Lisp

- steht für **lis**t **p**rocessing
- wurde 1958 von John McCarthy entwickelt
- wird in der KI-Programmierung verwendet
- wurde vor einigen Jahren noch durch spezielle Hardware unterstützt (Lisp-Maschinen)
- wurde in vielen Dialekten verwendet: Common Lisp, MacLisp, Interlisp, UCI-Lisp, PSL, FranzLisp, Scheme, NIL, T, Zetalisp
- heute: Common Lisp und Scheme

Common Lisp

- Sprachdefinition (keine Definition einer Implementierung)
- von vielen KI-Wissenschaftlern gemeinsam verfaßt

Vorteile

- interaktive Programmierung
- sehr gute Programmierumgebungen
- entwickelt für Symbolverarbeitung
- Äquivalenz von Programmen und Daten

Auswerter – Evaluator

```
> 5
5
> (+ 5 7)
12
> (+ 5 (* 2 3))
11
> (+ 5 2 3)
10
```

- Lisp wertet Ausdrücke aus.
- Jeder Ausdruck hat einen Wert.
- Ausdrücke sind atomar oder Listen.
- Funktionsaufruf ist Liste aus Funktionsname und Argumenten.
  $\Rightarrow$ **Präfix**notation für Funktionen
- Geschachtelte Ausdrücke möglich.
- Zuerst werden die Argumente ausgewertet, dann die Funktion gerufen.

Begriffe

Zahlen Beispiel: 3 4 5 6

Symbole Symbole sind Namen wie

`anna eva petra + * anna+*123`

Atome U.a. sind Zahlen und Symbole Atome.

Listen sind keine Atome.
Listen beginnen mit einer `(`, dann folgen die Listenelemente und zum Schluß kommt eine `)`.
Die Elemente sind beliebige Lisp-Objekte.

Ausdrücke Atome und Listen bilden symbolische Ausdrücke (s-expressions).

Forms sind Ausdrücke, die ausgewertet werden sollen.

Auswertung (Evaluation) ist die Berechnung des Werts eines Ausdrucks.

Funktionen: vordefiniert und benutzerdefiniert

Programme bestehen aus Funktionen, die sinnvoll zusammenarbeiten.

Symbole als Namen

```
> (setq x 5)
5
> x
5
> (+ x 7)
12
> y
Error: unbound variable y
> (* 5 y)
Error: unbound variable y
> (y 5 7)
Error: y is an undefined function
```

Symbole als Variablennamen

- `setq` weist einem Symbol einen Wert zu.
- Ein Symbol evaluiert zu seinem Wert.
- Symbole können auch in Ausdrücken auftreten.

Symbole als Funktionsnamen

An erster Position im Ausdruck werden Symbole als Funktionsnamen interpretiert.

Quote

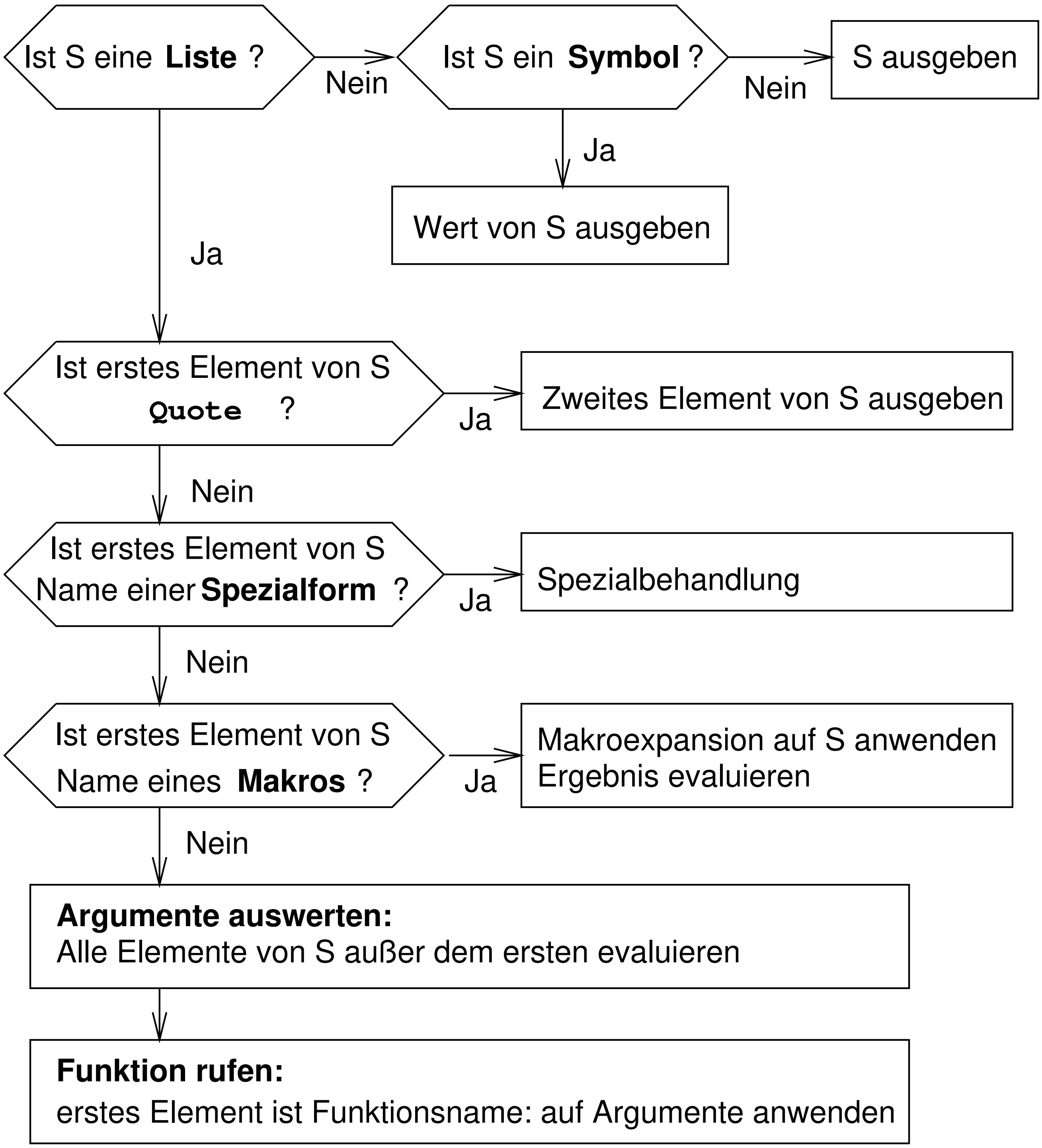
	`> (quote x)`
	`x`
`> x`	`> 'x`
`5`	`x`
`>(setq y x)`	`> (setq y 'x)`
`5`	`x`
`> y`	`> y`
`5`	`x`
`> (+ x 7)`	`> (+ 'x 7)`
`12`	`Error: x is not a number`
`> (+ 5 7)`	`> '(+ 5 7)`
`12`	`(+ 5 7)`
`> (+ y 7)`	`> ('+ 5 7)`
`12`	`Error: Illegal function object`

Gequoteter Ausdruck: `(quote` ⟨*ausdruck*⟩`)`

- Der Ausdruck ⟨*ausdruck*⟩ wird zurückgegeben.
- Kurznotation für `(quote` ⟨*ausdruck*⟩`)` ist `'`⟨*ausdruck*⟩.

Evaluationsregeln Common Lisp

Auswertung des Ausdrucks S:



Definition von Funktionen

```
> (defun quadrat (x)
    (* x x))
QUADRAT
> (quadrat 10)
100
> (quadrat (+ 5 7))
144
> (defun quadrat (x)
    (+ x x)
    (* x x))
> (quadrat 10)
100
```

Syntax:

`(defun` ⟨*name*⟩ ⟨*lambda-list*⟩ ⟨*ausdruck*⟩*`)`

- `defun` definiert eine Funktion.
- Die Lambda-Liste legt die Parameter fest.
- Der Rumpf besteht aus Ausdrücken.
- Die Ausdrücke dürfen die Parameter verwenden.
- Beim Aufruf wird der Rumpf der Funktion ausgeführt.
- Der Wert des **letzten** Ausdrucks wird zurückgegeben.

Funktionen mit Nebenwirkung

```
(defun hallo ()
  (print 'hallo))

(defun hallo-quad (x)
  (print 'hallo)
  (* x x))

> (hallo)
HALLO
HALLO

> (hallo-quad 10)
HALLO
100
```

```
(defun hallo-welt ()
  (print 'hallo)
  (print 'welt))
(defun arg-quad (x)
  (print x)
  (* x x))

> (hallo-welt)
HALLO
WELT
WELT
> (arg-quad 10)
10
100
> (+ (arg-quad 10) 22)
10
122
```

- `print` druckt sein Argument (Nebenwirkung).
- `print` gibt das Argument auch zurück.
- Alle Ausdrücke eines Prozedurrumpfs werden ausgewertet.
- Der Wert des letzten Ausdrucks wird zurückgegeben.

Listenverarbeitung

```
(setq za '(1 2 3)
      za2 '((1 2) (11 12)))
```

```
> (first '(1 2 3))
1
> (first za)
1
> (first (rest za))
2
> (first '(anna))
ANNA
> (first za2)
(1 2)
> (first (first za2))
1
> (first '())
NIL
> ()
NIL
> (+ 10 (first (rest za)))
12
```

```
> (rest '(1 2 3))
(2 3)
> (rest za)
(2 3)
> (rest (first za))
Error: 1 is not a listp
> (rest '(anna))
NIL
> (rest za2)
((11 12))
> (rest (rest za2))
NIL
> (rest '())
NIL
> (rest ())
NIL
```

- `first` gibt das erste Element der Liste zurück.
- `rest` gibt die Liste ohne das erste Element zurück.
- Die leere Liste `()` wird auch mit `NIL` bezeichnet.
  Sie evaluiert zu sich selbst.
- Listen dürfen geschachtelt sein.

`car` und `cdr`

```
(setq za '(1 2 3)
      za2 '((1 2) (11 12)))

> (car za)                      > (cdr za)
1                               (2 3)
> (car (cdr za))                > (cadr za)
2                               2
> (second za)
2
> (cdr (car za))                > (cdar za)
Error: 1 is not a listp         Error: 1 is not a listp
> (caddr za)                    > (third za)
3                               3
> (car za2)                     > (cdr za2)
(1 2)                           ((11 12))
> (caar za2)                    > (cddr za2)
1                               NIL
> (+ 10 (car (cdr za)))
12
```

- `car` gibt wie `first` das erste Element der Liste zurück.
- `cdr` und `rest` geben Liste ohne erstes Element zurück.
- Geschachtelte Aufrufe von `car` und `cdr` zu Aufruf der Form $\mathtt{c}\langle x\rangle\mathtt{r}$, $\langle x\rangle = \mathtt{a}|\mathtt{d}$ (maximal 4 Zeichen) kombinierbar.

Listenkonstruktoren

```
(setq za '(1 2 3)
      fr '(anna eva petra))
```

```
> (cons 0 za)
(0 1 2 3)
> (first (cons 0 za))
0
> (cons 1 2)
(1 . 2)
```

```
> (cons 0 '())
(0)
> (rest (cons 0 za))
(1 2 3)
> (cons '(1 2) 3)
((1 2) . 3)
```

```
> (list 1 2 3)
(1 2 3)
> (list 1 '() 2)
(1 NIL 2)
> (first (list 1 2 3))
1
```

```
> (list za)
((1 2 3))
> (list ())
(NIL)
> (rest (list 1 2 3))
(2 3)
```

```
> (append za fr)
(1 2 3 ANNA EVA PETRA)
> (append fr (list 'ina))
(ANNA EVA PETRA INA)
```

```
> (append za '())
(1 2 3)
```

Vergleich von `cons`, `list`, `append`

```
(setq za '(1 2 3)
      fr '(anna eva petra))

> (cons za fr)
((1 2 3) ANNA EVA PETRA)
> (cons za '())                > (cons '() '())
((1 2 3))                      (NIL)

> (list za fr)
((1 2 3) (ANNA EVA PETRA))
> (list za '())                > (list '() '())
((1 2 3) NIL)                  (NIL NIL)

> (append za fr)
(1 2 3 ANNA EVA PETRA)
> (append za '())              > (append '() '())
(1 2 3)                        NIL
```

- `cons` **fügt** ein Element vorne in eine Liste **an**.
- `list` **erzeugt** eine **neue** Liste aus seinen Argumenten.
- `append` **hängt** zwei Listen **zusammen**.

Teil 2

Grundlagen

- Variablen
- Funktionen als Datenobjekte
- Logik und Prädikate
- Verzweigungen
- Iterationen

Variablen und Gültigkeitsbereiche

Prinzip der im Gültigkeitsbereich **gebundenen** Variable:

Der Name ist (außen) unerheblich

```
(defun quadrat-neu (wert)
  (setq wert (* wert wert))    ;; lokal setzen
  (list 'ergebnis wert))

> (setq *wert-global* 10
        wert 77)               ;; globale Bindung
77
> (quadrat-neu *wert-global*)
(ERGEBNIS 100)
> *wert-global*
10
> wert
77
```

- In Lisp werden als Parameter *Werte* übergeben (call-by-value).
- Funktion hat einen eigenen Gültigkeitsbereich für ihre Parameter.
- Die Parameter sind außen nicht bekannt.
- Bei Änderung von gebundenen Variablen bleibt die Funktion nebenwirkungsfrei.

`let` und `let*`

```
> (setq daten '(1 2 3))
> (let ((wert1 (second daten))
        (wert2 (third daten)))
     (list (* wert1 wert1) (* wert2 wert2)))
(4 9)
```

```
(defun probe (x)
  (let ((x 10)
        (y (+ x 5)))
    y))
> (probe 20)
25
```

```
(defun probe* (x)
  (let* ((x 10)
         (y (+ x 5)))
    y))
> (probe* 20)
15
```

Syntax:

`(let (`$\{$`(`$\langle$*var*$\rangle$ $\langle$*form*$\rangle$`)`$\}^*$`)` $\langle$*form*$\rangle^*$`)`

- `let` und `let*` öffnen einen eigenen Gültigkeitsbereich.
- Zweck: lokale Hilfsvariablen
- evaluieren den ganzen Rumpf
- geben Wert des letzten Ausdrucks zurück.
- `let` bindet parallel
- `let*` bindet sequentiell

Gültigkeitsbereich (scope)

Bei geschachtelten Gültigkeitsbereichen beginnt die Suche nach der Variable beim innersten Bereich und endet beim äußersten (der globalen Umgebung).

Es gibt zwei Vorgehensweisen

lexikalisches Scoping: (Common Lisp, Scheme)

- Gültigkeitsbereiche durch Schachtelung des Codes definiert.
- Verfügbare Bindungen vor dem Programmlauf bekannt, daher auch „statisches“ Scoping

dynamisches Scoping: (einige alte Lispdialekte)

- Gültigkeitsbereiche durch Aufrufhierarchie definiert.
- Verfügbare Bindungen erst zur Laufzeit bekannt.
- Bindungen davon abhängig, **wo** eine Funktion gerufen wird. **Fehlerquelle!!!**
- Für `special` deklarierte Variablen gilt dynamisches Scoping.

Globale Variablen

- Globale Variablen können wie bisher einfach mit `setq` zugewiesen werden.
- Aus Dokumentationsgründen definiert man sie mit `defvar`
- Man benennt sie traditionell mit $*\langle\mathit{variable}\rangle*$

```
> (defvar *x* 15)
*x*
> *x*
15
> (defvar *x* 16)
*x*
> *x*
15
> (setq *x* 16)
16
> *x*
16
```

- `defvar` kann man nur einmal zur Definition verwenden.
- Später weist man Variablen mit `setq` zu.
  `setq` ohne `defvar` tut auch

Funktionen als Datenobjekte

Definition und Aufruf einer benannten Funktion:

```
> (defun flaeche (laenge breite)
    (* laenge breite))
FLAECHE
> (flaeche 20 5)
100
> flaeche
Error: unbound variable FLAECHE
```

Aufruf einer unbenannten Funktion:

```
> ((lambda (laenge breite) (* laenge breite)) 20 5)
100
> (lambda (laenge breite) (* laenge breite))
Error: ‘lambda’ is an undefined function
```

Syntax:

`(lambda` ⟨*lambda-liste*⟩ ⟨*ausdruck*⟩*`)`

- Mit `lambda` definiert man unbenannte Funktionen.
- Der Lambda-Ausdruck kann an Stelle eines Funktionsnamens stehen.
- Ein Lambda-Ausdruck kann nicht ausgewertet werden.

Aufruf der Funktionsobjekte

`#'` erzeugt aus einem Funktionsnamen oder einem Lambda-Ausdruck ein Funktionsobjekt.

`funcall` ruft ein Funktionsobjekt mit Argumenten auf:

```
> (funcall #'flaeche 20 5)
100
> (setq f1 #'flaeche)
#<Interpreted Function FLAECHE>
> (funcall f1 20 5)
100
> (setq f2 #'(lambda (l b) (* l b)))
#<Interpreted Function (unnamed) @ #x42385a>
> (funcall f2 20 5)
100
```

`apply` wendet ein Funktionsobjekt auf eine *Liste* von Argumenten an:

```
> (apply #'flaeche '(20 5))
100
> (apply #'flaeche 20 5 '())
100
> (apply f2 '(20 5))
100
> (apply #'+ 1 2 '(3 4))
10
```

Closures

Closures sind (unbenannte) Funktionen, die Variablen ihrer lexikalischen Umgebung referenzieren.

```
> (defun generator (s)
    #'(lambda (x) (setq x (+ x s))))
GENERATOR
> (setq f2 (generator 2))
#<Interpreted Closure (:INTERNAL GENERATOR) ...>
> (funcall f2 3)                 > (funcall f2 8)
5                                10

> (setq f7 (generator 7))
#<Interpreted Closure (:INTERNAL GENERATOR) ...>
> (funcall f7 3)                 > (funcall f7 8)
10                               15
```

Anwendung von Closures in `mapcar`

```
> (mapcar #'quadrat '(1 2 3 4))
(1 4 9 16)
> (defun elem-dazu (obj liste)
    (mapcar #'(lambda (elem)
                (list obj elem))
      liste))
> (elem-dazu 'anna '(apfel birne ananas))
((ANNA APFEL) (ANNA BIRNE) (ANNA ANANAS))
```

Logik

Logische Werte in Lisp:

- `NIL` bzw. `()` steht für *falsch*
- `T` steht für *wahr*
- alles andere gilt als *wahr*

Prädikate:

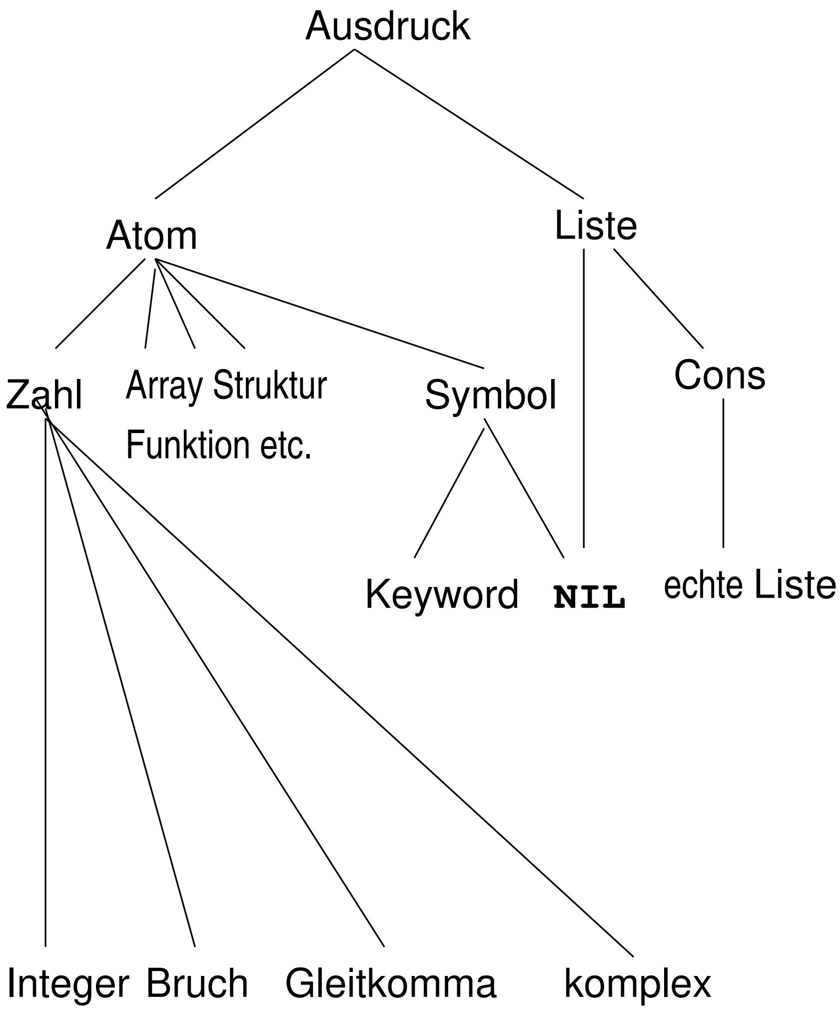
- Prädikate geben einen logischen Wert zurück
- Konvention: ein `p` oder `-p` am Ende des Namens.

Wichtige Gruppen von Tests:

- Zugehörigkeit zu Datentyp (1 Argument)
- Gleichheit (2 Arg.)
- Vergleiche (2 Arg.)
- Eigenschaft innerhalb des Datentyps (1 Arg.)

Außerdem: spezielle, auch selbstdefinierte Prädikate

Datentypen von Lisp



Zahlen, Zeichen, Strings, Symbole, Listen, Arrays, Hashtables, Readtables, Packages, Pathnames, Streams, Random-states, Strukturen, Funktionen, Conditions, Classes, Methods, Generic functions

- Lisp hat dynamische Typisierung
- manifeste Datentypen: Typtag an jedem Datenobjekt
- Überprüfung zur Laufzeit
- Typprädikate für jeden Typ
- `(type-of` ⟨*obj*⟩`)` gibt Typnamen zurück

Atome und Listen: `atom`, `listp`

```
> (setq farben '(rot gruen blau))
(ROT GRUEN BLAU)
```

```
> (atom 'farben)
T

> (atom 'rot)
T

> (atom farben)
NIL

> (atom
    '(rot gruen blau))
NIL

> (atom '())
T
```

```
> (listp 'farben)
NIL

> (listp 'rot)
NIL

> (listp farben)
T

> (listp
    '(rot gruen blau))
T

> (listp '())
T
```

- `listp` testet, ob das Argument eine Liste ist
- `atom` (nicht `atomp`!) testet, ob das Argument ein Atom ist

Zahlen und Symbole: `numberp`, `symbolp`

```
> (setq hoehe 1)
1
```

```
> (numberp 3.45)          > (symbolp 3.45)
T                         NIL

>(numberp hoehe)          > (symbolp hoehe)
T                         NIL

> (numberp 'hoehe)        > (symbolp 'hoehe)
NIL                       T

> (numberp farben)        > (symbolp farben)
NIL                       NIL
```

- `numberp` testet, ob das Argument eine Zahl ist
- `symbolp` testet, ob das Argument ein Symbol ist

`nil` – die leere Liste

```
> (eq nil '())              > ()
T                           NIL
> (equal nil '())           > '()
T                           NIL

> (atom nil)                > (atom '())
T                           T
> (symbolp nil)             > (symbolp '())
T                           T
> (listp nil)               > (consp '())
T                           NIL
```

- `nil` und `()` sind äquivalent.
- `nil` bzw. `()` ist gleichzeitig Symbol und Liste.
- `nil` evaluiert zu sich selbst.
- `nil` ist kein `cons`.
- `nil` steht für den Wahrheitswert *falsch*.

> *An atom in LISP and a list*
> *were dancing together a twist.*
> *But the list, getting ill*
> *bore hermaphrodite NIL,*
> *a monster, half atom, half list.*

`null` **und** `endp`

```
> (null '(rot gruen))    > (endp '(rot gruen))
NIL                      NIL

> (null ())              > (endp ())
T                        T

> (null 'farben)         > (endp 'farben)
NIL                      Error: illegal argument

> (null 3.45)            > (endp 3.45)
NIL                      Error: illegal argument
```

- `null` und `endp` testen beide auf leere Liste
- `endp` verlangt als Argument eine Liste
  $\Rightarrow$ testet, ob die Liste leer ist
- `null` erlaubt beliebige Datentypen als Argument
  $\Rightarrow$ testet, ob das Argument die leere Liste ist
- `(null` ⟨*elem*⟩`)` entspricht `(eq NIL` ⟨*elem*⟩`)`

Gleichheit

Alle sind gleich, aber manche sind gleicher.

Listen:

```
(setq alt '(1 2 3) neu '(1 2 3))
> (eq alt alt)              > (eq alt neu)
T                           NIL
> (equal alt neu)           > (equal alt '(1 2 3))
T                           T
```

Symbole:

```
(setq alt 'anna neu 'anna)
> (eq alt alt)              > (eq alt neu)
T                           T
```

Zahlen:

```
> (eql 4 4.0)               > (equal 4 4.0)
NIL                         NIL
> (= 4 4.0)                 > (equalp 4 4.0)
T                           T
```

- `eq` für Symbole, Listen in Spezialfällen. **(identisch)**
- `eql` für Zahlen gleichen Typs korrekt. **(identisch)**
- = für Zahlen. **(gleich)**
- `equal` für Listen, wenn keine Zahlen drin. **(gleich)**
- `equalp` für alle Datentypen. **(gleich)**

Vergleich von Zahlen

```
> (setq hoehe 1 breite 2 laenge 3)
3

> (> hoehe 2)
NIL

> (> 2 hoehe)
T

> (< hoehe breite)
T

> (> laenge breite hoehe)
T

> (> hoehe laenge breite)
NIL

> (< 1 2 3)
T
```

- <, <=, >, >=, =, /= verlangen Zahlen als Argumente.
- <, <=, >, >=, =, /= können *beliebig viele* Argumente haben.
- < und <= testen, ob die Argumente in aufsteigender, > und >= ob sie in absteigender Ordnung sind.

member

```
> (setq farben '(rot gruen blau))
(ROT GRUEN BLAU)

> (member 5 '(4 5 6 7))        > (member 3 '(4 5 6 7))
(5 6 7)                        NIL

> (member 'gruen farben)       > (member 'lila farben)
(GRUEN BLAU)                   NIL

> (member 'rot
     '((rot gruen blau) (rot gruen blau)))
NIL

> (member '(rot gruen blau)
     '((rot gruen blau) (blau schwarz)))
NIL

> (member '(rot gruen blau)
     '((rot gruen blau) (blau schwarz))
     :test #'equal)
((ROT GRUEN BLAU) (BLAU SCHWARZ))
```

- `member` testet (mit `eql`), ob ein Element in einer Liste ist
- `member` kann auch mit anderen über das Keyword `:test` angegebenen Prädikaten (z. B. `equal`) testen

Logische Prädikate `and`, `not` und `or`

- `NIL` bzw. `()` steht für falsch
- `T` und allgemein nicht-`NIL` gilt als wahr

```
> (not 2)
NIL
> (or (> 4 5) (< 3 6))
T
> (and (> 4 5) (< 3 6))
NIL
> (and 1 2 3)
3
> (or () 4 ())
4
```

- `and` und `or` sind Spezialformen.
- `or` wertet von links bis zum ersten *wahren* Ausdruck aus.
- `or` gibt den Wert dieses wahren Ausdrucks oder andernfalls `NIL` zurück.
- `and` wertet von links bis zum ersten *falschen* Ausdruck aus.
- `and` gibt `NIL` bzw. den Wert des letzten Ausdrucks zurück.
- Siehe auch `every` und `some`.

Verzweigung: cond

```
> (setq objekte '((quadrat 10) (rechteck 5 15)))
((QUADRAT 10) (RECHTECK 5 15))
> (defun flaeche (obj)
    (cond ((eq (first obj) 'rechteck)
           (* (second obj) (third obj)))
          ((eq (first obj) 'quadrat)
           (print "Oh ein Quadrat")
           (* (second obj) (second obj)))
          (t 'unbekannt)))
> (flaeche (first objekte))
"Oh ein Quadrat"
100
> (flaeche (second objekte))
75
> (flaeche '(dreieck 9)
UNBEKANNT
```

Syntax:

(cond $\{$(⟨*bedingung*⟩ ⟨*form*⟩*)$\}^*$)

- Eine `cond`-Anweisung besteht aus Zweigen, jeder Zweig besteht aus einer Bedingung und Anweisungen.
- Die Anweisungen des ersten Zweigs, dessen Bedingung erfüllt ist, werden ausgeführt.
- Der Wert der letzten Anweisung dieses Zweigs wird zurückgegeben.
- Bedingung `t` ist immer erfüllt.

Verzweigung: `case`

```
> (setq objekte '((quadrat 10) (rechteck 5 15)))
> (defun flaeche (obj)
    (case (first obj)
          (rechteck
           (* (second obj) (third obj)))
          (quadrat
           (print "Oh ein Quadrat")
           (* (second obj) (second obj)))
          (t 'unbekannt)))
> (flaeche (first objekte))
"Oh ein Quadrat"
100
> (flaeche (second objekte))
75
```

Syntax:

`(case` ⟨*keyform*⟩ {`(`⟨*keylist*⟩ ⟨*form*⟩*`)`}*`)`

- Schlüsselwert ⟨*keyform*⟩ wird ausgewertet.
- Zweig ist erfüllt, wenn Schlüsselwert mit einzelnem Schlüssel als ⟨*keylist*⟩ übereinstimmt (`eql`) oder in einer Schlüsselliste ⟨*keylist*⟩ enthalten ist (`member`).
- Ausdrücke des ersten erfüllten Zweigs werden ausgewertet und der Wert des letzen Ausdrucks zurückgegeben.
- Zweig mit Schlüssel `t` oder `otherwise` ist immer erfüllt.

Verzweigung: `if`, `when` und `unless`

if: `(if (>= x 0) (sqrt x) 'verboten)`

- Syntax: `(if` ⟨*bedingung*⟩ ⟨*form*⟩ [⟨*form*⟩]`)`
- Erste `form` ist *then*-Zweig.
- Zweite `form` ist *else*-Zweig
- Fehlt der *else*-Zweig und ist ⟨*bedingung*⟩ falsch, wird `NIL` zurückgegeben.

when: `(when (>= x 0) (sqrt x))`

- Syntax: `(when` ⟨*bedingung*⟩ ⟨*form*⟩*`)`
- Die Anweisungen werden ausgeführt, wenn die Bedingung erfüllt ist.
- `when` hat nur den *then*-Zweig eines `if`.

unless: `(unless (>= x 0) 'verboten)`

- Syntax: `(unless` ⟨*bedingung*⟩ ⟨*form*⟩*`)`
- Die Anweisungen werden ausgeführt, wenn die Bedingung *nicht* erfüllt ist.
- `unless` hat nur den *else*-Zweig eines `if`.

Iteration mit `do`

```
(defun summe-do (liste)
  (do ((l liste (rest l))
       (summe 0 (+ summe (first l))))
      ((null l) summe)))
```

Syntax:

(do ({(⟨*var*⟩ ⟨*init*⟩ ⟨*schritt*⟩)}*)
    (⟨*endtest*⟩ ⟨*ergebnis*⟩)
  ⟨*rumpf*⟩)

- `do` führt folgende Schritte zyklisch aus:
  1. `do` berechnet die Hilfsvariablen ⟨*var*⟩:
     Beim ersten Mal mit ⟨*init*⟩, später mit ⟨*schritt*⟩.
  2. Wenn ⟨*endtest*⟩ erfüllt,
     gibt `do` ⟨*ergebnis*⟩ zurück
  3. `do` führt den ⟨*rumpf*⟩ aus.
- `do` bindet parallel, wie `let`.
  Analog zu `let*` bindet `do*` sequentiell.
- Ablaufvorstellung: Tabelle
  Berechnungsvorschrift für die neue Tabellenzeile verwendet die Werte der letzten Tabellenzeile.

Ablauf der Iteration

Die Korrektheit der Schleife prüft man mit 3 Bedingungen:

- Die Vorbedingung beschreibt die Aufgabenstellung.
  Vorbedingung: `summe` $= 0$, `l` ganze Liste.
- Die Invariante gilt nach jedem Durchlauf des Rumpfs.
  Gesamtsumme = Summe über `l` + `summe`
- Nachbedingung und Invariante zusammen beschreiben das Ziel.
  Nachbedingung: Liste `l` ist leer.

Ablauftabelle für `(summe-do '(1 2 3 4 5))`

Variablen	`l`	`summe`
Init	`liste`	0
Schritt	`(rest l)`	`(+ summe (first l))`
Init	`(1 2 3 4 5)`	0
Zustand 1	`(2 3 4 5)`	1
Zustand 2	`(3 4 5)`	3
Zustand 3	`(4 5)`	6
Zustand 4	`(5)`	10
Zustand 5	`()`	15

In Zustand 5 ist die Nachbedingung `(null l)` erfüllt
Der Wert `summe` wird zurückgegeben.

Iteration über Listen mit `dolist`

```
> (defun drucke-elemente (liste)
    (dolist (elem liste :ende)
      (print elem)))
> (drucke-elemente '(anna eva petra))
ANNA
EVA
PETRA
:ENDE
```

Syntax:

`(dolist (`⟨*var*⟩ ⟨*liste*⟩ ⟨*erg*⟩`)` ⟨*rumpf*⟩`)`

- ⟨*liste*⟩ wird ausgewertet.
- ⟨*var*⟩ wird der Reihe nach an jedes Element von ⟨*liste*⟩ gebunden.
- Für jede Bindung wird ⟨*rumpf*⟩ ausgeführt.
- Abschließend wird ⟨*erg*⟩ ausgewertet und zurückgegeben.

Iteration, n-mal mit `dotimes`

```
> (defun zaehle-bis (n)
    (dotimes (i n :ende)
      (print i)))
> (zaehle-bis 3)
0
1
2
:ENDE
```

Syntax:

(dotimes ($\langle$*var*$\rangle$ $\langle$*anzahl*$\rangle$ $\langle$*erg*$\rangle$) $\langle$*rumpf*$\rangle$)

- $\langle$*anzahl*$\rangle$ wird ausgewertet.
- $\langle$*var*$\rangle$ wird der Reihe nach an jede ganze Zahl von 0 bis $\langle$*anzahl*$\rangle$ $- 1$ gebunden.
- Für jede Bindung wird $\langle$*rumpf*$\rangle$ ausgeführt.
- Abschließend wird $\langle$*erg*$\rangle$ ausgewertet und zurückgegeben.

„Schleife total“ mit `loop`

Schleifen nach den Konventionen von imperativen Sprachen schreibt man mit `loop`.

- Ausgangspunkt: unendliche Schleife
- Herausspringen: `(return)`
- Schleifenkonstrukte: `repeat`, `while`, `for`
- Vorher/Nachher: `initially`, `finally`
- Werte sammeln: `collect`
- Lokale Variablen: `with`

```
(defun zaehle-while (n)
  (loop initially (print :anfang)
        with i = 1
        while (<= i n) do
         (print i)
         (setq i (1+ i))
        finally (print :ende)))
```

```
(defun zaehle-for (n)
  (loop initially (print :anfang)
        for i from 1 to n do
        (print i)
        finally (print :ende)))
```

Beim Aufruf mit 5 drucken beide Funktionen:

```
:ANFANG 1 2 3 4 5 :ENDE
```

Teil 3

Lisp-Spezialitäten

- Listenverarbeitung: Destruktiv vs. Kopierend
- Parameter von Funktionen
- Lisp als Programmiersystem

Destruktiv vs. Kopierend

Bei zusammengesetzten Datentypen (z. B. Listen) beschreibt die Vorstellung von *Call-by-Value* die Argumentübergabe nicht ausreichend.

Kopierend: Argumente werden nicht geändert

- Umgang im Call-by-Value-Stil, Funktion ist nebenwirkungsfrei
- Ergebnis teilt sich Zellen mit Argumenten
- geänderte Teile neu
- Speicherverbrauch
- Funktionswert ist wichtig

Destruktiv: Zerstört Argumente (undefinierte Änderung)

- Wenn man Argumente danach garantiert nicht benötigt
- Kein Speicherverbrauch
- Funktionswert ist wichtig
  Änderung zu berücksichtigen

Mutierend Objekte werden (absichtlich) geändert

- Zuweisungsorientierter Stil mit expliziten Kopiervorgängen
- Funktion hat Nebenwirkung
- Kein Speicherverbrauch
- Funktionswert ist unerheblich

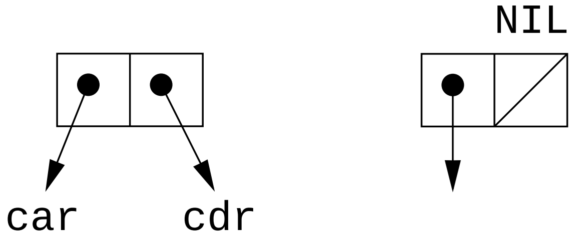
Listenrepräsentation

Adresse: Nummer einer Speicherzelle

Speicherzelle: kann Werte oder Adresssen von Speicherzellen enthalten

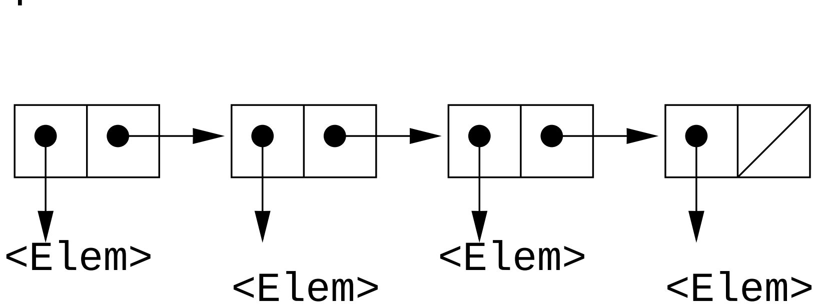
Zeiger: ist die in einer Speicherzelle abgelegte Adresse einer Speicherzelle

Zweizeigerzelle (Lisp-Konvention): Speicherzelle für zwei Zeiger, sogenanntes *dotted-pair*



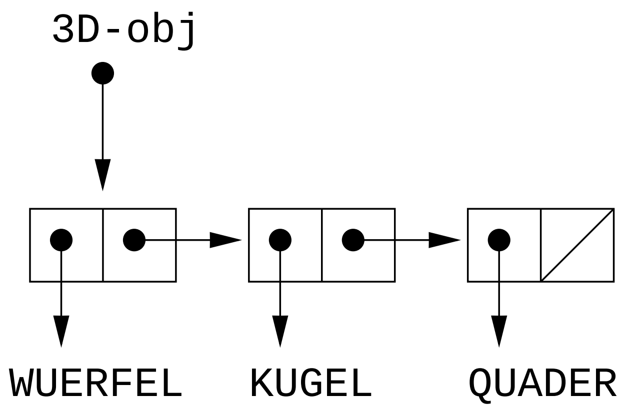
Liste: verkettete Mengen von Speicherzellen, wobei

- linke Zeiger (`car`) auf die einzelnen Listenelemente zeigen,
- rechte Zeiger (`cdr`) die Verbindung zur nächsten Speicherzelle herstellen.

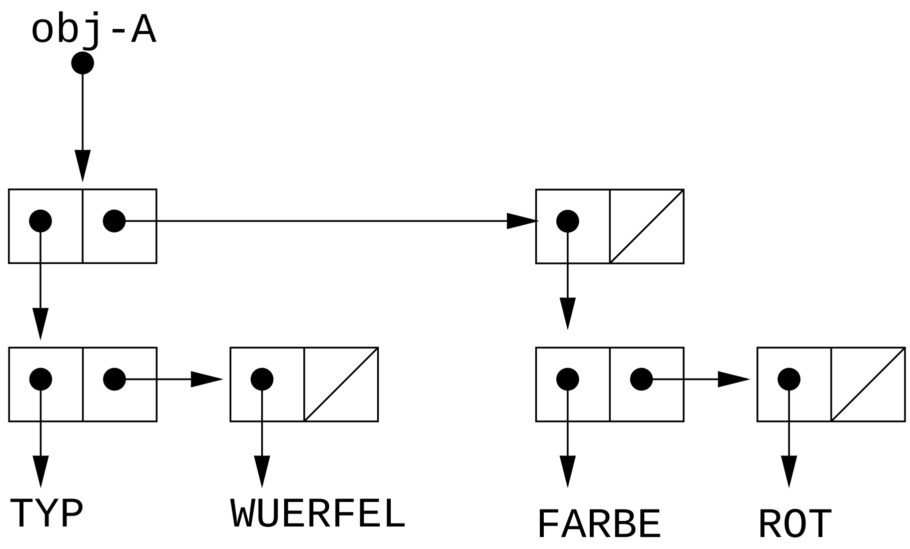


Listenrepräsentation

```
> (setq 3d-obj
        '(wuerfel kugel quader))
(WUERFEL KUGEL QUADER)
```

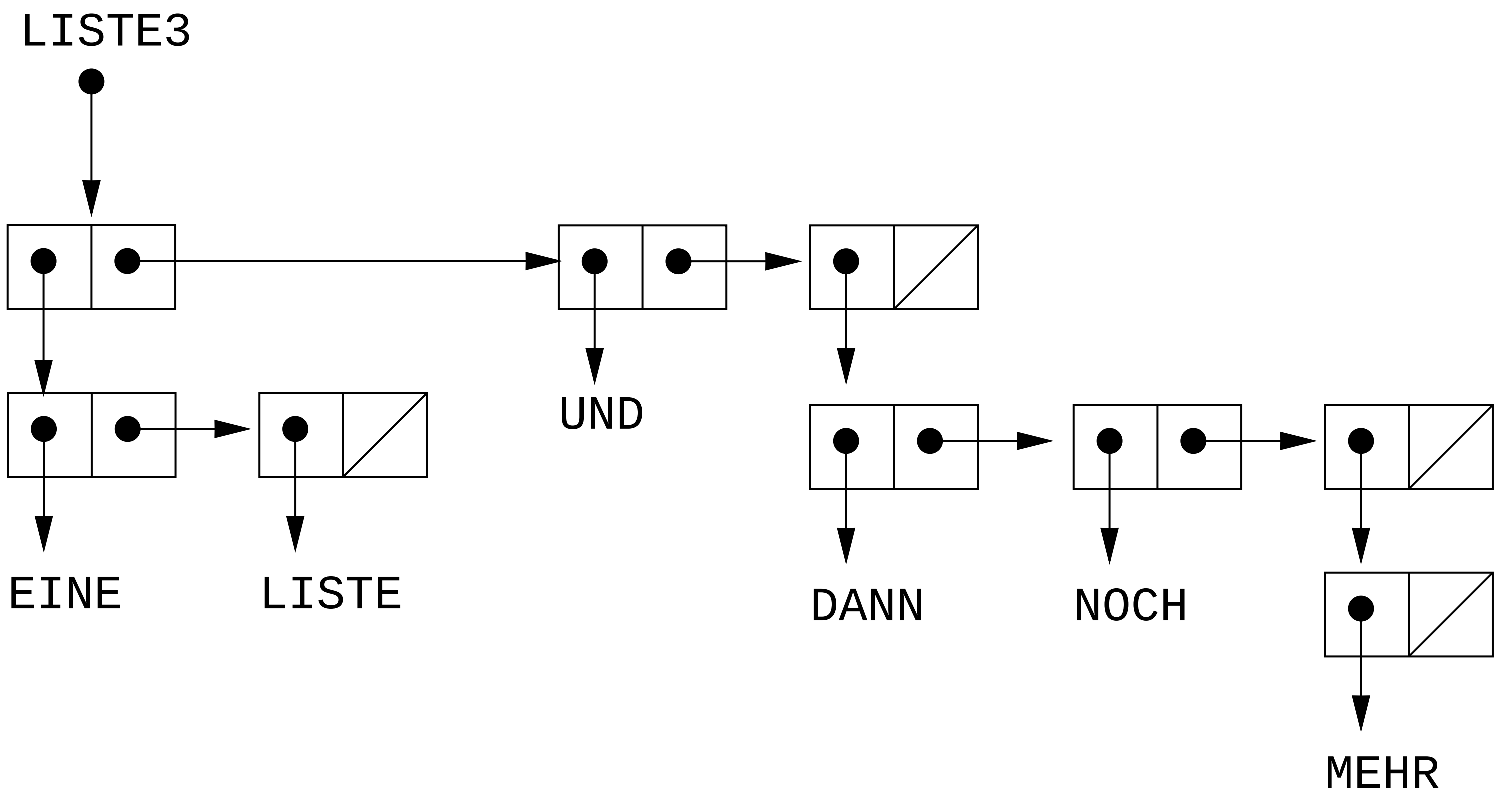



```
> (setq obj-A '((typ wuerfel) (farbe rot))
((TYP WUERFEL) (FARBE ROT))
```



Listenrepräsentation

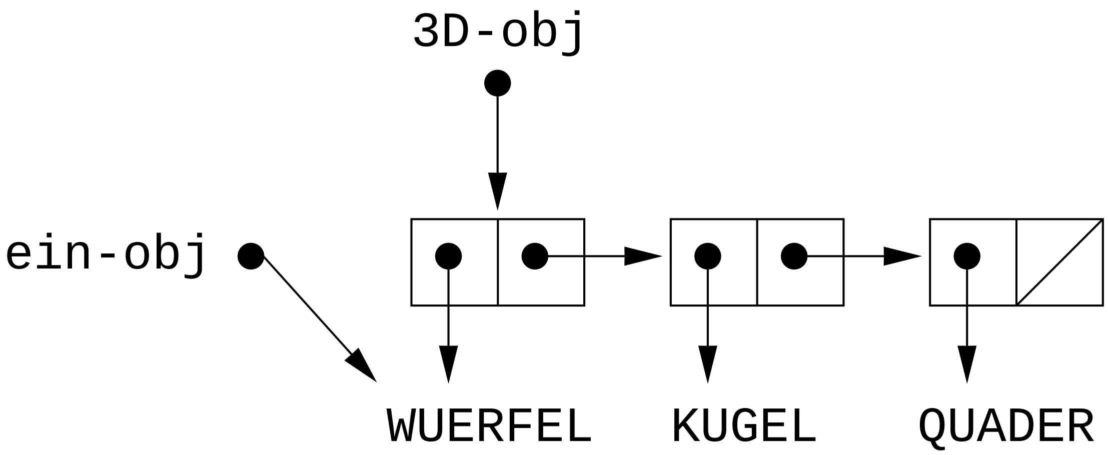
```
> (setq liste3 '((eine liste) und (dann noch (mehr))))
((EINE LISTE) UND (DANN NOCH (MEHR)))
```



Zugriff auf Listenzellen – `car` und `cdr`

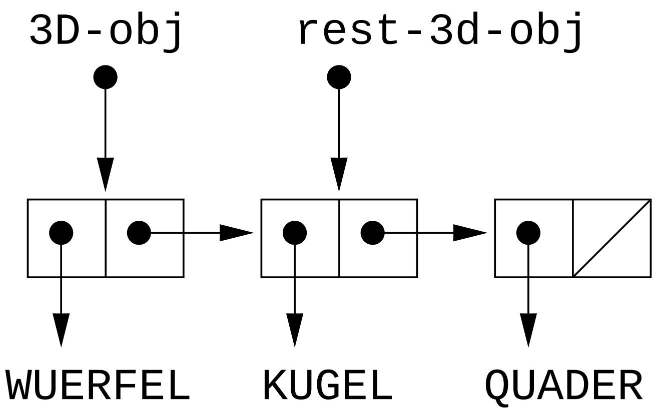
Erstes Element – `car` bzw. `first`

```
> (setq ein-obj (car 3d-obj))
WUERFEL
```



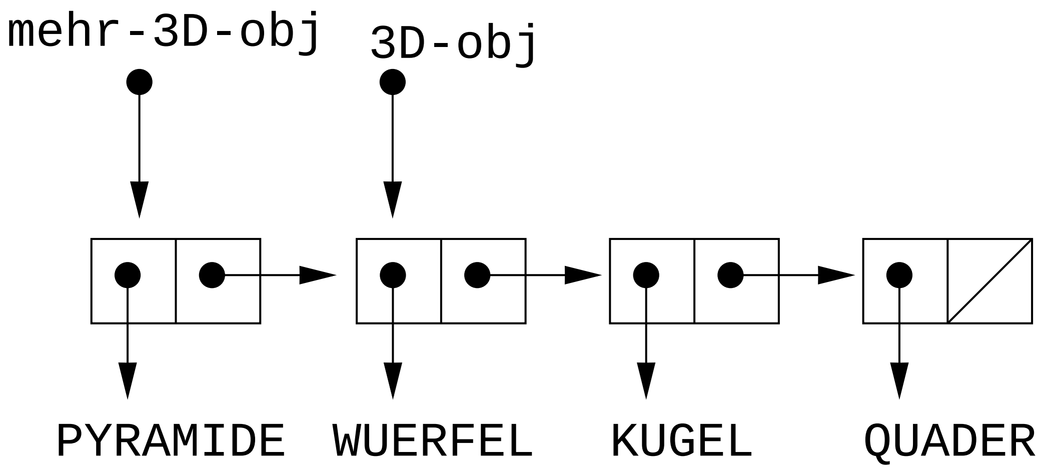
Rest der Liste – `cdr` bzw. `rest`

```
> (setq rest-3d-obj (cdr 3d-obj))
(KUGEL QUADER)
```

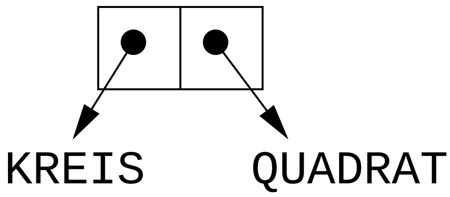


Listenzellen erzeugen – `cons`

```
> (setq mehr-3d-obj (cons 'pyramide 3d-obj))
(PYRAMIDE WUERFEL KUGEL QUADER)
```




```
> (cons 'kreis 'quadrat)
(KREIS . QUADRAT)
```




```
> (setq liste
        (cons 'wuerfel
              (cons 'kugel
                    (cons 'quader nil))))
(WUERFEL KUGEL QUADER)
```

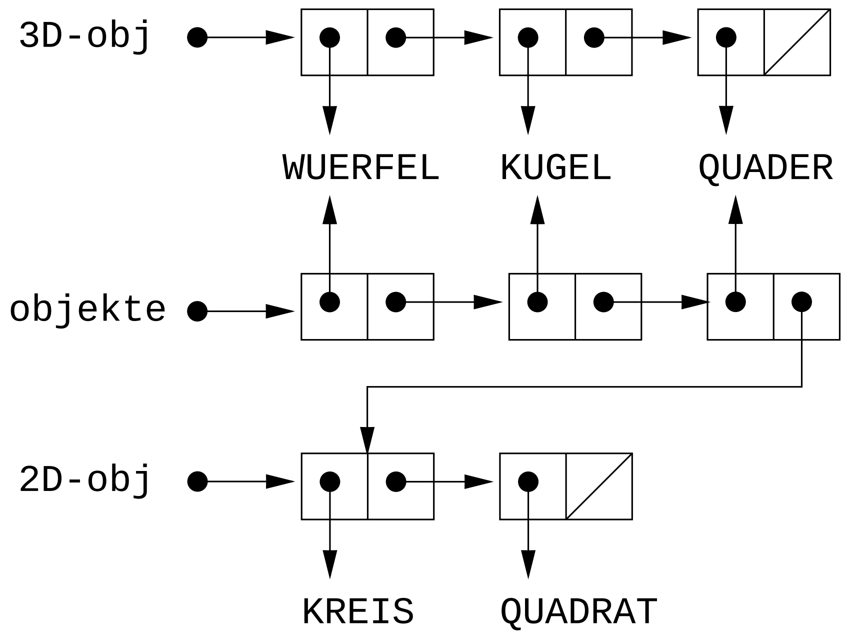
Kopierend vereinigen: `append`

```
> (setq 3d-obj '(wuerfel kugel quader))
(WUERFEL KUGEL QUADER)
> (setq 2d-obj '(kreis quadrat))
(KREIS QUADRAT)

> (setq objekte (append 3d-obj 2d-obj)
(WUERFEL KUGEL QUADER KREIS QUADRAT)

> 3d-obj
(WUERFEL KUGEL QUADER)
```

Kopierend, nicht destruktiv:



Keines der beiden Argumente wurde geändert,
aber `2d-obj` und `objekte` **teilen** sich Zellen.

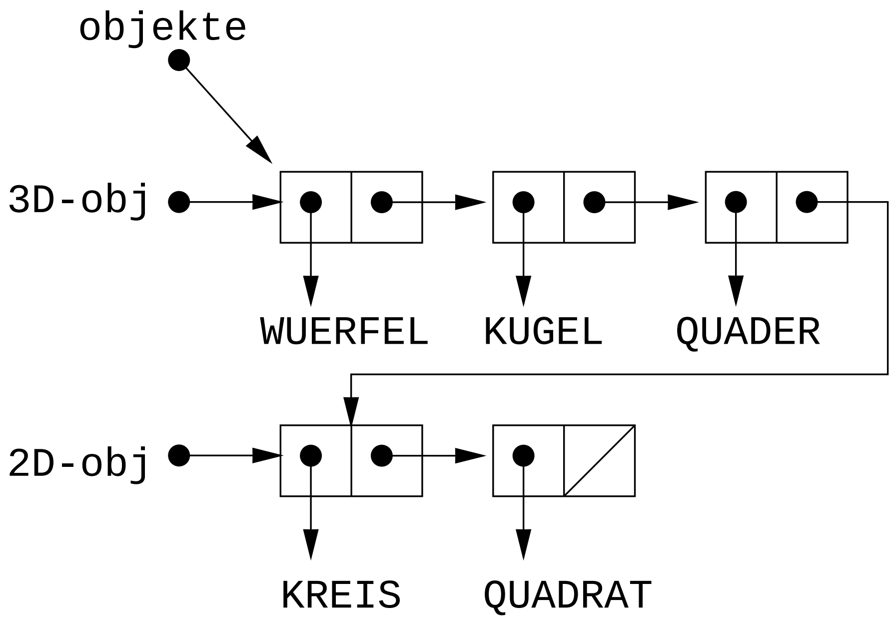
Destruktiv vereinigen: `nconc`

```
> (setq 3d-obj '(wuerfel kugel quader))
(WUERFEL KUGEL QUADER)
> (setq 2d-obj '(kreis quadrat))
(KREIS QUADRAT)

> (setq objekte (nconc 3d-obj 2d-obj))
(WUERFEL KUGEL QUADER KREIS QUADRAT)

> 3d-obj
(WUERFEL KUGEL QUADER KREIS QUADRAT)
```

Destruktiv:



Die erste Liste `3d-obj` wurde geändert
und `2d-obj` und `objekte` **teilen** sich Zellen.

Vergleich von `nconc` und `append`

```
>(setq 3d-obj '(wuerfel kugel quader))
>(setq 2d-obj '(kreis quadrat))
>(setq rest-3d-obj (cdr 3d-obj))
```

append:

```
> (setq objekte
        (append 3d-obj
                2d-obj))
(WUERFEL KUGEL QUADER
 KREIS QUADRAT)

> 3d-obj
(WUERFEL KUGEL QUADER)

> 2d-obj
(KREIS QUADRAT)

> rest-3d-obj
(KUGEL QUADER)
```

nconc:

```
> (setq objekte
        (nconc 3d-obj
               2d-obj))
(WUERFEL KUGEL QUADER
 KREIS QUADRAT)

> 3d-obj
(WUERFEL KUGEL QUADER
 KREIS QUADRAT)

> 2d-obj
(KREIS QUADRAT)

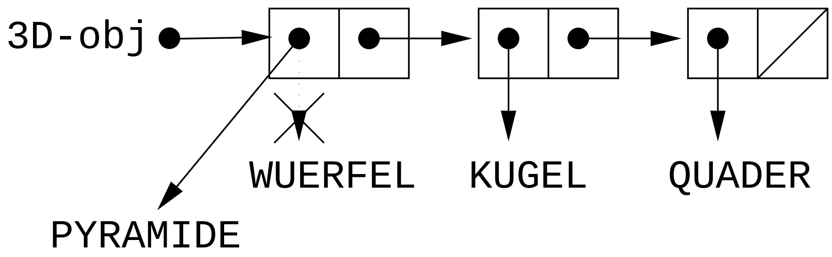
> rest-3d-obj
(KUGEL QUADER KREIS
 QUADRAT)
```

- `nconc` und `append` hängen zwei Listen aneinander.
- `nconc` ändert alle bis auf die letzte der Listen.
- `append` läßt seine Argumente unverändert.

Mutierende Veränderung mit `setf`

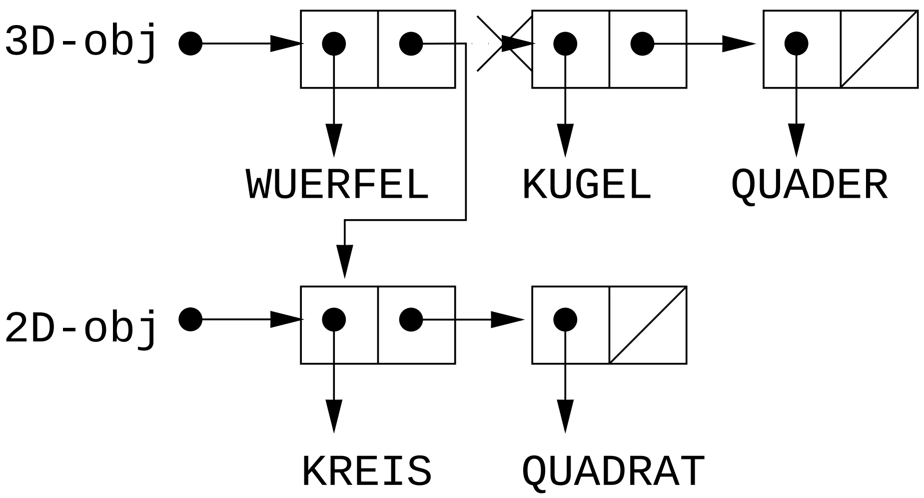
setf car

```
> (setq 3d-obj '(wuerfel kugel quader))
> (setf (car 3d-obj) 'pyramide)
> 3d-obj
(PYRAMIDE KUGEL QUADER)
```



setf cdr

```
> (setq 3d-obj '(wuerfel kugel quader))
> (setq 2d-obj '(kreis quadrat))
> (setf (cdr 3d-obj) 2d-obj)
> 3d-obj
(WUERFEL KREIS QUADRAT)
```

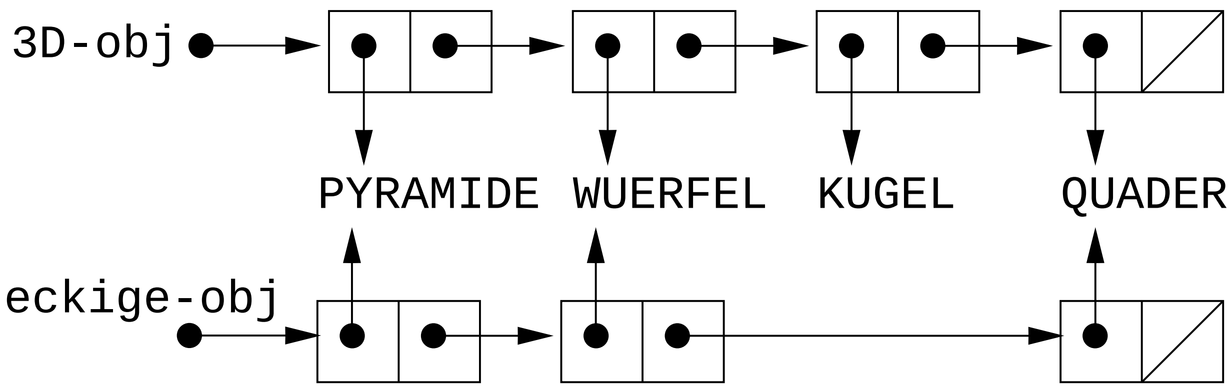


Elementе entfernen mit `remove`

```
> (setq 3d-obj '(pyramide wuerfel kugel quader))
(PYRAMIDE WUERFEL KUGEL QUADER)

> (setq eckige-obj (remove 'kugel 3d-obj))
(PYRAMIDE WUERFEL QUADER)

> 3d-obj
(PYRAMIDE WUERFEL KUGEL QUADER)
```



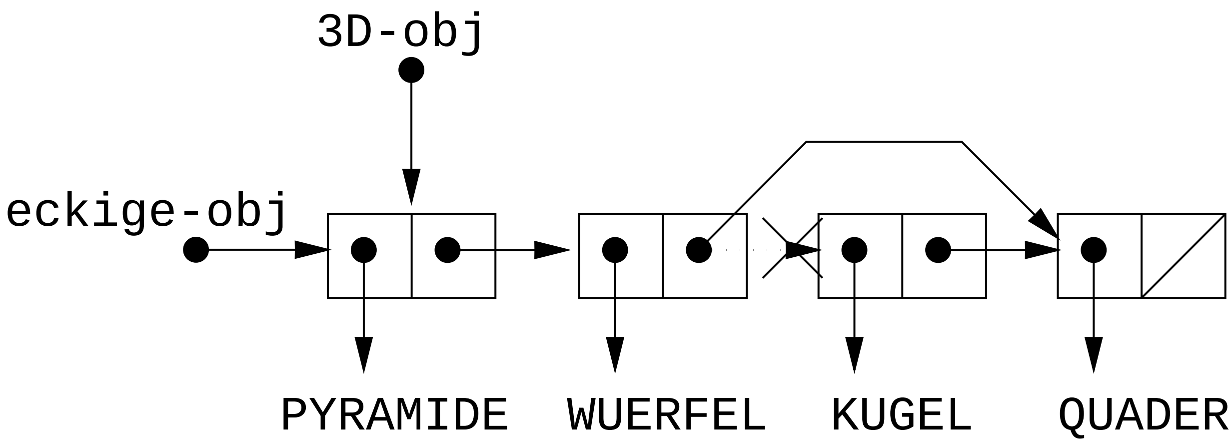
- `remove` löscht alle Vorkommen eines Elements aus einer Liste.
- `remove` läßt die ursprüngliche Liste unverändert.

Elemente entfernen mit `delete`

```
> (setq 3d-obj '(pyramide wuerfel kugel quader))
(PYRAMIDE WUERFEL KUGEL QUADER)

> (setq eckige-obj (delete 'kugel 3d-obj))
(PYRAMIDE WUERFEL QUADER)

> 3d-obj
(PYRAMIDE WUERFEL QUADER)
```



- `delete` löscht alle Vorkommen eines Elements aus einer Liste.
- `delete` verändert (zerstört) die ursprüngliche Liste.
- Beim Löschen des ersten Elements:
  Funktionswert von `delete` korrekt,
  jedoch *keine* Änderung am Argument.
  ⇒ Immer Funktionswert weiterverwenden!

Destruktive vs. kopierende Funktionen

- `cons` erzeugt neue Listen durch Anlegen neuer Zeiger in freien Cons-Zellen.
- `cons` erzeugt ein Punktpaar (*dotted-pair*), wenn das zweite Argument ein Atom ($\neq$ `NIL`) ist.
- `list` erzeugt eine neue Liste mit den Argumenten als Elemente.
- `append` und `remove` erzeugen zum Kopieren und Löschen eine neue Listenstruktur und kopieren Zeiger.
- `nconc`, `delete`, `setf car` und `setf cdr` verändern die Listen destruktiv, d. h. sie legen keine Kopien an, sondern verändern die Zeiger der ursprünglichen Liste.
- Hinweis: auch bei nicht-destruktiven Funktionen können Ergebnis und Argumente gemeinsame Zellen haben.

Listendurchwanderung

Bei der Durchwanderung (*mapping*) geht man die Liste durch und wendet eine Funktion an.

- Durchwanderungsfunktionen sind Funktionen höherer Ordnung.
- Die Funktion wird entweder auf das aktuelle Element (`car`) oder auf die gesamte Liste angewandt.
- Das Gesamtergebnis der Durchwanderung wird
  - nicht akkumuliert und nicht zurückgegeben, wenn nur die Nebenwirkungen wichtig sind,
  - als Liste der Einzelergebnisse gesammelt zurückgegeben,
  - als destruktive Vereinigung (mit `nconc`) der Einzelergebnisse zurückgegeben.

	Ergebnis		
arbeitet auf	kein	sammeln	vereinigen
Einzelelement	`mapc`	`mapcar`	`mapcan`
aktuelle Liste	`mapl`	`maplist`	`mapcon`

- `mapc` und `mapl` geben die Liste unverändert zurück.
- `mapcan` und `mapcon` verändern die Ergebnisse der Funktion destruktiv.

Beispiele zur Durchwanderung

```
> (mapc #'print '(1 2 3))
1
2
3
(1 2 3)

> (mapl #'print '(1 2 3))
(1 2 3)
(2 3)
(3)
(1 2 3)

> (mapcar #'quadrat '(1 2 3 4))
(1 4 9 16)

> (maplist #'identity '(1 2 3 4))
((1 2 3 4) (2 3 4) (3 4) (4))
> (maplist #'length '(1 2 3 4))
(4 3 2 1)

> (mapcan #'(lambda (el) (list el el)) '(1 2 3 4))
(1 1 2 2 3 3 4 4)

> (mapcon #'(lambda (el) (list el el)) '(1 2 3))
((1 2 3) (1 2 3) (2 3) (2 3) (3) (3))
```

Parameter von Funktionen

(Fast) alle vordefinierten Konstruktionen kann man selber nachschreiben.

```
(defun quadrat (x)
  (* x x))
> (quadrat 10)
100
> (quadrat)
Error: Quadrat got 0 args, wanted 1 arg
> (quadrat 10 20)
Error: Quadrat got 2 args, wanted 1 arg
```

Beispiel: + arbeitet mit beliebiger Argumentzahl

```
>(+ 1 2)
3
>(+)   >(+ 1)   >(+ 1 2)    >(+ 1 2 3)   >(+ 1 2 3 4)
0      1        3           6            10
```

Syntax der Lambda-Liste einer Funktion:

(defun ⟨*name*⟩
  (⟨*var*⟩*
    [&optional {⟨*var*⟩ | (⟨*var*⟩ ⟨*default*⟩ [⟨*svar*⟩])}*]
    [&rest ⟨*var*⟩]
    [&key {⟨*var*⟩ | (⟨*var*⟩ ⟨*default*⟩ [⟨*svar*⟩])}*])
  ⟨*form*⟩*)

Optionale Parameter

Optionale Parameter müssen beim Aufruf nicht angegeben werden:

```
(defun gib (x &optional y)
  (list x y))
> (gib 'anna)          > (gib 'anna 'apfel)
(anna NIL)             (anna apfel)
```

Angabe eines Defaultwerts für optionale Parameter:

```
(defun gib (x &optional (y 'birne))
  (list x y))
> (gib 'anna)          > (gib 'anna 'apfel)
(anna birne)           (anna apfel)
```

Zuvorderst stehende Argumentwerte sind verwendbar:

```
(defun gib (x &optional (y x))
  (list x y))
> (gib 'anna)          > (gib 'anna 'apfel)
(anna anna)            (anna apfel)
```

Abfrage, ob optionaler Parameter angegeben:

```
(defun gib (x &optional (y 'birne gesetztp))
  (list x y gesetztp))
> (gib 'anna)          > (gib 'anna 'birne)
(anna birne NIL)       (anna birne T)
```

Schlüsselwortparameter

```
(defun gib (x &key frucht auto)
  (list x frucht auto))


> (gib 'anna)              > (gib 'anna :auto 'bmw
(anna NIL NIL)                          :frucht 'apfel)
                           (anna apfel bmw)
```

Angabe eines Defaultwerts:

```
(defun gib (x &key frucht (auto 'vw))
  (list x frucht auto))


> (gib 'anna)              > (gib 'anna :frucht 'apfel)
(anna NIL vw)              (anna apfel vw)
> (gib 'anna :auto 'bmw)
(anna NIL bmw)
```

Vorsicht bei Kombination von optionalen und Schlüsselwort-Parametern:

```
(defun gib (&optional x auto &key frucht)
  (list x auto frucht))
> (gib)                     > (gib 'anna 'bmw)
(NIL NIL NIL)               (anna bmw NIL)
> (gib 'anna 'bmw :frucht 'apfel)
(anna bmw apfel)
> (gib :frucht 'apfel)
(:frucht apfel NIL)
```

Restparameter

Der Restparameter sammelt alle verbliebenen Argumente in einer Liste auf.

```
(defun gib (x &rest r)
  (list x r))

> (gib anna)
(anna NIL)
> (gib anna apfel)    > (gib anna apfel birne)
(anna (apfel))      (anna (apfel birne))
```

Optionale Parameter werden vor dem Restparameter belegt:

```
(defun gib (x &optional auto &rest obst)
  (list x auto obst))

> (gib 'anna)           > (gib 'anna 'bmw)
(anna NIL NIL)         (anna bmw NIL)
> (gib 'anna 'bmw 'apfel 'birne 'banane)
(anna bmw (apfel birne banane))
> (gib 'anna 'apfel 'birne 'banane)
(anna apfel (birne banane))
```

Programmiersystem

Lisp ist nicht nur eine Programmiersprache,
sondern ein Programmier*system*.

- Laufzeitspeicherverwaltung
- Dynamische (Laufzeit-)Definition von Funktionen
  $\Longrightarrow$ Erzeugen und Laden von Programmen
- Kompilation
- Standardisierte Ein- und Ausgabe für Daten
- Fehler- (bzw. Ausnahmen-)Behandlung zur Laufzeit
- Debugging

Garbage collection (Speicherbereinigung)

```
> (setq liste '(1 2 3 4))
(1 2 3 4)
> (setq liste '(anna eva petra))
(ANNA EVA PETRA)
```

Woher kommen neue Zellen ?

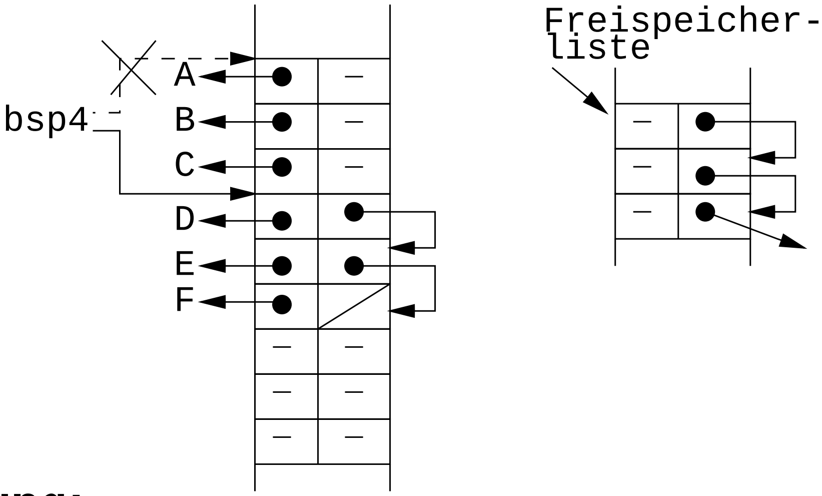
Wohin gehen ausgediente Zellen ?

- Die Speicherverwaltung reserviert Platz für Objekte.
- Objekte, auf die kein Verweis (Zeiger) mehr existiert, werden „Müll“ (*garbage*).
- Lisp führt gelegentlich eine *Garbage-Collection* durch.
- Die momentan abgearbeitete Funktion wird zur *Garbage-Collection* unterbrochen.
- Vorteile gegenüber manueller Verwaltung:
  - Beliebige Objekte können wie einfache Werte behandelt werden.
  - Effizienz, besonders bei hohem Anteil von *Garbage*.
- Nachteil: Unterbrechung kann zu einem ungünstigen Zeitpunkt notwendig werden.

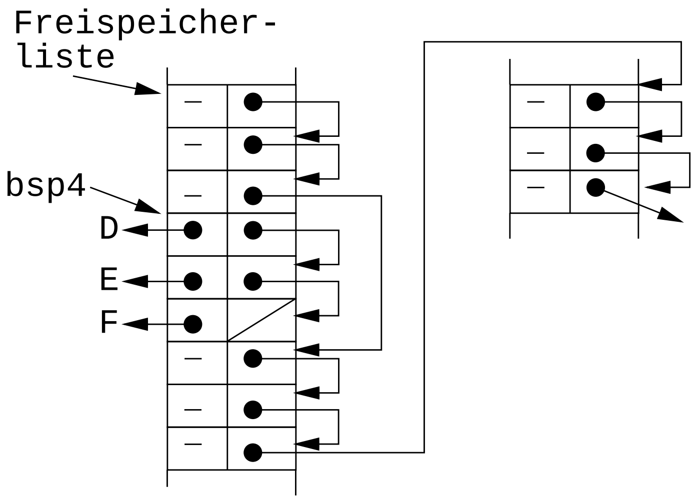
Mark and Sweep Garbage Collection

```
> (setq bsp4 (list 'a 'b 'c))
(A B C)
> (setq bsp4 (list 'd 'e 'f))
(D E F)
```

Markierung:



Bereinigung:



- **Markierungs**phase: kennzeichne Speicherzellen, die aktuell in Bearbeitung sind
- **Bereinigungs**phase: erkenne alle nicht markierten Zellen und füge sie zur Freispeicherliste hinzu

Fehlerbehandlung

Fehlerbegriff des CL-Standards:

- *"an error is signalled"*
  Fehlermeldung wird gegeben
- *"it is an error"*
  Aufruf unzulässig, Verhalten undefiniert

Fehlerarten:

1. Stillschweigendes Fehlverhalten –
   Falscher Rückgabewert
2. Abbruch der Ausführung mit Fehlermeldung
3. „Aufhängen“ des Programms
   Gründe:
   (a) unendliche Schleife
   (b) ewiges Warten auf externes Ereignis
       (z. B. Peripherie, Benutzer)
4. Absturz des Lispsystems

Eval

In Lisp wird Code durch Listen dargestellt, die der Evaluator auswertet. `eval` ruft den Evaluator explizit auf.

```
> (setq x 15)
15
> (setq y 'x)
X
> y
X
> (eval y)
15
> (setq code '((* 3 4) (+ 3 4)))
((* 3 4) (+ 3 4))
> (eval (first code))
12
> (eval (second code))
7
```

Achtung: Wird `eval` innerhalb einer Funktion gerufen, dann wertet es den Code trotzdem in der **globalen** Umgebung aus.

- `eval` erschwert die Analyse von Programmen und verringert die Effizienz.
- Der Einsatz von `eval` ist nur gerechtfertigt, wenn zur Laufzeit eingegebene Ausdrücke ausgewertet werden müssen und `eval` dabei einen komplizierten Interpreter ersetzt.

Leistung einer Programmiersprache

1. Gewöhnlicher Code
   - Funktions-, Variablensymbole und Konstanten
   - statisch in ein Maschinenprogramm übersetzbar
   - in prozeduralen Programmiersprachen möglich.
   - Einsatz unproblematisch
2. Funktionen als Datentyp
   - benannte Funktionen oder Funktionskonstanten
   - statische Realisierung durch Funktionszeiger
   - spezielle Programmiertechniken
3. Compiler/Interpreter zur Laufzeit
   - Funktionen zur Laufzeit definieren
Verwendungszweck: Programmierwerkzeuge
   - Unbenannte Funktionen definieren:
wenn Anfrage erst zur Laufzeit bekannt ist
4. Eval
   - Code wird ausgewertet
   - Es entstehen zusätzlich zu 3. noch Probleme mit den Gültigkeitsbereichen (*scope*) von Variablen.
   - Hilfmittel zum Bau von Interpretern

Teil 4

Datentypen

- Liste
- Character und String
- Sequence und Array
- Symbol-Property
- Hashtabelle
- Struktur

Übersicht über die Lisp-Datentypen

- Zahlen
- Characters
- Strings
- Symbole
- Listen
- Arrays
- Hashtables
- Readtables
- Packages
- Pathnames
- Streams
- Random-states
- Structures
- Functions
- Conditions
- Classes
- Methods
- Generic functions

Zahlen

LISP kennt unterschiedliche Datentypen für Zahlen:

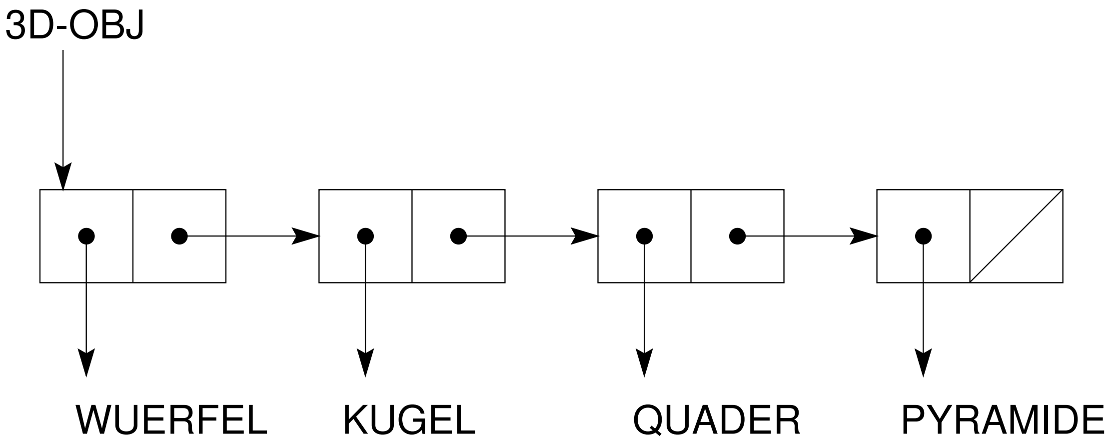
- `number` – `real` und `complex` (Bsp.: `#C(0 1)` $= i$)
- `real` – `float` und `rational`
- `float` – unterschiedlicher Präzision: short, single (Bsp.: `2.1f3`), double, long
- `rational` – `integer` und `ratio` (Bsp.: `1/3`)
- `integer` – `fixnum` (kleine Integer) und `bignum` (große bis sehr große Integer)

Beispiel für Operationen auf Zahlen:

```
> (numberp 123)
T
> (+ 2/3 (sqrt 4) (floor (expt 2.1 3)) (min 1 (abs -2) 3))
12.666667
```

Listen

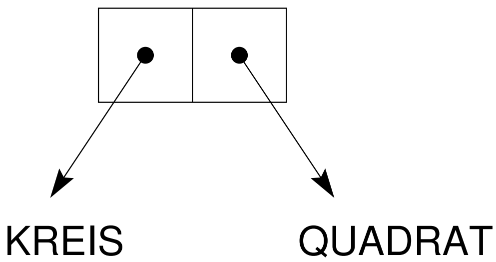
```
> (setq 3D-obj
    '(wuerfel kugel quader pyramide))
(WUERFEL KUGEL QUADER PYRAMIDE)
```



- eine echte Liste ist eine leere Liste
  oder eine *Cons*-Zelle, deren `cdr` wieder eine Liste ist
- `cdr` des letzten Elements zeigt auf `NIL`

Dotted-Pair:

```
> (cons 'kreis 'quadrat)
(KREIS . QUADRAT)
```



- *dotted pair*: Cons-Zelle, deren `cdr` nicht auf Liste zeigt.

Listen (Forts.)

Test auf Liste/Cons-Zelle:

```
> (listp '(a b c))
T
> (listp ())
T
> (listp '(a . b))
T
> (listp 'a)
NIL
```

```
> (consp '(a b c))
T
> (consp ())
NIL
> (consp '(a . b))
T
> (consp 'a)
NIL
```

Test auf leere Liste/das Ende einer Liste:

```
> (null '(a b c))
NIL
> (null '())
T
> (null 'a)
NIL
```

```
> (endp '(a b c))
NIL
> (endp '())
T
> (endp 'a)
Error: Illegal argument
       to endp: A
> (endp (rest '(a)))
T
```

Listen (Forts.)

```
> (a . (b c))
(A B C)

> (cons (first '(a b c)) (rest '(a b c)))
(A B C)

> (list* 'a 'b '(c))
(A B C)


> (append '(fortran lisp) '(algol pascal))
(FORTRAN LISP ALGOL PASCAL)

> (push 'fortran '(lisp algol pascal))
(FORTRAN LISP ALGOL PASCAL)

> (list-length '(a b c))
3

> (nth 1 '(fortran lisp algol pascal))
LISP

> (last '(fortran lisp algol pascal))
(PASCAL)

> (butlast '(fortran lisp algol pascal))
(FORTRAN LISP ALGOL)

> (member 'lisp '(fortran lisp algol pascal))
(LISP ALGOL PASCAL)
```

Assoziationslisten/A-Listen

```
> (setq obj-a
    '((name a)(typ quadrat)(farbe rot)(laenge 5)))
((NAME A) (TYP QUADRAT) (FARBE ROT) (LAENGE 5))

> (assoc 'typ obj-a)
(TYP QUADRAT)

> (second (assoc 'typ obj-a))
QUADRAT
```

- Assoziationsliste besteht aus gewöhnlicher Liste, deren Elemente als *Attribut-Werte-Paare* interpretiert werden
- Funktion `assoc` gibt zu einem *Attribut* (key) eine Liste aus *Attribut* und *Wert* (value) zurück

Alternativen zum Aufbau von A-Listen:

```
> (setq obj-b
     (acons 'name 'b
            '((typ . kreis) (farbe . weiss))))
((NAME . B) (TYP . KREIS) (FARBE . WEISS))

> (setq obj-b
     (pairlis '(name typ) '(b kreis)
              '((farbe . weiss))))
((TYP . KREIS) (NAME . B) (FARBE . WEISS))
```

Assoziationslisten/A-Listen (Forts.)

```
> (defun make-obj (name typ farbe laenge)
     (pairlis '(name typ farbe laenge)
              (list name typ farbe laenge)))
MAKE-OBJ

> (defun get-farbe (obj)
     (cdr (assoc 'farbe obj)))
GET-FARBE


> (setq objects
        (list (make-obj 'a 'wuerfel 'rot 5)
              (make-obj 'b 'wuerfel 'gelb 7)
              (make-obj 'c 'wuerfel 'blau 8)))
(((NAME . A) (TYP . WUERFEL) (FARBE . ROT) (LAENGE . 5))
 ((NAME . B) (TYP . WUERFEL) (FARBE . GELB) (LAENGE . 7))
 ((NAME . C) (TYP . WUERFEL) (FARBE . BLAU) (LAENGE . 8)))


> (get-farbe (third objects)) ;; von Wuerfel C
BLAU

> (get-farbe obj-b) ;; siehe vorherige Folie(n)
WEISS
```

Implementierung von `assoc`

```
(defun assoc (key list &key (test #'eql))
  (if (null list)
      'nil
    (if (funcall test key (car (first list)))
        (first list)
      (assoc key (rest list) :test test))))
```

Verwendung:

```
> (assoc 'b
         '((a 1) (b 2) (c 3)))
(B 2)

> (assoc '(b b)
         '(((a a) 1) ((b b) 2) ((c c) 3)))
NIL

> (assoc '(b b)
         '(((a a) 1) ((b b) 2) ((c c) 3))
         :test #'equal)
((B B) 2)
```

Zeichen/Characters

Eingabenotation:

```
> #\A                      > #\Space
#\A                        #\space
```

Prädikate:

```
> (characterp #\a)
T

> (characterp 'a)
NIL

> (alpha-char-p #\a)
T

> (alpha-char-p #\Space)
NIL
```

Vergleichsprädikate:

```
> (char= #\a #\A)
NIL

> (char= #\a #\a)
T

> (char< #\a #\b)
T
```

Zeichen/Characters (Forts.)

Prädikate für bestimmte Eigenschaften:

```
> (upper-case-p #\B)
T

> (down-case-p #\B)
NIL

> (both-case-p #\B)
T

> (both-case-p #\.)
NIL
```

Umwandlungen:

```
> (char-upcase #\c)
#\C

> (char-downcase #\Z)
#\z
```

```
> (character 'z)
#\Z
> (character 'space)
NIL
```

```
> (char-code #\Z)
90
> (code-char 89)
#\Y
```

Zeichenketten/Strings

Prädikate:

```
> (stringp "Lisp-Kurs")
T

> (stringp 'lisp)
NIL

> (arrayp "Lisp-Kurs")
T
```

- *Strings* sind Vektoren (eindimensionale Arrays) aus *Characters*.

Vergleichsprädikate:

```
> (string= "Lisp-Kurs" "Lisp-Kurs")
T

> (string= "Lisp-Kurs" "lisp-kurs")
NIL

> (string< "alles" "aller")
NIL

> (string> "alles" "aller")
4
```

Zeichenketten/Strings (Forts.)

Selektion einzelner Zeichen:

```
> (char "Lisp-Kurs" 4)
#\-
```

Konstruktion eines Strings:

```
> (make-string 5 :initial-element #\!)
"!!!!!"
```

Umwandlungen:

```
> (string "alles klar")
"alles klar"

> (string 'obj-a)
"OBJ-A"

> (string 123)
Error: Cannot convert 123 into a string

> (princ-to-string 123)
"123"

> (string-capitalize "lisp-kurs ist toll")
"Lisp-Kurs Ist Toll"

> (string-upcase "lisp-kurs ist toll")
"LISP-KURS IST TOLL"
```

Aufzählungen/Sequences

- Listen und Vektoren (Arrays, Strings) sind *Sequences*.
- *Sequences* bestehen aus einer Menge geordneter Elemente; erstes Element hat Index 0.
- Auf Sequences sind *Operationen* definiert, die dadurch sowohl auf Listen als auch auf Vektoren definiert sind:

```
elt        reverse       map        count
length     nreverse      some       sort
subseq     concatenate   every      merge
find       position      notany     mismatch
fill       search        notevery   substitute
replace    copy-seq      reduce     nsubstitute
remove                   delete
remove-duplicates        delete-duplicates
```

Beispiele:

```
> (position 'kreis '(quadrat kreis dreieck))
1

> (position '#\a "quadrat")
2

> (position '#\a "kreis")
NIL
```

Aufzählungen/Sequences (Forts.)

```
> (find 'kreis '(quadrat kreis dreieck))
KREIS

> (find '#\a "quadrat")
#\a

> (find '#\a "kreis")
NIL

> (find '(kreis) '(quadrat (kreis) dreieck))
NIL

> (find '(kreis) '(quadrat (kreis) dreieck)
        :test #'equal)
(KREIS)


> (concatenate 'string "Lisp" "-" "Kurs")
"Lisp-Kurs"

> (concatenate 'list "Lisp" "-" "Kurs")
(#\L #\i #\s #\p #\- #\K #\u #\r #\s)

> (concatenate 'vector "Lisp" "-" "Kurs")
#(#\L #\i #\s #\p #\- #\K #\u #\r #\s)
```

Aufzählungen/Sequences (Forts.)

```
> (reverse '(quadrat kreis dreieck))
(DREIECK KREIS QUADRAT)

> (reverse "Kreis")
"sierK"


> (subseq "Lisp-Kurs" 0 4)
"Lisp"

> (subseq '(quadrat kreis dreieck) 4 5)
NIL


> (fill "Lisp-Kurs" #\* :start 5 :end 8)
"Lisp-***s"

> (substitute #\A #\a "Banane")
"BAnAne"

> (remove #\a "Banane")
"Bnne"
```

Aufzählungen/Sequences (Forts.)

`remove`, `count`, `find` **mit Prädikatargument:**

```
(setq objects
  '(((NAME . A) (TYP . WUERFEL) (FARBE . ROT) (LAENGE . 5))
    ((NAME . B) (TYP . WUERFEL) (FARBE . GELB) (LAENGE . 7))
    ((NAME . C) (TYP . WUERFEL) (FARBE . BLAU) (LAENGE . 8))))
(defun gelb-p (object)
  (equal (get-farbe object) 'gelb))
```

```
> (remove-if #'gelb-p objects)
```

```
(((NAME . A) (TYP . WUERFEL) (FARBE . ROT) (LAENGE . 5))
 ((NAME . C) (TYP . WUERFEL) (FARBE . BLAU) (LAENGE . 8)))
```

```
> (remove-if-not #'gelb-p objects)
```

```
(((NAME . B) (TYP . WUERFEL) (FARBE . GELB) (LAENGE . 7)))
```

```
> (count-if #'gelb-p objects)
1

> (count-if-not #'gelb-p objects)
2

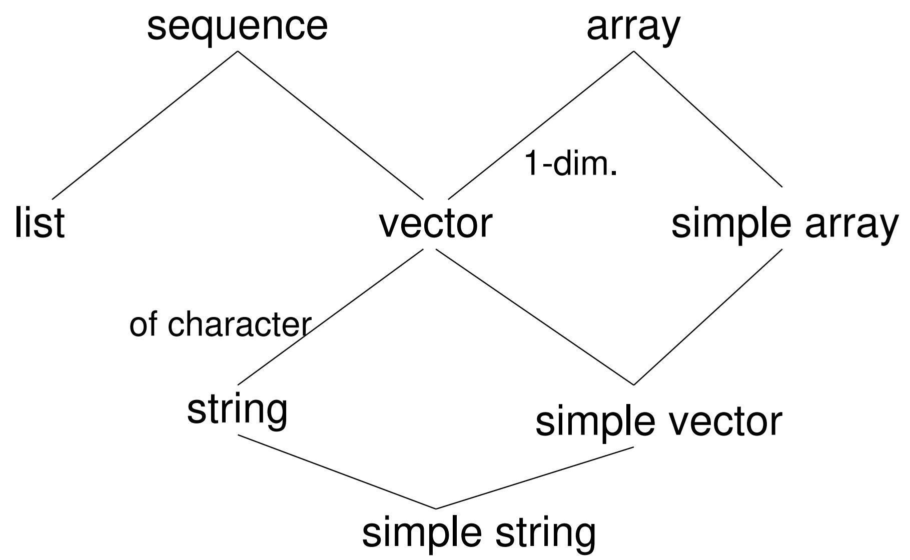
> (find-if #'gelb-p objects)
```

```
((NAME . B) (TYP . WUERFEL) (FARBE . GELB) (LAENGE . 7))
```

```
> (find-if-not #'gelb-p objects)
```

```
((NAME . A) (TYP . WUERFEL) (FARBE . ROT) (LAENGE . 5))
```

Sequences, Arrays und deren Untertypen



Print-Form einer Sequence/eines Array:

Vektoren: `#(`⟨*list-of-elements*⟩`)`

Arrays: `#`⟨*dim*⟩`A(`⟨*list-of-elements*⟩`)`

Strings: `"`⟨*characters*⟩`"`

Beispiele:

```
#(a b c)             ; Vektor mit 3 Elementen
#()                  ; Leerer Vektor

#(a b c)             ; 1-dim. Array
#1A(a b c)           ; dto.
#2A((a b c) (d e f)) ; 2-dim. Array

#(#\a #\b #\c)       ; = String "abc"
"abc"
```

Arrays

Erzeugung:

```
> (setq *kiste*
        (make-array 5 :initial-element nil))
#(NIL NIL NIL NIL NIL)
```

0	1	2	3	4
NIL	NIL	NIL	NIL	NIL

Zugriff auf ein Array-Element:

```
> (aref *kiste* 0)
NIL

> (setf (aref *kiste* 0) 'b)
B
```

0	1	2	3	4
B	NIL	NIL	NIL	NIL

```
> (aref *kiste* 0)
B

> (aref *kiste* 5)
Error: Index(s) to array function is/are out
       of range of the array.
```

- Erzeugen eines Arrays mit `make-array`
- Zugriff auf ein Array-Element mit `aref`
- Schreiben mit `(setf (aref ...)` ⟨*value*⟩`)`
- Erstes Element eines Arrays hat Index 0;
  Indizierung: *row-major*/zeilenweise

Arrays (Forts.)

Mehrdimensionale Arrays:

```
>(setq *schachtel*
        (make-array '(2 5) :initial-element 9))
#2A((9 9 9 9 9) (9 9 9 9 9))

> (aref *schachtel* 0 3)
9
```

	0	1	2	3	4
0	9	9	9	9	9
1	9	9	9	9	9

Array vorbesetzen:

```
> (setq *meine-kiste*
        (make-array 5
               :initial-contents '(A B C D E)))
#(A B C D E)

> (aref *meine-kiste* 3)
D
```

0	1	2	3	4
A	B	C	D	E

```
> (setq *meine-schachtel*
        (make-array '(2 3)
          :initial-contents '((A B C) (D E F))))
#2A((A B C) (D E F))

> (aref *meine-schachtel* 0 1)
B
```

	0	1	2
0	A	B	C
1	D	E	F

Eigenschaftslisten/Property-Listen/P-Listen

Allgemeines:

- Symbol besitzt gleichzeitig *Wert* und *property value*.
- *property values* werden (wie A-Listen) symbolisch indiziert.
- *Property-Liste* oder *P-Liste* eines Symbols enthält alle Namen und Werte von Properties, die zu diesem Symbol gehören.
- Änderungen von Properties sind immer *global*, da jedes Symbol global existiert und eindeutig ist.

Verwendung:

- globale Datenbank, wenn Indizierung über Symbol(name) „natürlich“ ist
- Abspeichern von zusätzlichen Eigenschaften der Wert- oder Funktionsbindung und von interner Information
- Gilt als antiquiert weil globale Speicherung schlecht für Programmstruktur

Properties – `get` und `(get setf)`

```
> (get 'B 'farbe)
NIL

> (setf (get 'B 'farbe) 'rot)
ROT

> (get 'B 'farbe)
ROT

> (get 'B 'liegt-auf)
NIL

> (setf (get 'B 'liegt-auf) '(C))
(C)

> (push 'A (get 'B 'liegt-auf))
(A C)

> (get 'B 'liegt-auf)
(A C)
```

- *property lists* werden für Symbole vereinbart, nicht für Symbol-Wert-Bindungen.
- Änderungen von Properties sind *immer* global.
- `(get` ⟨*symbol*⟩ ⟨*propname*⟩`)` liefert den Wert einer Property bzw. `NIL`, wenn diese nicht existiert.
- `(setf (get` ⟨*symbol*⟩ ⟨*propname*⟩`)` ⟨*value*⟩`)` setzt für ein Symbol den Wert einer Property.

Properties – `symbol-plist` und `remprop`

```
> (setf (get 'B 'farbe) 'weiss)
WEISS

> (setf (get 'B 'form) 'wuerfel)
WUERFEL

> (symbol-plist 'B)
(FORM WUERFEL LIEGT-AUF (A C) FARBE WEISS)

> (remprop 'B 'farbe)
T

> (symbol-plist 'B)
(FORM WUERFEL LIEGT-AUF (A C))
```

- `(symbol-plist` ⟨*symbol*⟩`)` liefert die vollständige Property-Liste eines Symbols.
- `(remprop` ⟨*symbol*⟩ ⟨*propname*⟩`)` löscht eine Property für ein Symbol.
- Wird eine vorhandene Property nochmals gesetzt, wird im Gegensatz zur A-Liste der alte Wert überschrieben.

Hashtabellen (Streuspeicherverfahren)

```
> (setq *tel-tab*
        (make-hash-table :size 200
                         :test #'equal))
#<equal hash-table with 0 entries @ #x41072a>

> (setf (gethash "Maier" *tel-tab*) 345678)
345678

> (setf (gethash "Mueller" *tel-tab*) 876543)
876543

> (gethash "Maier" *tel-tab*)
345678
T

> (gethash "Szymanowski" *tel-tab*)
NIL
NIL

> (hash-table-count *tel-tab*)
2

> (clrhash *tel-tab*)
#<equal hash-table with 0 entries @ #x41072a>
```

Hashtabellen (Forts.)

- Erzeugen einer Hashtabelle mit `make-hash-table` (mit optionaler Vergleichsfunktion, Anfangsgröße, Größe und Grenzwert bei notwendiger Tabellenvergrößerung)
- Hashtabellenobjekte in Lisp werden, wenn notwendig, *automatisch* vergrößert!
- Zugriff auf einen Eintrag mit `gethash` zum Lesen bzw. `(setf (gethash ...) ...)` zum Schreiben
- Schlüssel kann jedes Lisp-Objekt sein, nicht nur Symbole
- Löschen eines Eintrages mit `remhash`
- Löschen der gesamten Hashtabelle mit `clrhash`
- Durchlaufen: `maphash` (Reihenfolge ist unbestimmt!)

```
> (maphash #'(lambda (key value)
               (format t "~% Name: ~a Nummer: ~a"
                       key value))
           *tel-tab*)
 Name: Mueller Nummer: 876543
 Name: Maier Nummer: 345678
NIL
```

Strukturen – Motivation

Beispiel: Repräsentation eines geometrischen Objekts.

Bisher: Verwendung von A-Listen

```
> (defun make-wuerfel (&key (name 'no-name)
                            (typ 'wuerfel)
                            (laenge 1)
                            (farbe 'rot))
    (list (list 'name name)(list 'typ typ)
          (list 'laenge laenge)(list 'farbe farbe)))
```

```
> (setq objekt1 (make-wuerfel :name   'wuerfel-A
                              :laenge 8  :farbe 'rot))
((NAME WUERFEL-A)(TYP WUERFEL)(LAENGE 8)(FARBE ROT))
```

Besser: Verwendung von Strukturen

```
> (defstruct (wuerfel)
     (name 'no-name)
     (laenge 1)
     (farbe 'keine-farbe)
     (anz-flaechen 6))
WUERFEL
```

```
> (setq objekt1 (make-wuerfel :name   'wuerfel-A
                              :laenge 8  :farbe  'rot))
#S(WUERFEL :name WUERFEL-A :laenge 8
           :farbe ROT :anz-flaechen 6)
```

Strukturen – Einführung

Funktion `defstruct` definiert einen Verbund/Record:

- eine Art neuer Datentyp (hier: Typ `wuerfel`)
- Datenstruktur mit *benannten* Feldern (*Slots*)
- automatisch definierte *Zugriffsfunktionen* für jedes Feld
- automatisch definierter *Konstruktor* zur Erzeugung einer neuen Instanz
- automatisch definiertes Typprädikat

Beispiel:

```
> (defstruct (wuerfel)
     (name 'no-name))
```

erzeugt automatisch:

- Zugriffsfunktion `wuerfel-name` für Feld `name`
- Konstruktor `make-wuerfel`
- Prädikat `wuerfel-p`

Strukturen – Zugriff auf Felder

Selektion (Lesezugriff):

```
> (wuerfel-farbe objekt1)
ROT
```

```
> (wuerfel-laenge objekt1)
8
```

Mutation (Modifikation, Schreibzugriff):

```
> (setf (wuerfel-farbe objekt1) 'BLAU)
BLAU
```

```
> (wuerfel-farbe objekt1)
BLAU
```

Typprädikat:

```
> (wuerfel-p objekt1)
T
> (wuerfel-p 56)
NIL
```

```
> (type-of objekt1)
WUERFEL
```

Strukturen – Printformen

Ausgabe eines Objekts:

```
> objekt1
#S(WUERFEL :name WUERFEL-A :laenge 8
           :farbe ROT :anz-flaechen 6)
```

Ausgabe einer Beschreibung:

```
> (describe objekt1)
It is a structure of type WUERFEL.
It has these slots:
  NAME                    WUERFEL-A
  LAENGE                  8
  FARBE                   BLAU
  ANZ-FLAECHEN            6
```

Strukturen – Hierarchische Definition

Strukturdefinitionen können mit `:include` von anderen Strukturen erben:

```
> (defstruct (koerper)
      (name 'no-name)
      (farbe 'keine-farbe))
KOERPER

> (defstruct (kugel (:include koerper))
      (anz-flaechen 1 :read-only t))
KUGEL

> (defstruct (wuerfel (:include koerper))
      (laenge 1)
      (anz-flaechen 6 :read-only t))
WUERFEL


> (setq objekt1 (make-wuerfel :name 'wuerfel-A))
#S(WUERFEL :name WUERFEL-A :farbe KEINE-FARBE
           :laenge 1 :anz-flaechen 6)

> (wuerfel-name objekt1)
WUERFEL-A

> (wuerfel-farbe objekt1)
KEINE-FARBE

> (koerper-name objekt1)
WUERFEL-A
```

Strukturen – Zusammenfassung

- Strukturen dienen der Datenabstraktion.
- Strukturen definieren einen Verbund/Record von Feldern (*Slots*) und Feldwerten.
- Bei der Definition ist die Angabe von Standardwerten möglich.
- Erben von bereits definierten Typen ist (mit `:include`) möglich.
- Zugriffsfunktionen, Konstruktor und Typprädikat werden bei der Definition der Struktur mit `defstruct` automatisch erzeugt.
- `(make-`⟨*name*⟩ `:`⟨*feld-1*⟩ ⟨*wert-1*⟩ ... `:`⟨*:feld-n*⟩ ⟨*wert-n*⟩`)` erzeugt eine Instanz der Stuktur ⟨*name*⟩.
- ⟨*name*⟩-⟨*feld-i*⟩ selektiert den Wert von ⟨*feld-i*⟩.
- `setf` ⟨*feld-i*⟩ wird zur Mutation verwendet.
- Prädikat ⟨*name*⟩`-p` überprüft, ob ein Objekt von diesem Typ ist.

Defstruct-Option `:constructor`

`:constructor` ⟨*name*⟩ ⟨*arglist*⟩

Objekt wird durch Aufruf der Funktion `(`⟨*name*⟩ `...)` statt `(make-`⟨*strukturname*⟩ `...)` erzeugt.

Beispiel ohne `:constructor`:

```
> (defstruct foo a b)
FOO

> (make-foo :a 1 :b 2)
#S(FOO :A 1 :B 2)
```

Beispiel mit `:constructor`:

```
> (defstruct
    (foo (:constructor
            create (a &key ((:val-b b)))))
    a b)
FOO

> (create 1 :val-b 2)
#S(FOO :A 1 :B 2)

> (make-foo :a 1 :b 2)
Error: Function ‘make-foo’ undefined.
```

Defstruct-Option `:print-function`

`:print-function` ⟨*lambda-expr*⟩

spezifiziert, wie das Objekt ausgegeben werden soll.

Beispiel:

```
(defstruct
  (foo (:print-function
          (lambda (obj stream depth)
             (declare (ignore depth))
             (print
                (list 'defstruct-foo
                      'slot-a (foo-a obj)
                      'slot-b (foo-b obj))
               stream))))
  a b)


> (make-foo :a 1 :b 2)
(DEFSTRUCT-FOO SLOT-A 1 SLOT-B 2)
```

Defstruct-Option `:conc-name`

`:conc-name` ⟨*symbol*⟩

spezifiziert den gemeinsamen Präfix in den Namen der Slot-Zugriffsfunktionen (kein Präfix, falls ⟨*symbol*⟩ = `NIL`).

Beispiel:

```
(defstruct
  (foo (:conc-name nil)
       (:print-function
          (lambda (obj stream depth)
             (declare (ignore depth))
             (print
                (list 'defstruct-foo
                      'slot-a (a obj)
                      'slot-b (b obj))
                stream))))
  a b)


> (setq obj (make-foo :a 1 :b 2))
(DEFSTRUCT-FOO SLOT-A 1 SLOT-B 2)

> (a obj)
1

> (foo-a obj)
Error: Function ‘foo-a’ undefined.
```

Teil 5

Programmiertechniken

- Rekursion und Iteration
- Abstraktion
- Programmierstil
- Funktionale Programmierung

Rekursion vs. Iteration

Definition: Länge einer Liste

1. Eine leere Liste `()` hat die Länge 0.
2. Eine Liste ist um 1 länger als ihr Rest (`cdr`/`rest`).

- Rekursive Beschreibung:
  Induktionsanfang und -schluß ausprogrammieren

```
(defun laenge-rek (liste)
  (if (null liste)
      0
    (1+ (laenge-rek (cdr liste)))))
```

- Iteration mit `do`:
  Schritt programmieren – Verifikation mit Invariante

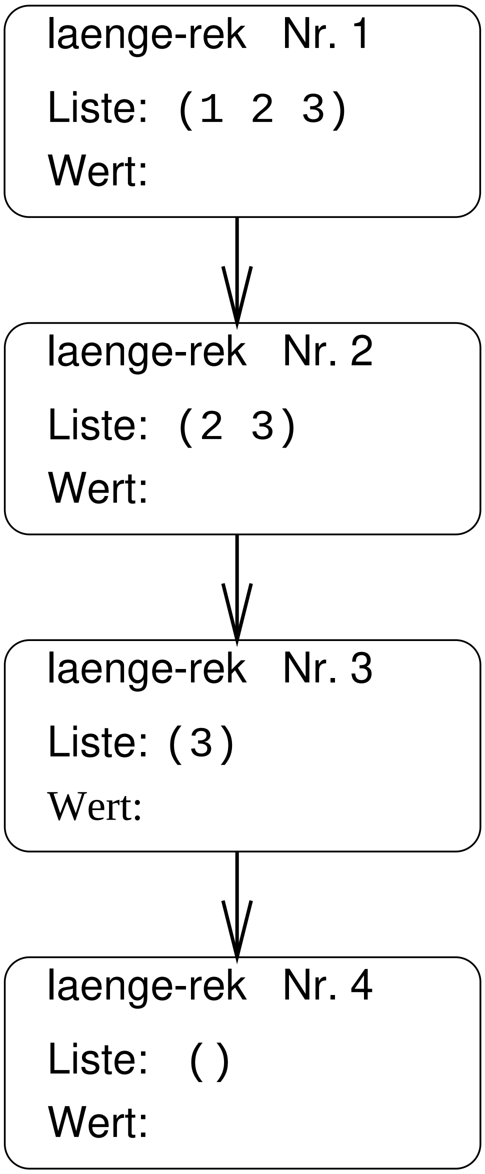
```
(defun laenge-do (liste)
  (do ((l liste (cdr l))
       (erg 0 (1+ erg)))
      ((null l) erg)))
```

- Iteration rekursiv formuliert:

```
(defun laenge-iter (liste)
  (laenge-ergvar liste 0))
(defun laenge-ergvar (liste erg)
  (if (null liste)
      erg
    (laenge-ergvar (cdr liste) (1+ erg))))
```

Ausführung der Rekursion

Aufruf: `(laenge-rek '(1 2 3))`



- Jeder Aufruf ist für sich korrekt.
- Die Argumente werden auf dem Weg in die Tiefe bearbeitet.
- Die Werte werden auf dem Rückweg berechnet.

Ablauf der Iteration

Die Korrektheit der Schleife prüft man mit 3 Bedingungen:

- Die Vorbedingung beschreibt die Aufgabenstellung.
  Vorbedingung: `erg` $= 0$, `l` ist komplette Liste.
- Die Invariante gilt nach jedem Durchlauf des Rumpfs.
  Gesamtlänge = Länge der Restliste `l` + `erg`
- Nachbedingung und Invariante zusammen beschreiben das Ziel.
  Nachbedingung: Liste `l` leer.

Ablauftabelle für `(laenge-do '(1 2 3))`

Variablen	`l`	`erg`
Init	`liste`	0
Schritt	`(cdr l)`	`(1+ erg)`
Init	`(1 2 3)`	0
Zustand 1	`(2 3)`	1
Zustand 2	`(3)`	2
Zustand 3	`()`	3

In Zustand 3 ist die Nachbedingung `(null l)` erfüllt, der Wert von `erg`, 3, wird zurückgegeben.

Durchwandern von Listen

Rekursives Durchwandern:

```
(defun mapcar-rek (func liste)
  (if (null liste)
      liste
    (cons (funcall func (first liste))
          (mapcar-rek func (rest liste)))))
```

Rekursives Durchwandern – destruktive Fassung:

```
(defun nmapcar-rek (func liste)
  (unless (null liste)
    (setf (first liste)
          (funcall func (first liste)))
    (nmapcar-rek func (rest liste))
    liste))
```

Iteratives Durchwandern mit `do`:

(Destruktives `nreverse` statt `reverse` hier verwendbar, da Elemente der Ergebnisliste mit `cons` *neu* erzeugt werden.)

```
(defun mapcar-do (func liste)
  (do ((l liste (rest l))
       (erg ()
            (cons (funcall func (first l))
                  erg)))
      ((null l) (nreverse erg))))
```

Faustregel: rekursives Durchwandern erhält die Reihenfolge,
iteratives Durchwandern dreht die Liste um.

Tiefenrekursion

Kombination der Ergebnisse mehrerer rekursiver Aufrufe
⇒ *nicht endständige* Rekursion
Bei Listen: *Tiefenrekursion*

Beispiel: Anzahl der Atome einer geschachtelten Liste als Verallgemeinerung der Funktion `laenge`:

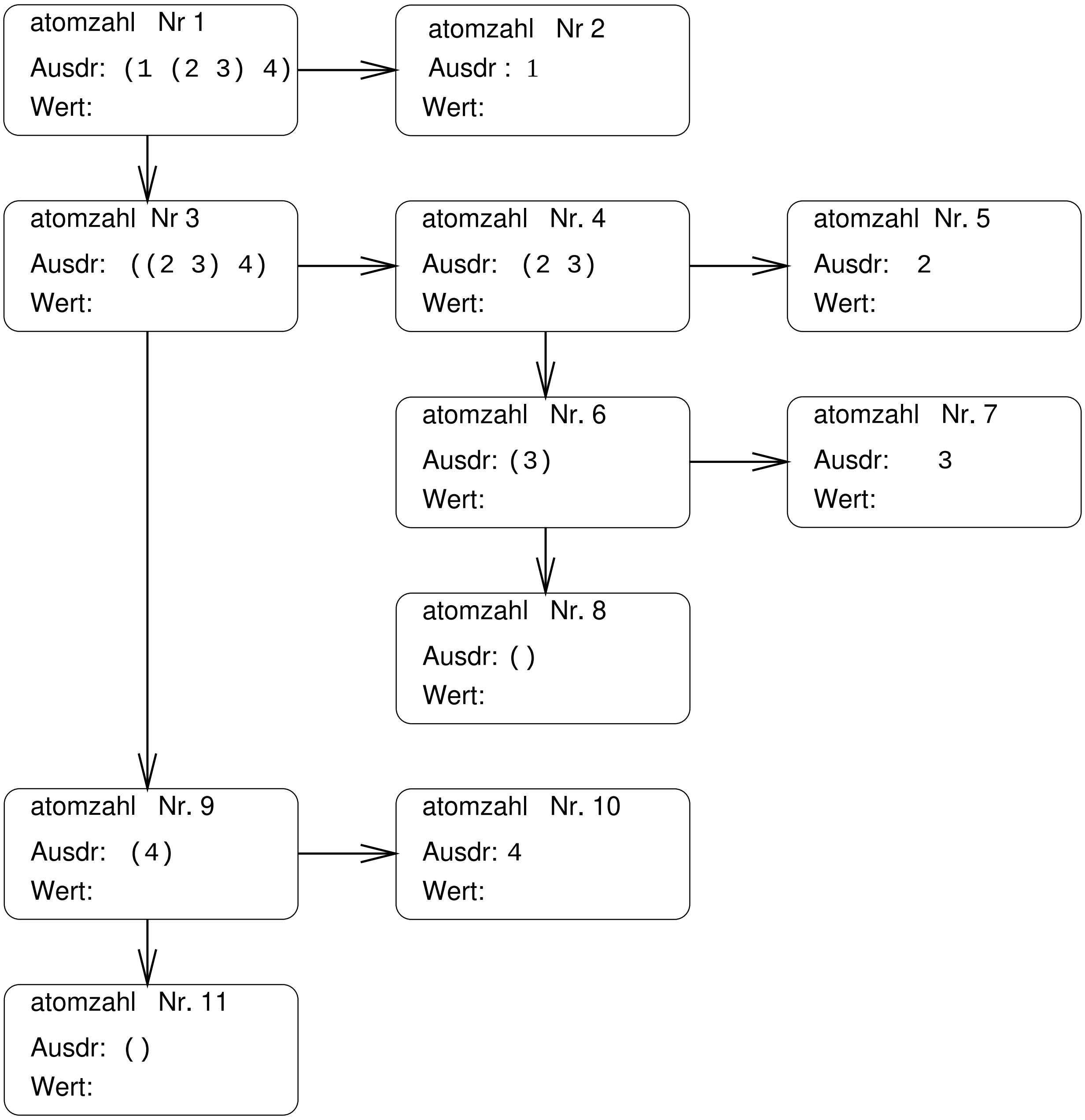
```
(defun atomzahl (liste)
  (cond ((null liste) 0)
        ((atom (first liste))
         (+ 1
            (atomzahl (rest liste))))
        (t (+ (atomzahl (first liste))
              (atomzahl (rest liste))))))
```

Vereinfachte rekursive Definition:

- Die leere Liste `()` enthält 0 Atome.
- Jedes Atom „enthält“ 1 Atom.
- Listen enthalten soviel Atome wie Kopf und Rest zusammen.

```
(defun atomzahl (expr)
  (cond ((null expr) 0)
        ((atom expr) 1)
        (t (+ (atomzahl (first expr))
              (atomzahl (rest expr))))))
```

Ablauf der Tiefenrekursion



- Es bildet sich ein Aufrufbaum.
- Aktueller Zustand des Stapels:
  Weg von Wurzel zum augenblicklichen Knoten.
- Keine (einfache) iterative Fassung möglich.

Abstraktion

1. Abstraktionsebenen im Programm (z. B. ADT)
   - hilfreich beim Top-Down-Entwurf
   - höhere Ebene enthält Plan, untere Ebene führt aus
   - Schnittstellen bleiben stabil
   - Ebenen unabhängig änderbar
2. Abstraktion durch Generalisierung:
   Erkennen struktureller Gemeinsamkeiten
   (a) Trennung zwischen Struktur und Konkretisierung
   (b) statt ausprogrammiertem Wissen:
   explizite Darstellung + Interpreter
   (c) weitere Argumente einführen
   (d) allgemeinere Argumente zulassen

Vorteile:

- Abstraktionsformen dienen der *Separation of Concerns*
- Ziel: selbstdokumentierendes Programm

Generalisierung mit einer Datenliste

Generalisierungsschritt: Code $\Longrightarrow$ Interpreter + Daten

Ausprogrammiert: Briefporto, Einfachversion:

```
(defun brief (gewicht)
  (cond ((< gewicht 20) 1.10)
        ((< gewicht 100) 2.20)
        ((< gewicht 250) 3.00)
        ((< gewicht 1000) 4.40)))
```

Datenliste: erleichtert es, Berechnungsmodus zu ändern.

```
(setq *porto* '((20 1.10) (100 2.20)
                (250 3.00) (1000 4.40)))
(defun bis-zu (gewicht daten)
  (let ((porto (find-if #'(lambda (datum)
                            (< gewicht (first datum)))
                        daten)))
    (when porto (second porto))))
(defun brief (gewicht)
  (bis-zu gewicht *porto*))
```

Begriffe einführen: Gebühren werden oft erhöht,
der Berechnungsmodus wird jedoch selten geändert.

```
(setq *briefe* '((20 :standard) (100 :kompakt)
                 (250 :gross) (1000 :schwer)))
(setq *porto* '((:standard 1.10) (:kompakt 2.20)
                (:gross 3.00) (:schwer 4.40)))
(defun brief (gewicht)
  (second (assoc (bis-zu gewicht *briefe*) *porto*)))
```

ADT Zeitspanne

Zeitspannen werden in Tagen, Stunden, Minuten und Sekunden angegeben.
Sie sollen intern in einer Normalform abgelegt werden.

Konstruktor:

```
(defun zeitspanne (&key (d 0) (h 0)
                        (min 0) (sec 0))
  (let* ((normsec (mod sec 60))
         (realmin (+ min (floor sec 60)))
         (normmin (mod realmin 60))
         (realh   (+ h (floor realmin 60)))
         (normh   (mod realh 24))
         (reald   (+ d (floor realh 24))))
    (list normsec normmin normh reald)))
```

Eine der Zugriffsfunktionen:

```
(defun stunden (z)
  (third z))
```

Ausgabefunktion:

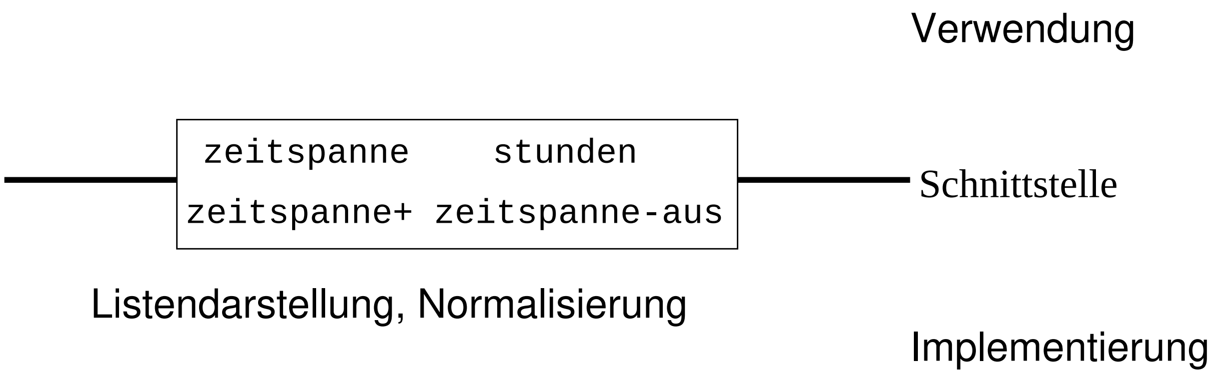
```
(defun zeitspanne-aus (z)
  (list :d (fourth z)   :h (third z)
        :min (second z) :sec (first z)))
```

Eine mögliche Operation auf Zeitspannen:

```
(defun zeitspanne-+ (z1 z2)
  (norm (mapcar #'+ z1 z2) *zeitbasen*))
```

ADT Zeitspanne II

Datenabstraktionsbarrieren:



Beispiele:

```
> (setq z1 (zeitspanne :d 2    :h 25
                       :min 70 :sec 70))
(10 11 2 3)
> (zeitspanne-aus z1)
(:D 3 :H 2 :MIN 11 :SEC 10)
> (stunden z1)
2
```

Konstruktor kann mit dem Ergebnis der gezeigten Ausgabefunktion arbeiten:

```
> (equal (apply #'zeitspanne (zeitspanne-aus z1))
         z1)
T
```

Generalisierung

Ziffern bezüglich Basis normalisieren:

```
(defun normbase (werte basis)
  (if (or (null werte)
          (and (zerop (first werte))
               (null (rest werte))))
      ()
    (cons (mod (first werte) basis)
          (normbase
            (cons (+ (if (second werte)
                         (second werte)
                       0)
                     (floor (first werte) basis))
                  (rest (rest werte)))
            basis))))
```

Statt fester Basis endliche Liste von Basen:

```
(defun norm (werte basen)
  (if (null basen)
      werte
    (cons (mod (first werte) (first basen))
          (norm (cons (+ (if (second werte)
                             (second werte)
                           0)
                         (floor (first werte)
                                (first basen)))
                      (rest (rest werte)))
                (rest basen)))))
```

Abstraktion von der Zeitspanne

- Zeitspanne als Stellenwertsystem

```
(defvar *zeitbasen* '(60 60 24))

(defun zeitspanne (&key (d 0) (h 0)
                        (min 0) (sec 0))
  (norm (list sec min h d) *zeitbasen*))
```

  Typisch: Konkretisierung nutzt Funktionalität von `norm` nicht vollständig aus.

- Fernziel der Abstraktion:
  Funktion, die einen Stellenwertsystem-ADT für eine gegebene Liste der Stellennamen und -basen erzeugt.

```
(defun defstellenwertsys (name spec)
  ...)
```

  Anwendung auf Zeitspannen:

```
> (defstellenwertsys 'zeitspanne
                     '((sec 60) (min 60) ...))
```

  Realisierung des ADT z. B. als Liste oder Struktur mit Zugriffsfunktionen und internen Daten.

  Vergleiche: objektorientierte Programmierung

Vergleich rekursiver Funktionen

```
(defun copy (liste)
  (cond ((null liste) ())
        (t (cons (first liste)
                 (copy (rest liste))))))

(defun summe (liste)
  (cond ((null liste) 0)
        (t (+ (first liste)
              (summe (rest liste))))))

(defun laenge (liste)
  (cond ((null liste) ())
        (t (+ 1
              (laenge (rest liste))))))

(defun my-mapcar (func liste)
  (cond ((null liste) ())
        (t (cons (funcall func (first liste))
                 (my-mapcar (rest liste))))))
```

Alle Funktionen haben im Prinzip denselben Aufbau
⇒ Generalisierung ist möglich und naheliegend

Flache Rekursion abstrakt

```
(defun akku (liste komb start)
  (cond ((null liste) start)
        (t (funcall komb
                (first liste)
                (akku (rest liste) komb start)))))

(defun copy (liste)
  (akku liste #'cons ()))
(defun summe (liste)
  (akku liste #'+ 0))
(defun produkt (liste)
  (akku liste #'* 1))

(defun laenge (liste)
  (akku liste
        #'(lambda (kopf resterg)
            (1+ resterg))
        0))

(defun my-mapcar (func liste)
  (akku liste
        #'(lambda (kopf resterg)
            (cons (funcall func kopf)
                  resterg))
        ()))
```

Tiefe Rekursion abstrakt

Definition von `atomzahl`:

```
(defun atomzahl (expr)
  (cond ((null expr) 0)
        ((atom expr) 1)
        (t (+ (atomzahl (first expr))
              (atomzahl (rest expr))))))
```

Diese Tiefenrekursion kann verallgemeinert werden:

```
(defun tief-akku (expr komb blatt start)
  (cond ((null expr) start)
        ((atom expr) (funcall blatt expr))
        (t (funcall
             komb
             (tief-akku (first expr)
                        komb blatt start)
             (tief-akku (rest expr)
                        komb blatt start)))))
```

```
(defun atomzahl (liste)
  (tief-akku liste
             #'+
             #'(lambda (blatt) 1)
             0))
```

Tiefe Rekursion abstrakt

Mit Hilfe der verallgemeinerte Funktion `tief-akku` können sehr einfach weitere Funktionen realisiert werden:

```
(defun tiefcopy (liste)
  (tief-akku liste #'cons #'identity  ()))

(defun atomsumme (liste)
  (tief-akku liste #'+ #'identity 0))

(defun flach (liste)
  (tief-akku liste #'append #'list ()))
```

Verwendungsbeispiele:

```
> (atomzahl '((1 2) (3) (4 (5 6 7))))
7
> (atomsumme '((1 2) (3) (4 (5 6 7))))
28
> (flach '((1 2) (3) (4 (5 6 7))))
(1 2 3 4 5 6 7)
```

Tiefe Rekursion II

Problem: keine Unterscheidung zwischen Rekursion in die Tiefe und in die Breite

Abhilfe: separate Kombinierer für Breiten- (`breit`) und Tiefenrekursion (`tief`)

```
(defun tief-akku-neu (expr breit tief blatt start)
  (cond ((null expr) start)
        ((atom expr) (funcall blatt expr))
        (t (funcall breit
             (funcall
               tief
               (tief-akku-neu (first expr)
                              breit tief blatt start)
               (first expr))
             (tief-akku-neu (rest expr)
                            breit tief blatt start)))))
```

```
(defun verzw-grad (liste)
  (tief-akku-neu
   liste
   #'cons
   #'(lambda (kopferg kopf)
       (if (listp kopf)
           (list (length kopf) kopferg)
         kopferg))
   #'identity
   ()))
```

```
> (verzw-grad  '((a a) (a b a) (a (b a) b c)))
((2 (A A)) (3 (A B A)) (4 (A (2 (B A)) B C)))
```

Kontrollabstraktion

Wenn man aus dem Code eine einfache sprachliche Beschreibung der Tätigkeit ablesen kann, ist der Code verständlich.

- Gib mir die Wurzeln der Zahlen aus `*werte*`.

  ```
  (mapcar #'sqrt *werte*)
  ```

- Ist einer der Werte geeignet?

  ```
  (some #'geeignet-p *werte*)
  ```

- Wieviele Werte sind kleiner als 20?

  ```
  (count-if #'(lambda (w) (< w 20)) *werte*)
  ```

- Gib mir alle Werte unter 20.

  ```
  (remove-if-not #'(lambda (w) (< w 20)) *werte*)
  ```

- Multipliziere zwei Listen mit Werten elementweise und summiere auf (Skalarprodukt).

  ```
  (reduce #'+ (mapcar #'* w1 w2))
  ```

- Sortiere Teileliste alphabetisch aufsteigend nach Namen.

  ```
  (sort *teile* #'string<= :key #'teil-name)
  ```

Code mit Kontrollabstraktion ist näher an diesen Sprechweisen als Rekursion oder Schleifen.

Grundbegriffe des Stils

Oft gibt es verschiedene Möglichkeiten, etwas in einem Programm zu realisieren.

Ergebnis dieser Ermessensentscheidungen nennt man *Stil*.

Eine gut geschriebenes Programm ist:

- verständlich
- erweiterbar
- effizient
- einfach zu entwickeln
- ermöglicht einfache Fehlersuche

Ziele:

- Korrektheit
- Robustheit
- Kompatibilität

Adressaten

Wer liest ein Programm ?

- Menschen
- Compiler
- Struktureditor
- Programmierwerkzeuge

Für den menschlichen Leser: Stilregeln

explizit: Das Entscheidende am Programm offenlegen. Keine versteckten Effekte eines Konstrukts mitverwenden. Keine Tricks: *don't be too clever!*

klar und knapp: Lines of code **not** written.

spezifisch: Nicht zu generell sein, dann sieht man am Konstrukt, was wirklich passiert.

erweiterbar: Programmteile funktionieren auch mit erweiterten Voraussetzungen.

konsistent: Gleiches gleich lösen.

konventionell: Gebräuchliche Konstrukte verwenden.

Separation of concerns: Verschiedene Bereiche getrennt behandeln, nicht zwei Dinge auf einmal tun wollen!

Programmgliederungsmittel

- Hierarchie von Funktionen (funktionale Dekomposition):
  Daten global
  Code: Definitionen, Variablen, Makros, Funktionen
  jeweils getrennt
- Modulstruktur:
  Hierarchie von Modulen mit privaten Daten
  Datenkapsel: Verwaltungsmodul mit Schnittstelle
  Code: Funktion und Variablen nach Modulen zusammen
- Datenabstraktion:
  Abstrakte Datentypen: Datentyp mit speziellen Zugriffen
  Erweitern durch Benutzen

  Code: Definition und Zugriff zusammen
- Objektorientiertes System:
  Klassen: systematische Definition von ADTs
  Verhalten der Objekte (Funktionen) an Klassen
  gebunden
  Erweitern durch Vererbung (Spezialisierung)
- Kontrollflußabstraktion:
  Spezielles Verhalten und Steuerung (Kontrolle) trennen
  Code: nach Aufgabenbereichen getrennt
- Spezialarchitekturen:
  z. B. Benutzerschnittstelle mit Callbacks
  Interpreter oder Verfahren mit Laufzeitcompilation
  Erweitern durch Einhängen eigenen Verhaltens

Kodierstil

- Code einrücken.
- Namenskonventionen einhalten.
  - Globale Variablen $*\langle \textit{name} \rangle*$
  - Prädikate enden auf `p` oder `-p`
- Selten benötigte oder von ferne referenzierte Namen nicht abkürzen!
  Rein lokal verwendete Bezeichner kann man abkürzen.
- Spezifisch und korrekt benennen.

  Bsp.: Parameter nicht `liste`, sondern `a-liste` nennen, wenn es eine A-Liste sein muß.

  Parameter nicht `liste` nennen, wenn es auch ein String sein kann.

  Auch gut: Abstrakt, nach Inhalt nennen, z. B. `bauteile`, Datentyp in den Kommentar.
- Ähnliches ähnlich gestalten!

  Wenn Funktionen dieselben Argumente haben, sollte auch deren Reihenfolge gleich sein.

Kommentare

Expliziter Code ist besser als jeder Kommentar. Kommentare ergänzen nur, was man am Code nicht sehen kann.

- Kommentarkonventionen:

`;;;;`	Abschnittsüberschriften
`;;;`	Beschreibungen vor Funktionen, Variablen ...
`;;`	Eigene Zeilen innnerhalb von Funktionen; wie Code eingerückt
`;`	Hinter einer Codezeile

- Bei Änderungen vor Funktion oder am Abschnitt Datum und Person angeben.
- Datenformate für Parameter etc. beschreiben.
- In der Funktion nicht kommentieren, was man ohnehin sieht, sondern Aktionen auf höherer Ebene beschreiben.
- Ungewöhnliches und Tricks vermerken.
- Kommunikation über Kommentare:
  Zeichen wie `xxx` oder `!!!` für „Da muß ich/jemand anderes nochmal dran.“
  Auch Fragen an andere Entwickler möglich.
- Innerhalb von Funktionen nicht überkommentieren, sonst verliert man den Überblick über den Code.

Design von Funktionen

- nicht zu groß (unter 1 Bildschirmseite)
  beliebte Größe: 4–20 Zeilen

- möglichst wenige, klar definierte Außenbeziehungen,
  am besten nur Parameter und Wert.
  Globaler Zustand: lieber eine Struktur als viele
  Einzelvariablen.

- Möglichst jeweils nur eine Eigenschaft:

  – Wertberechnung *oder* mutierende Nebenwirkung
    oder Ein-Ausgabe

  – Durchwandern vs. Einzelverarbeitung

- Konstrukte mit lokalen Auswirkungen benutzen:
  `let`, Fertigschleifen, blocks . . .
  Gültigkeitsbereiche sind hier sofort erkennbar.

- Zugriffe auf globale Variablen:
  für (Lese- und Schreib-)Zugriffe auf globale Daten.
  Änderung von Zustandsvariablen.
  Sonst: lieber als weiteren Parameter mitgeben.

- *„gefährliche“* Konstrukte:
  Nur für ihren speziellen Anwendungsbereich verwenden!
  Von anderen Berechnungen trennen – notfalls mit
  Makros.
  Reichweite der Auswirkung begrenzen.

Funktionale Programmierung

In Lisp sind Funktionen vollwertige Datenobjekte (*first class objekt*).

Daher darf man Funktionen

- als Variablenwert verwenden
- anderen Funktionen als Argument übergeben

  Wichtig: Argumentstruktur der übergebenen Funktion
- als Funktionswert zurückgeben

  Beachte: Variablenbindungen der ausgegebenen Funktion
  Closure: Variablen von äußerer lexikalischer Umgebungen mit eingebunden

Funktionen, die Funktionen als Wert oder Argument haben heißen *Funktionen höherer Ordnung*.

Verzögerte Bewertung

Problem: Wert kann noch nicht ausgerechnet werden.

Grund: beispielsweise fehlen noch Angaben
oder es besteht noch kein Bedarf
oder die vollständige Berechnung ist nicht möglich (z. B. unendliche Reihen).

Lösung: Berechnungsvorschrift mit aktuellem Berechnungszustand in einen *Thunk* verpacken.

Thunk:

- Funktion ohne Argumente, ohne Peripheriezugriff und ohne Zugriff auf globale Variablen (also keine Seiteneffekte).
- Thunk wird bei Bedarf aktiviert/aufgerufen.
- Berechnung ist vorherbestimmt.
- Realisierung: Berechnungsvorschrift wird mit Makro `delay` in Thunk verpackt und mit Funktion `force` aktiviert.

`delay` und `force`

```
(defmacro delay (form)
  `#'(lambda () ,form))
```

```
(defun force (thunk)
  (funcall thunk))
```

Das Makro `delay` verpackt `form` in einen Thunk, die Funktion `force` führt Thunk und damit `form` aus.

Beispiel:

```
(defun schweine-* (x y)
  (print "Grunzt bei der Arbeit")
  (* x y))
```

```
> (+ (schweine-* 3 4) 5)
"Grunzt bei der Arbeit"
17
> (setq *thunk* (delay (schweine-* 3 4)))
#<Interpreted Function (unnamed) @ #x68b4da>
> (force *thunk*)
"Grunzt bei der Arbeit"
12
```

Aufrufe von `delay` und `force` sind hier gleichwertig mit:

```
> (setq *thunk* #'(lambda () (schweine-* 3 4)))
#<Interpreted Function (unnamed) @ #x68f4b2>
> (funcall *thunk*)
"Grunzt bei der Arbeit"
12
```

ADT Folge

Darstellung: Eine Folge ist eine Liste aus

- dem Kopfglied `folgen-kopf` und
- einem Thunk, der die Restglieder der Folge *erzeugt*.

Dieser Rest ist Objekt des ADT Folge.

Konstruktor: `erz-folge` erhält Kopfglied und Rest-Thunk.

```
(defun erz-folge (kopf rest)
  (cons kopf rest))
```

Selektoren: `folgen-kopf` gibt das Kopfglied zurück, `folgen-rest` ein Objekt des ADT Folge ohne erstes Glied.

```
(defun folgen-kopf (folge)
  (first folge))

(defun folgen-rest (folge)
  (force (rest folge)))
```

Folgenglieder in Liste: (`n` Stück)

```
(defun folgen-glieder (folge n)
  (unless (zerop n)
    (cons (folgen-kopf folge)
          (folgen-glieder (folgen-rest folge)
                          (1- n)))))
```

Erzeugung von Folgen

Natürliche Zahlen:

```
(defun natzahlen-ab (n)
  (erz-folge n
             (delay (natzahlen-ab (1+ n)))))
```

```
> (setq *natzahlen* (natzahlen-ab 1))
(1 . #<Interpreted Closure (:INTERNAL #) @ #x8>)
> (folgen-kopf *natzahlen*)
1
> (folgen-rest *natzahlen*)
(2 . #<Interpreted Closure (:INTERNAL #) @ #x9>)
> (folgen-glieder (natzahlen-ab 20) 5)
(20 21 22 23 24)
```

Allgemeine Schrittfunktion:

```
(defun schrittfolge (start schritt)
  (erz-folge
    start
    (delay (schrittfolge (funcall schritt start)
                         schritt))))
```

Beispiel: Folge $a_{n+1} = 2a_n$

```
(setq *pot2*
      (schrittfolge 1 #'(lambda (x) (* 2 x))))
> (folgen-glieder *pot2* 10)
(1 2 4 8 16 32 64 128 256 512)
```

Rechnen mit Folgen

Addition:

```
(defun folgen+ (str1 str2)
  (erz-folge (+ (folgen-kopf str1)
                (folgen-kopf str2))
             (delay (folgen+ (folgen-rest str1)
                             (folgen-rest str2)))))
```

Beispiel: Folge $a_n = 2n \qquad (= n + n)$

```
> (folgen-glieder (folgen+ *natzahlen* *natzahlen*)
                  10)
(2 4 6 8 10 12 14 16 18 20)
```

Allgemeine Kombination:

```
(defun folgen-map (func &rest folgen)
  (erz-folge
    (apply func (mapcar #'folgen-kopf folgen))
    (delay (apply #'folgen-map
                  func
                  (mapcar #'folgen-rest folgen)))))
```

Beispiel: Folge $a_n = n^3 \qquad (= n * n * n)$

```
> (folgen-glieder
    (folgen-map #'*
        *natzahlen* *natzahlen* *natzahlen*)
    10)
(1 8 27 64 125 216 343 512 729 1000)
```

Teil 6

Ergänzungen

- Input und Output
- Behandlung von Dateinamen (*Pathnames*)
- Symbole und Packages
- Kontrollfluß (Fehler etc.)

Ein-/Ausgabe und Streams

- Ausgabefunktionen schreiben Daten (ihr Argument) auf Streams.
- Eingabefunktionen lesen Daten von Streams und geben sie als Lisp-Objekte zurück.

Streams bilden abstrakte Schnittstelle zur Peripherie:

- **Terminalstreams**
  - `*standard-input*` und `*standard-output*` sind Defaultstreams für Ein-/Ausgabefunktionen.

    Default ist `*terminal-io*`, kann aber geändert werden.
  - `*terminal-io*` Terminal (Konsole) des Benutzers.
  - Die Streams `*error-output*`, `*query-io*`, `*debug-io*` und `*trace-output*` erlauben es, die entsprechende Ausgabe bei Bedarf umzulenken.
- **Dateien**
- **Stringstreams** arbeiten intern, ohne Peripheriegerät;

  Ausgabe in String, Lesen aus String ist möglich.

Ausgabefunktionen

```
> (progn               > (progn               > (progn
    (prin1 "abc")          (print "abc")          (princ "abc")
    (prin1 'a))            (print 'a))            (princ 'a))
"abc"A                                        abcA
A                      "abc"                  A
                       A
                       A
```

- `prin1` gibt die *printed representation* aus; kann anschließend (meist) mit `read` eingelesen werden.
- `print` ist wie `prin1`, zusätzlich wird eine neue Zeile begonnen und danach ein Leerzeichen eingefügt.
- `princ` ist wie `prin1`, jedoch werden keine Escape-Zeichen eingefügt – das Ergebnis sieht für den Menschen „schöner"/lesbarer aus.
- Alle Funktionen geben ihr Argument zurück und haben ein weiteres, optionales Argument: den Ausgabestream.
- Weitere Funktionen:

`terpri`	gibt Zeilenvorschub aus
`pprint`	„Pretty-Printing"
`princ-to-string`	Drucken „in" einen String
`write`	konfigurierbare Ausgabe
`format`	formatierte Ausgabe

Formatierte Ausgabe mit `format`

`format` setzt Argumente in ein Ausgabemuster (Formatstring) ein, Steueranweisungen in diesem Ausgabemuster beginnen mit `~`.

Beispiele:

```
> (format t "Erst ~S dann ~A?"
          "A sagen," "B sagen")
Erst "A sagen," dann B sagen?
NIL
```

```
> (format t "Oben: ~D ~%Unten: ~6,2F " 1 2)
Oben: 1
Unten:   2.00
NIL
```

```
> (format nil "~Dfast~Dyou" 2 4)
"2fast4you"
```

```
> (format t "Paare:~{ (~S,~S)~}."
          '(Peter Eva
            Stefan Ellen
            Walter Jackie))
Paare: (PETER,EVA) (STEFAN,ELLEN) (WALTER,JACKIE).
NIL
```

`format` – formatierte Ausgabe (Forts.)

Syntax:

(format ⟨*stream*⟩ ⟨*string*⟩ ⟨*arg1*⟩...⟨*argn*⟩)

- das erste Argument ⟨*stream*⟩ spezifiziert den Ausgabestream:
  - falls (eq ⟨*stream*⟩ t), dann wird auf `*standard-output*` geschrieben,
  - falls (eq ⟨*stream*⟩ nil), dann wird nichts ausgegeben; der Funktionswert von `format` ist ein String mit der Ausgabe,
  - sonst wird auf ⟨*stream*⟩ geschrieben.
- ⟨*string*⟩ enthält spezielle, durch `~` gekennzeichnete Druckanweisungen; eine Auswahl der wichtigsten:

ID	*Argument*	*Effekt*
`~%`	–	Zeilenvorschub
`~A`	Lisp-Objekt	Ausgabe ohne Escape-Zeichen
`~S`	S-Expr.	Ausgabe mit Escape-Zeichen
		(einlesbare *printed representation*)
`~D`	Integer	Ausgabe als Dezimalzahl
`~F`	Zahl	Ausgabe als Gleitkommazahl

Eingabefunktionen

```
> (read)                  > (read-line)
Benutzer: 2 <Return>      Benutzer: 2 <Return>
Wert:     2               Wert:     "2"

> (read)                   > (read-line)
Benutzer: (1 2 <Return>    Benutzer: (1 2 <Return>
          4)               Wert:     "(1 2"
Wert:     (1 2 4)
```

- `read` liest das nächste Datenobjekt *vollständig* ein; wenn notwendig, auch über das Zeilenende hinweg.
- `read` meldet einen Fehler am Ende der Eingabe (Stream oder Datei); Verhalten bei *EOF* kann geändert werden:

  `(read` ⟨*stream*⟩ ⟨*eof-error-p*⟩ ⟨*eof-value*⟩ ⟨*recursive-p*⟩`)`

- `read-line` liest genau eine Zeile und gibt sie als String zurück.
- `read-char` liest das nächste Zeichen und gibt es zurück.
- `peek-char` ist wie `read-char`, nur bleibt das Zeichen in der Eingabe stehen.

Streams

- Streams sind Objekte, die als *Datenquellen* oder *-senken* dienen.
- Streams werden mit Dateien, (interaktiven) Terminals oder Strings verbunden.
- Streams akzeptieren oder liefern einen speziellen Typ von Daten, z. B. nur Characters, Integers, ...

  Daten in Lisp werden normalerweise von *character streams* gelesen oder auf solche geschrieben.
- **Öffnen:** Streams können mit der Option `:direction` zum Nur-Lesen (`:input`), Nur-Schreiben (`:output`), Lesen-Schreiben (`:bidirectional`) oder zum Testen (`:probe`) geöffnet werden.
- **Schließen:** Streams werden mit der Funktion `close` geschlossen, d. h. Ein-/Ausgabe von bzw. auf diesen Stream ist danach nicht mehr möglich.
- **Typprädikat:** `streamp` testet, ob ein Objekt ein Stream-Objekt ist.

Dateien – Funktion `open`

```
(let ((mystream
        (open "meinedatei"
              :direction :output
              :if-does-not-exist :create
              :if-exists :overwrite)))
  (print 'hallo mystream)
  (close mystream))
```

- `(open` ⟨*dateiname*⟩ `:direction` ⟨*richtung*⟩ `... )`

  erzeugt einen Stream, öffnet die angegebene Datei ⟨*dateiname*⟩, weist den Stream der Datei zu und gibt ihn als Ergebnis zurück.

- `open` hat weitere optionale Argumente, wie den Datentyp und das Verhalten in bestimmten Fällen.

- `(close` ⟨*stream*⟩`)` schließt den angegebenen Stream.

- `(probe-file` ⟨*file*⟩`)` ≡ `(open` ⟨*file*⟩ `:direction :probe)`

- `read`, `read-line`, `read-char`, `peek-char` lesen von `*standard-input*`, wenn kein Stream als optionales Argument angegeben ist, sonst von diesem Stream.

- `write`, `princ`, `prin1`, `print`, `terpri` schreiben auf `*standard-output*`, wenn kein Stream als optionales Argument angegeben ist, sonst auf diesen Stream.

Dateien – Makro `with-open-file`

```
> (with-open-file
      (mystream "meinedatei" :direction :output)
    (print 'hallo mystream))
HALLO

> (with-open-file
      (mystream "meinedatei" :direction :input)
    (print (read mystream))
    (print (read mystream)))
HALLO
Error: eof encountered on stream #<.....>

> (with-open-file
      (mystream "meinedatei" :direction :input)
    (print (read mystream nil 'ende))
    (print (read mystream nil 'ende)))
HALLO
ENDE
ENDE
```

- `(with-open-file (`⟨*stream*⟩ ⟨*dateiname*⟩ `...)` ⟨*rumpf*⟩`)`

  öffnet die Datei ⟨*dateiname*⟩, erzeugt ein Streamobjekt ⟨*stream*⟩, auf das im ⟨*rumpf*⟩ zugegriffen werden kann.

- Optionen für das Öffnen wie bei der Funktion `open`.

- Datei wird bei Verlassen des Blocks auch bei Fehlern *automatisch* geschlossen.

Makro `with-open-file` – Beispiel

```
(setq *tel-tab*
      (make-hash-table :size 200 :test #'equal))
(setf (gethash "Maier" *tel-tab*) 345678)
(setf (gethash "Mueller" *tel-tab*) 876543)
```

Sichern der Hashtabelle:

```
(with-open-file (str "hashdat" :direction :output)
  (maphash #'(lambda (key val)
               (format str "~% (~s ~s)" key val))
           *tel-tab*))
```

Einlesen der Hashtabellen-Daten:

```
;; (clrhash *tel-tab*)
(with-open-file (str "hashdat" :direction :input)
  (do ((eintrag (read str nil) (read str nil)))
      ((not eintrag) nil)
    (if (and (listp eintrag)
             (= 2 (list-length eintrag))
             (not (gethash (first eintrag)
                           *tel-tab*)))
        (setf (gethash (first eintrag) *tel-tab*)
              (second eintrag))
      (warn "Eintrag ~S ist fehlerhaft!"
            eintrag))))
```

Dateinamen in Filesystemen – Pathnames

- In Lisp gibt es zwei Möglichkeiten, den Namen einer Datei oder eines Verzeichnisses zu repräsentieren:

  1. übliche Zeichenketten (*namestrings*),
  2. Pfadnamen (*pathnames*).

```
> (with-open-file (str "lispkurs/beispieldatei"
                       :direction :output)
    (print "Hallo" str)
    (print "Leute" str))
```

- Repräsentation mit Strings ist abhängig vom Betriebssystem, *Pathnames* sind unabhängig und flexibler.

- Komponenten eines *Pathname*:

  – Host (Name des Filesystems)
  – Device (Name des logischen oder physischen Geräts)
  – Directory (Verzeichnis, Dateigruppierung)
  – Name (Datei, evtl. auch Menge von Dateien)
  – Type (meist Dateierweiterung)
  – Version (Versionsnummer)

  Die meisten Betriebssysteme bzw. Implementierungen unterstützen nur eine Teilmenge dieser Komponenten.

Pathnames – Beispiele

Konversion von Namestrings zu Pathnames:

```
> (setq file
    (parse-namestring "lispkurs/pn/file.lisp"))
#p"lispkurs/pn/file.lisp"

> (pathname-name file)
"file"

> (pathname-directory file)
(:relative "lispkurs" "pn")
```

Konstruktion von Pathnames aus Komponenten:

```
> (make-pathname :directory '(:relative "lispkurs")
                 :name "file"   :type "lisp")
#p"lispkurs/file.lisp"

> (make-pathname :directory '(:relative :up)
                 :name "file")
#p"../file"

> (make-pathname :directory '(:relative :up :wild)
                 :name "file")
#p"../*/file"
```

Pathnames – Zusammenfassung

Ein *Pathname* ist eine (unvollständige) Spezifikation eines Dateinamens.

- `(parse-namestring` ⟨*thing*⟩`)` gibt ein Pathname-Objekt für ⟨*thing*⟩ (String, Pathname oder Stream) zurück, meist bei interaktiver Eingabe von Dateinamen verwendet.
- Die Funktionen `pathname-host`, `pathname-device`, `pathname-directory`, `pathname-name`, `pathname-type`, `pathname-version` geben die entsprechenden Pathname-Komponenten als (Liste von) Symbole, Strings oder Zahlen zurück.
- `make-pathname` baut ein Pathname-Objekt mit den angegebenen Komponenten auf.
- `(pathname` ⟨*object*⟩`)` konvertiert das ⟨*object*⟩ (Pathname, String, Symbol oder Stream) zu einem Pathname.
- `(truename` ⟨*pathname*⟩`)` versucht für ⟨*pathname*⟩ den „richtigen Namen“ im Filesystem herauszufinden; keine Wildcards erlaubt.
- `(pathnamep` ⟨*object*⟩`)` ≡ `(typep` ⟨*object*⟩ `'pathname)`

Logical Pathnames

- Pathnames sind nicht portabel, bei Änderungen an der Verzeichnisstruktur müssen diese im Programm angepaßt werden.
- Idee: *logical Pathnames*
  - Vorschriften legen Abbildung von „logischen Pfadnamen“ auf „reale Pfadnamen“ fest
  - logische Pfadnamen sind portabel
  - Abbildungsvorschriften sind *nicht* portabel
- Syntax für logische Pfadnamen:

  $[\langle \textit{host} \rangle \textbf{:}]\ [\textbf{;}]\ \{\langle \textit{directory} \rangle \textbf{;}\}^*\ [\langle \textit{name} \rangle]\ [\textbf{.}\langle \textit{type} \rangle\ [\textbf{.}\langle \textit{version} \rangle]]$

  Beispiel: `prog:src;dir1;dir2;file.ext.5`
- *logical host* „`sys`“ ist für Lisp-Implementierung reserviert; muß nicht existieren!
- Funktionen und Makros, wie `load`, `open`, `with-open-file`, `probe-file`, `compile-file`, usw. können mit logischen Pfadnamen verwendet werden.

Logical Pathnames (Forts.)

Erfragen der Abbildungsvorschrift:

```
> (logical-pathname-translations "sys")
((#l";**;*.*" #p"/usr/local/lisp/lib-allegro/"))
```

Abbildung von logische auf normale Pathnames:

```
> (translate-logical-pathname "sys:hosts.cl")
#p"/usr/local/lisp/lib-allegro/hosts.cl"
```

```
> (translate-logical-pathname #l"sys:hosts.cl")
#p"/usr/local/lisp/lib-allegro/hosts.cl"
```

Beispiele mit Funktion, die Dateinamen verwendet:

```
> (probe-file "sys:hosts.cl")
#p"/usr/local/lisp/lib-allegro/hosts.cl"
```

```
> (probe-file "sys:foo.bar")
nil
```

Eigene Abbildungsvorschrift

Definieren einer eigenen Abbildungsvorschrift für logischen Host „`test`":

```
(setf (logical-pathname-translations "test")
  `(("src;**;*.*" "lispkurs/sources/")
    ("doc;**;*.*" "lispkurs/doc/")
    ("*.*"        "lispkurs/etc/*.*")
    ("**;*.*"     "lispkurs/**/*.*")))
```

Verwendung:

```
> (translate-logical-pathname #l"test:foo.bar")
#p"lispkurs/etc/foo.bar"
```

```
> (translate-logical-pathname #l"test:src;foo.bar")
#p"lispkurs/sources/foo.bar"
```

```
> (translate-logical-pathname
     "test:src;dir1;dir2;file.ext.10")
#p"lispkurs/sources/dir1/dir2/file.ext"
```

Eigene Abbildungsvorschrift (Forts.)

Veränderung einer bestehenden Abbildung:

```
> (translate-logical-pathname
     #l"test:kernel;foo.bar")
#p"lispkurs/kernel/foo.bar"

> (push '("kernel;**;*.*" "lispkurs/main/")
    (logical-pathname-translations "test"))
((#l"kernel;**;*.*" #p"lispkurs/main/")
 # )

> (translate-logical-pathname
     #l"test:kernel;foo.bar")
#p"lispkurs/main/foo.bar"
```

Merke:

- Verwende nur *logical Pathnames* im Programm.
- Benutzer muß dann *nur* die Abbildungsvorschriften für seinen Rechner festlegen/erweitern.
- Abbildungsvorschriften sind *nicht* Teil des Programms.
- Programmierer kann/sollte Beispiel(e) für Abbildungsvorschriften beilegen, die aber (fast immer) angepaßt werden müssen.

Symbole und Packages

Ein Symbol ist ein Datenobjekt, das folgende Komponenten besitzt:

print name	`symbol-name`	Name (String)
value	`symbol-value`	Wert
function	`symbol-function`	Funktionsdef.
property list	`symbol-plist`	Property-Liste
package cell	`symbol-package`	Home-Package oder NIL

Beispiele:

```
> (setq foo 'bar)
BAR
> (defun foo () 'baz)
FOO

> (symbol-name 'foo)
"FOO"
> foo
BAR
> (foo)
BAZ
> #'foo
#<Interpreted Function FOO>
> (symbol-package 'foo)
#<The COMMON-LISP-USER package>
```

Packages – Einführung

Problem: mehrere Lisp-Pakete von verschiedenen Autoren
$\Longrightarrow$ Namenskollisionen, falls gleiche Symbolnamen verwendet werden

Lösung: Jedes Symbol erhält einen Zusatz, sein *package*.

- Package: (Hash-)Tabelle für die Abbildung von Symbol*namen* auf Symbole.
- Mittel zur Organisation des *Namespace* und zum *Information Hiding*.

  Jedes Softwaremodul sollte im eigenen Package existieren.
- Zu jedem Zeitpunkt genau *ein* aktuelles Package.

  Wert der globalen Variable `*package*` gibt *aktuelles* Package an.
- Lisp-Reader `(read ...)` sucht und erzeugt eingelesene Symbole im aktuellen Package.

Erzeugung von Symbolen in einem Package

- Package *besitzt* eine Reihe von Symbolen; diese sind in diesem Package *interned*; es ist ihr *Home-Package*.
- Lisp-Reader `(read ...)` erzeugt eingelesene Symbole im aktuellen Package, d. h diese sind *interned*.
- Mit `(intern ...)` werden neue *interned* Symbole erzeugt.
- Mit `(make-symbol ...)` oder `(gensym)` können *uninterned* Symbole (Symbole ohne Home-Package) erzeugt werden.

```
> (symbol-package 'uvw)
#<The COMMON-LISP-USER package>
```

```
> (setq x (intern "UVW"))
UVW
> (symbol-package x)
#<The COMMON-LISP-USER package>
> (eq x 'uvw)
T
```

```
> (setq y (make-symbol "UVW"))
#:UVW
> (symbol-package y)
NIL
> (eq y 'uvw)
NIL
```

Importieren und Exportieren von Symbolen

- Symbole werden mit der Funktion `(export ...)` zu *externen*, von außen zugreifbaren Symbolen gemacht.
- Alle externen Symbole eines Packages `foo` können mit `(use-package "FOO")` zu internen gemacht werden; die externen Symbole werden damit *ererbt*.
- Einzelne Symbole können mit `(import 'foo:bar)` zu internen gemacht werden.
- Package-Vererbung ist nicht transitiv und auch nicht streng hierarchisch; Package-Strukturen sind schwer zu verstehen.

Referenzierung von Symbolen:

`bar`	internes Symbol im aktuellen Package
`foo:bar`	externes Symbol des Package `foo`
`foo::bar`	nicht-externes Symbol des Package `foo`
`:bar`	Keywords

Wichtige Packages

common-lisp-user (oder **user**, **cl-user**) – das voreingestellte Package, es ererbt alle Symbole von `common-lisp`.

common-lisp (oder **cl**, **lisp**) – enthält alle Symbole für Common-Lisp-Variablen und -Funktionen.

keyword – Symbole beginnen mit `:` und haben sich selbst als Wert.

system – reserviert für implementierungsabhängige Dinge.

```
> :test
:TEST
> (symbol-package ':test)
#<The KEYWORD package>
> (eq :test 'test)
NIL
> (eq :test keyword:test)
T
```

Wichtige Package-Befehle

- `(in-package "utilities")`
- `(use-package "LISP")`
- `(find-package "utilities")`
- `(make-package "utilities" :nicknames '("util"))`
- `(defpackage "utilities" (:nicknames "util")...)`

Beispiel:

```
(in-package "utilities")

(export 'my-append)

(defun my-append (list1 list2)
  (cond ((null list2) list1)
        (t (do-my-append list1 list2))))

(defun do-my-append (list1 list2)
  (cond ((null list1) list2)
        (t (cons (first list1) (do-my-append (rest list1) list2)))))
```

Verwendungsbeispiel

Datei `util-pckg.lisp`**:**

```
(in-package "USER")

(defpackage "UTILITIES"
  (:nicknames "UTILS")
  (:use "LISP" "CLOS")
  (:export "MY-APPEND"))
```

Datei `utilities.lisp`**:**

```
(in-package "UTILS")

(defun my-append ( ... )
  ... )
```

Datei `load-utils.lisp`**:**

```
(in-package "USER")

(load "util-pckg")
(load "utilities")

(use-package "UTILS")
```

Weitere nützliche Package-Befehle

- Erstellen eines Package, falls nicht existent:

```
(unless (find-package "FOO")
  (make-package "FOO" :use "CL"))
```

- Umbenennen eines Package:

```
(rename-package "FOO" "NEW-FOO" '("NICK-FOO"))
```

- Löschen eines Package:

```
(in-package "USER")
(delete-package "FOO")
```

- Liste aller Packages/Beschreibung eines Package:

```
(list-all-packages)
(describe (find-package "CL"))
```

- Suchen nach einem bestimmten Symbol:

```
(find-all-symbols "MY-SYMBOL")
(find-symbol "MY-SYMBOL" "FOO")
```

- Bearbeitung aller Symbole eines Package:

```
(do-symbols (x "FOO") (print x))
(do-external-symbols (x "FOO") (print x))
```

Fehlermeldung `error`

```
(defun quad (x)
  (unless (numberp x)
    (error "Quad mag ~s nicht als Argument" x))
  (* x x))
> (quad 'anna)
Error: Quad mag ANNA nicht als Argument
```

Syntax `(error <formatstring> <args>)`

- Ausgabe geht nach `*error-output*`
- Berechnung wird abgebrochen
- Verhalten wie bei vordefinierten Fehlermeldungen betritt Debugger
- Argumente wie `format`, ohne Angabe des Streams

Fehler abfangen `ignore-errors`

Syntax `(ignore-errors <form>*)`

- Normalfall: Wert des letzten Ausdrucks
- Wenn Fehler durch `error` auftritt: `NIL`.
- Fehler- und Normalfall durch Nebenwert unterscheidbar
- Mißbrauch: `ignore-errors` statt Debugging

`block` **und** `return`

statischer, nichtlokaler Ausstieg

```
(defun fak (x)
  (when (= x 0)(return-from 'fak 1))
  (* x (fak (- x 1))))
```

Syntax:
```
(block <name> <form*>)
(return-from <name> <value>)
(return <value>)
```

- `block` erzeugt Block
  `defun`: implizit Block (Name: Funktionsname)
  Iterationskonstrukte: Block mit Name `nil`
- `return-from` springt aus Block mit geg. Name heraus
  `return` springt aus Blocks mit Namen `nil`.
- Wert des Blocks (Funktionsaufruf, Iteration): `<value>`
- Statischer Ausstieg:
  `return` muß lexikalisch im Block stehen.
- Nichtlokaler Ausstieg:
  Sprung aus Schachtelung von Blocks heraus möglich.
- Lokales lexikalisches Goto: `tagbody` und `go`
- Zweck lexikalischer Sprünge:
  Bau neuer Sprachelemente.
  Andernfalls erschwert es Lesbarkeit des Codes

`catch` und `throw`

dynamischer, nichtlokaler Ausstieg

```
(catch 'passt-nicht
  (schraube-ein dies)
  (mapcar #'baue-ein teile))
```

Tief unten in der Aufrufhierarchie:

```
(if (zu-gross-p teil)(throw 'passt-nicht ()))
```

Syntax:
```
(catch <name> <form*>)
(throw <name> <resultform>)
```

- dynamischer Ausstieg: solange Rumpf von `catch` läuft:
  Sprung von beliebiger Codestelle aus mit `throw` möglich
- gleichnamiger `catch`-Ausdruck terminiert dann sofort
  Ergebnis: 2. Argument von `throw`
- Ohne Sprung:
  Wert von `catch` Wert des letzten Ausdrucks.
- `throw` ohne gleichnamiges `catch` ist Fehler
- Zweck: vereinfachter Abbruch der Berechnung in Ausnahmesituationen
- Bei Gebrauch in Normalfällen:
  Erschwert Verständnis des Codes sehr.

Integrität sichern: `unwind-protect`

Aufräumarbeiten bei Fehlerabbruch

```
(unwind-protect
   (progn (motor-an)
          (bohrmaschine-ab)
          (bohrmaschine-auf))
   (motor-aus))
```

Syntax: `(unwind-protect <prot-form><cleanup-form>)`

- `<prot-form>` wird evaluiert
- `<prot-form>` endet z.B.:
  - normal
  - durch Fehler mit `error`
  - durch `catch` und `throw`
- `<cleanup-form>` wird **in jedem Fall** ausgeführt.
- Wert: Wert von `<prot-form>`
- Fehlerabbruch (`error`):
  Aufräumen erst bei Verlassen des Debuggers.

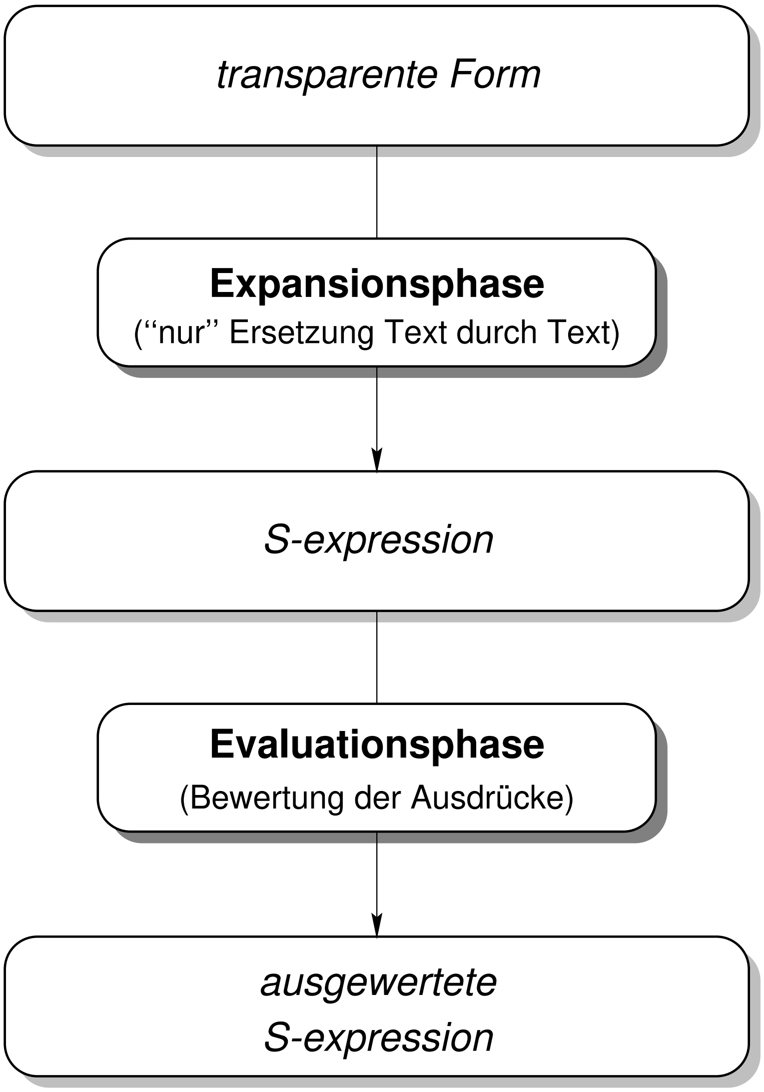
Teil 7

Makros

- Makros und Backquote
- Anwenden von Makros
- Abarbeitung von Makros

Makros – Überblick

Der Begriff „Makro“ wurde zuerst in der Assemblerprogrammierung eingesetzt. Unter einem Makro versteht man im allgemeinen einen Text (ein Befehl), der in einen anderen Text (mehrere Befehle) expandiert.



Verwendungsarten von Makros

- ADT-Schnittstellenmakros.

  Einfache Zugriffe auf ADT ohne Effizienzverlust.

- Code einhüllen: Bsp. `with-open-file`

  Makros als Abkürzung für oft verwendete Codestücke,

  zum Einhalten von Konventionen für sicheren Code . . .

- Implementierung von Kontrollstrukturen und Zuweisungen.

  Hier dürfen die Argumente nicht evaluiert werden, Makro notwendig.

- Erstellen einer Spezialsprache mit eigener Syntax.

  Möglichkeiten der Makro-Lambdaliste wertvoll.

Stilregeln:

- Expansionsphase sollte möglichst wenig tun,

  keine Seiteneffekte!

- Argumente, auf die die Expansionsphase zugreift, müssen im Makroaufruf Konstanten sein.

- Beide Phasen rufen Hilfs*funktionen* – nicht Hilfsmakros.

- Normalerweise braucht man `eval` nicht.

Makros – Einführendes Beispiel

Beispiel: Zugriffsfunktionen auf Attribute von Objekten aus der *blocks world*, jedes Objekt wird als *Liste* von Attributwerten repräsentiert.

```
> (setq objekt1 '(wuerfel-A wuerfel 8 rot))
(WUERFEL-A WUERFEL 8 ROT)
```

Zugriff auf Typ:

```
> (cadr objekt1)
WUERFEL
```

Zugriff auf Farbe:

```
> (cadddr objekt1)
ROT
```

Nachteil: Allein aus dem Code (hier: Funktion `cadr` bzw. `cadddr`) ist nicht zu erkennen, daß diese Zugriffe den Typ bzw. die Farbe des Objektes als Ergebnis hat.

Makros – Einführendes Beispiel (Forts.)

Anstatt einer Repräsentation eines Objekts als Liste von Attributwerten können *Properties* eines Symbols verwendet werden.

```
> (prog1
    (setf (get 'objekt1 'name) 'wuerfel-A)
    (setf (get 'objekt1 'typ) 'wuerfel)
    (setf (get 'objekt1 'laenge) 8)
    (setf (get 'objekt1 'farbe) 'rot))
WUERFEL-A

> (symbol-plist 'objekt1)
(FARBE ROT LAENGE 8 TYP WUERFEL NAME WUERFEL-A)
```

Zugriff auf Farbe:

```
> (get 'objekt1 'farbe)
ROT
```

Vorteil: Leicht erkennbar, auf welches Attribut eines Objektes zugegriffen wird.

Nachteil: Wenn die Repräsentation der Daten nochmals geändert wird, müssen **alle** Stellen im Code gefunden und angepaßt werden, wo auf die Objektattribute zugegriffen wird.

Makros – Einführendes Beispiel (Forts.)

Lösung: Festlegung einer Schnittstelle, Zugriff auf Attribute erfolgt über *Funktionen*.

Funktion für Repräsentation als Liste:

```
> (defun get-farbe (objekt)
    (cadddr objekt))
GET-FARBE

> (get-farbe objekt1)
ROT
```

Funktion für Repräsentation durch Symbol-Properties:

```
> (defun get-farbe (objekt)
    (get objekt 'farbe))
GET-FARBE

> (get-farbe 'objekt1)
ROT
```

Nachteile: Zusätzlicher Overhead für den Funktionsaufruf vermindert Effizienz.

Argument einer Funktion wird *immer* ausgewertet, auch wenn man Auswertung vermeiden will.

Makros – Schnittstelle mittels Funktionen

Funktionen haben u. a. folgende *Nachteile*:

Aufruf-Overhead: Jeder zusätzliche Funktionsaufruf führt zum Auswerten der Argumente, zum Abspeichern der berechneten Argumentwerte, zum Binden dieser Argumentwerte an die Parameter der Funktion und schließlich zum Funktionsaufruf mit diesen Bindungen.

Neben diesem Overhead ist auch zu bedenken, daß Funktionsaufrufe meist vom Compiler nicht optimiert (z. B. als Inline-Code) werden können.

Festgelegte Reihenfolge der Argumentauswertung: Wie man an dem gezeigten Beispiel sieht, muß für den Zugriff auf die Listenrepräsentation das Argument `objekt1` ausgewertet werden, um die Liste zu erhalten. Bei der Property-Repräsentation dagegen darf das Argument `objekt1` nicht ausgewertet werden, da man ja nicht auf den Wert des Symbols, sondern auf das Symbol und dessen Properties zugreifen will. Daher muß man im zweiten Fall das Argument mit einem Quote versehen.

Will man eine einheitliche Aufrufschnittstelle, die auch bei einem Wechsel der internen Repräsentation nicht geändert werden muß, so müßte man hier also festlegen, daß das Argument der Funktion `get-farbe` immer mit einem Quote versehen werden muß. (Für die Listenrepräsentation müßte man dann die Auswertung des Arguments wieder erzwingen.)

Makros – einführendes Beispiel (Forts.)

Lösung: Festlegung einer Schnittstelle, Zugriff auf Attribute erfolgt über *Makros*.

Makro für Repräsentation als Liste:

```
> (defmacro get-farbe (objekt)
    (list 'cadddr objekt))
GET-FARBE

> (get-farbe objekt1)
ROT
```

Makro für Repräsentation durch Symbol-Properties:

```
> (defmacro get-farbe (objekt)
    (list 'get
          (list 'quote objekt)
          ''farbe))
GET-FARBE

> (get-farbe objekt1)
ROT
```

Makros – Schnittstelle mittels Makros

Eine Makrodefinition hat eine ähnliche Struktur wie eine Funktionsdefinition. Auf den ersten Blick unterscheiden sich beide nur durch die Verwendung von `defmacro` statt `defun`; die Angabe der Makroparameter gleicht denen für Funktionsparameter.

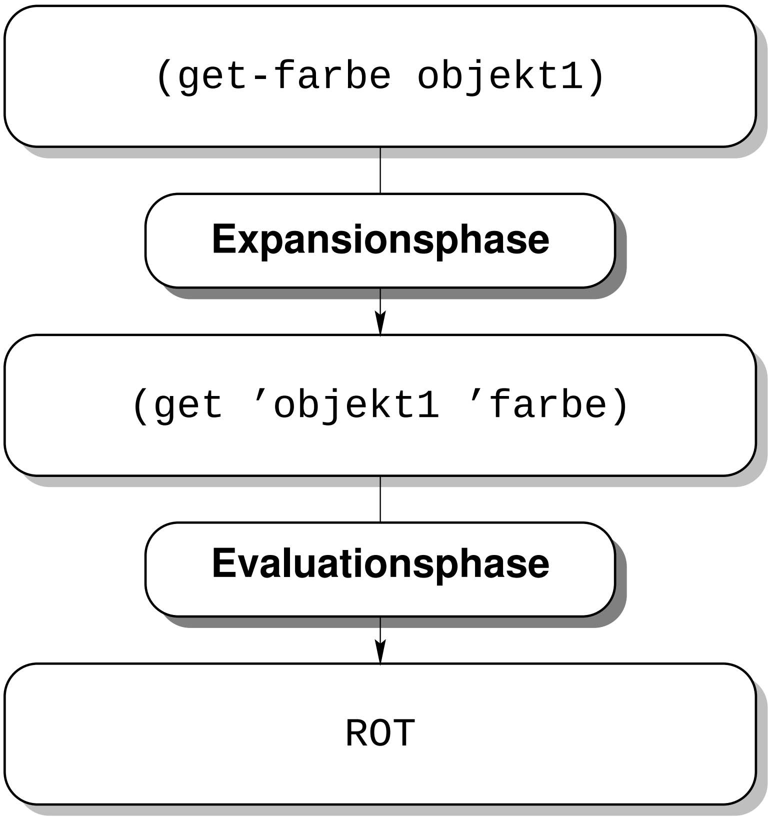
Makros und Funktionen unterscheiden sich aber darin, daß der berechnete *Funktionswert eines Makros* ein Stück Lisp-Code darstellt, das nach der Makroexpansion vom Interpreter wie ein gewöhnliches Programmstück abgearbeitet wird. (Im Unterschied dazu wird der Wert einer Funktion anschließend vom Code des Programms weiter verarbeitet, der Funktionswert ist also nicht Teil des Programms.)

Außerdem werden die *Argumente eines Makros* bei der Makroexpansion *nicht ausgewertet*. Dadurch kann man Makros so definieren, daß sich die Aufrufe, wie man hier auch sehen kann, nicht unterscheiden – unabhängig von der gewählten Repräsentation. In beiden Fällen muß das Argument `objekt1` *nicht* mit einem Quote versehen werden. Die Entscheidung, ob das Argument nun ausgewertet wird oder nicht, fällt erst durch das expandierte Stück Code.

Makros – Expansion

Makros werden in **zwei Stufen** evaluiert:

1. Expansion:
   - Die Argumente des Makros werden gelesen, aber **nicht** evaluiert.
   - Der Körper des Makros wird **expandiert**, das Resultat ist ein Stück Code mit den gelesenen Makroargumenten.
2. Dieses generierte Stück Code wird **evaluiert**.



Makros – Expansion

Unterschiede Funktions- und Makroauswertung:

Ein Funktionsaufruf erfolgt in folgenden Schritten:

1. Holen der Funktionsdefinition ,
2. Evaluierung aller Argumente von links nach rechts,
3. Aufruf der Funktion, dabei Binden der Funktionsparameter an die evaluierten Argumente,
4. Rückgabe/Ergebnis ist der Funktionswert.

Ein Makroaufruf erfolgt in folgenden Schritten:

1. Holen der Makrodefinition,
2. Lesen der Argumente *ohne* Evaluierung,
3. Expansion des Makros, dabei Binden der Makroparameter an die *nicht* evaluierten Argumente,
   (a) Körper des Makros wird evaluiert, also ausgeführt,
   (b) Ergebnis der Evaluierung des Makros ist ein beliebiges Stück Lisp-Code,
4. Generiertes Stück Lisp-Code wird wie üblich evaluiert.

Wichtig:

- Ergebnis des Makros ist Lisp-Code.
- Im Körper des Makros können beliebige Anweisungen und Manipulationen (z. B. der Argumente) erfolgen.
- Die Argumente eines Makros werden *nicht* evaluiert (im Gegensatz zu den Argumenten einer Funktion).

Makros – Expansion an einem Beispiel

Gegeben ist folgendes Makro:

```
(defmacro get-farbe (objekt)
  (list 'get
        (list 'quote objekt)
        ''farbe))
```

Aufruf des Makros `get-farbe`**:**

```
(get-farbe objekt1)
```

1. Lese das Argument „`objekt1`" für den notwendigen Parameter `objekt`.
2. Evaluiere den Makrokörper mit diesem Argument:

   $\texttt{(list 'get (list 'quote } \overbrace{\texttt{objekt}}^{\texttt{objekt1}} \texttt{) ''farbe)}$

3. Damit erhält man die Expansion des Makros:

   ```
   (get 'objekt1 'farbe)
   ```

4. Das erhaltene Codestück wird anschließend evaluiert:

   ```
   ROT
   ```

Makros – Explizite Makroexpansion

Der Expansionszwischenschritt ist nicht sichtbar, man kann aber mit `macroexpand` (bzw. `macroexpand-1`) die expandierte Form betrachten:

```
> (macroexpand '(get-farbe objekt1))
(GET 'OBJEKT1 'FARBE)
T
```

```
> (macroexpand-1 '(get-farbe objekt1))
(GET 'OBJEKT1 'FARBE)
T
```

Hinweise:

- `macroexpand` wird vom Lisp-Interpreter für die *Expansionsphase* benutzt.
- `macroexpand` hat zwei Ergebnisse: die expandierte Form und ein Flag, ob eine Expansion stattfand.
- `macroexpand-1` führt genau einen Expansionsschritt aus, `macroexpand` expandiert immer vollständig.

Makros – Explizite Makroexpansion (Forts.)

Die Lisp-Funktion `pop` ist als Makro implementiert:

```
> (macroexpand '(pop my-stack))
(PROG1 (CAR MY-STACK)
       (SETQ MY-STACK (CDR MY-STACK)))
T
```

Die Definition eines eigenen Makros `my-pop` könnte lauten:

```
> (defmacro my-pop (stack)
    (list 'prog1
          (list 'car stack)
          (list 'setq stack
                      (list 'cdr stack))))
MY-POP
```

```
> (macroexpand '(my-pop my-stack))
(PROG1 (CAR MY-STACK)
       (SETQ MY-STACK (CDR MY-STACK)))
T
```

`my-pop` kann wie `pop` verwendet werden:

```
> (setq my-stack '(a b c))
(A B C)
> (my-pop my-stack)
A
> my-stack
(B C)
```

Tips zum Erstellen von Makros

1. Ein Beispiel für die Verwendung aufschreiben.
2. Den Lisp-Code angeben, den das Makro produzieren soll.
3. Ein Makro schreiben, das Ersteres in Letzteres überführt.

   Hierzu mit den Funktionen `list`, `cons`, `append` die Listenstruktur des Lisp-Codes aufbauen, dabei alle unveränderlichen Symbole mit einem Quote versehen, die Makroparameter ohne Quote lassen, falls sie evaluiert werden sollen.
4. Mit `macroexpand` und Beispiel prüfen.

Beispiel:

1. `(my-push elem stack)`
2. `(setq stack (cons elem stack))`
3. ```
   (defmacro my-push (elem stack)
     (list 'setq
           stack
           (list 'cons elem stack)))
   ```
   ```

Tips zum Erstellen von Makros (Forts.)

Beispiel:

1. `(my-push elem stack)`

2. `(setq stack (cons elem stack))`

3.
```
(defmacro my-push (elem stack)
  (list 'setq
        stack
        (list 'cons elem stack)))
```

Überprüfen:

```
> (macroexpand '(my-push my-elem my-stack))
(SETQ MY-STACK (CONS MY-ELEM MY-STACK))
T

> my-stack
(B C)

> (my-push 'a my-stack)
(A B C)

> my-stack
(A B C)
```

Makros – Schlüsselwort `&body`

- `&body` ist identisch mit `&rest`.
- `&body` steht für den „Rumpf"/„Body" der einzusetzenden Lisp-Form (Rumpf: beliebig viele Ausdrücke)
- `&body` kann *nur* mit `defmacro` verwendet werden.
- `&body` erhöht die Lesbarkeit der Makrodefinition.

Beispiel: `dolist`

```
(defun print-list (l)
  (dolist (var l)
    (print '-)
    (prin1 var)))
```

Das Makro `dolist` hat ein notwendiges Argument, anschließend können *beliebig viele* weitere Argumente folgen.

Allgemeine Form von `dolist`:

```
(dolist (var listform result)
   exp1
   exp2
   ...)
```

Makros – Schlüsselwort `&body` (Forts.)

Beispiel: Definition von `dolist` mit `defmacro`

```
(dolist (var listform result)
   exp1 exp2 ...)
```

soll expandieren zu:

```
(do* ((ltemp listform    (cdr ltemp))
      (var   (car ltemp) (car ltemp)))
     ((null ltemp) result)
  exp1 exp2 ...)
```

Definition des Makros:

```
(defmacro my-dolist (parg &body exp)
  (append
    (list 'do*
          (list (list 'ltemp
                      (second parg)
                      '(cdr ltemp))
                (list (first parg)
                      '(car ltemp)
                      '(car ltemp)))
          (list '(null ltemp)
                (third parg)))
    exp))
```

Makros – Schlüsselwort `&body` (Forts.)

Definition des Makros:

```
(defmacro my-dolist (parg &body exp)
  (append
    (list 'do*
          (list (list 'ltemp
                      (second parg)
                      '(cdr ltemp))
                (list (first parg)
                      '(car ltemp)
                      '(car ltemp)))
          (list '(null ltemp)
                (third parg)))
    exp))
```

Vereinfachung durch spezielle Makro-Parameterangabe:

```
(defmacro my-dolist
          ((val listform &optional result) &body exp)
  (append
    (list 'do*
          (list (list 'ltemp
                      listform
                      '(cdr ltemp))
                (list val
                      '(car ltemp)
                      '(car ltemp)))
          (list '(null ltemp) result))
    exp))
```

Makros – Vor- und Nachteile

+ Makros sind ein gutes Mittel zur Datenabstraktion:
  - Implementierung kann durch Makros „versteckt“ werden (Schnittstelle!).
  - Beibehalten der Schnittstelle auch bei Änderung der darunterliegenden Implementierung möglich.

– Programmanalyse wird durch Makros erschwert:

  Abhängigkeiten sind beim Ladevorgang zu beachten.

+ Makros ermöglichen die Definition einer auf Lisp aufgesetzten Sprache:
  - Struktur und Bedeutung der Parameter/Argumente kann beliebig festgelegt werden.
  - Argumente werden bei der Expansion nicht evaluiert, so daß Evaluierung und Evaluierungsreihenfolge frei festgelegt werden kann.

– Makros haben üblicherweise eine komplexe Parameterstruktur.

Lisp-Funktionen: einige spezielle Funktionen sind Makros:

`defun`, `prog`, `cond`, `do`, `setf`, `defmacro`, `...`

```
> (macroexpand '(incf n 2))
(SETQ N (+ N 2))
T
```

Makros – Backquote

Nach dem Backquote-Zeichen werden Ausdrücke wie beim Quote-Zeichen **nicht** ausgewertet, es sei denn, vor ihnen steht ein Komma:

```
> `(a b c)
(A B C)

> (setq l '(x y z))
(X Y Z)

> `(a ,l c)
(A (X Y Z) C)

> `(a ,@l c)
(A X Y Z C)
```

Komma `,` (unquote) und Komma-At `,@` (unquote splice) können an jeder Stelle *innerhalb* eines „Backquote-Ausdruckes“ stehen:

```
> `(a (b (c (d ,l e) f) g) h)
(A (B (C (D (X Y Z) E) F) G) H)
```

Makros – Backquote (Forts.)

Das Backquote-Zeichen ‘ ist mit einer Funktion assoziiert, die Code generiert, der bei Bewertung die endgültige Liste erzeugt:

```
> ‘(a ,l c)
(A (X Y Z) C)

> (list ’a l ’c))
(A (X Y Z) C)

> ‘’(b (c (d ,l e) f) g)
’(B (C (D (X Y Z) E) F) G)

> (list ’quote
        (list ’b (list ’c (list ’d l ’e) ’f) ’g))
’(B (C (D (X Y Z) E) F) G)
```

Nach einem Komma `,` oder Komma-At `,@` steht ein beliebiger Lisp-Ausdruck, der bewertet wird:

```
> ‘(a ,(first l) c ,@(rest l)
    d ,(cons ’e ‘(f ,l g)) h)
(A X C Y Z D (E F (X Y Z) G) H)
```

Makros – Beispiel für Backquote

Das vorgestellte Beispiel des Makros `my-push`:

1. Verwendung:

   ```
   (my-push elem stack)
   ```

2. Zielform:

   ```
   (setq stack (cons elem stack))
   ```

3. Makro ohne Backquote:

   ```
   (defmacro my-push (elem stack)
     (list 'setq
           stack
           (list 'cons elem stack)))
   ```

Vereinfachung durch Verwendung eines Backquote:

```
(defmacro my-push (elem stack)
  `(setq ,stack (cons ,elem ,stack)))
```

- Ähnlichkeit von Makrodefinition und Zielcode ist viel offensichtlicher.
- Grundmuster mit Backquote übersteht Expansion ohne Auswertung, an Stellen mit , werden Argumente eingesetzt.

Macros – Beispiel für Backquote (Forts.)

Das vorgestellte Makro `my-dolist`:

```
(defmacro my-dolist
        ((val listform &optional result) &body exp)
  (append
    (list 'do*
          (list (list 'ltemp
                      listform
                      '(rest ltemp))
                (list val
                      '(first ltemp)
                      '(first ltemp)))
          (list '(null ltemp) result))
    exp))
```

Zur Erinnerung: Zielform ist

```
(do* ((ltemp listform      (rest ltemp))
      (var   (first ltemp) (first ltemp)))
     ((null ltemp) result)
  exp1 exp2 ...)
```

Vereinfachung durch Verwendung eines Backquote:

```
(defmacro my-dolist
        ((val listform &optional result) &body exp)
  `(do* ((ltemp ,listform      (rest ltemp))
         (,val   (first ltemp) (first ltemp)))
        ((null ltemp) ,result)
      ,@exp))
```

Makros – Funktion `gensym`

```
(defmacro my-dolist
        ((val listform &optional result) &body exp)
  `(do* ((ltemp ,listform      (rest ltemp))
         (,val   (first ltemp) (first ltemp)))
       ((null ltemp) ,result)
     ,@exp))
```

Problem: Der Benutzer von `my-dolist` könnte ebenso ein Symbol namens `ltemp` verwenden und innerhalb des Schleifenkörpers `exp` darauf zugreifen wollen. Da wir dieses Symbol lokal mit einem anderen Wert gebunden haben, erhält er damit falsche Werte.

Bsp.: `(my-dolist (x '(1 2 3)) (print (setq ltemp x)))`

Abhilfe: Mit der Funktion `gensym` ein neues, noch nicht verwendetes Symbol erzeugen und statt `ltemp` verwenden.

```
(defmacro my-dolist
        ((val listform &optional result) &body exp)
  (let ((ltemp (gensym)))
    `(do* ((,ltemp ,listform      (rest ,ltemp))
           (,var    (first ,ltemp) (first ,ltemp)))
         ((null ,ltemp) ,result)
       ,@exp)))
```

Makros – Funktion `gensym` (Forts.)

Gesucht: Makro `dostream` für Streams (analog zu `dolist` für Listen).

Stream wird elementweise bis zum Ende gelesen.
Rumpf wird mit entsprechend gebundener Elementvariable ausgeführt.

Syntax: `(dostream (`⟨*var*⟩ ⟨*stream*⟩ `:line` ⟨*nil/t*⟩`)` ⟨*body*⟩`)`

```
(defmacro dostream ((elem stream &key line) &body body)
  (let ((eof (gensym)))
    `(do ((,elem (,(if line 'read-line 'read) ,stream nil ',eof)
                 (,(if line 'read-line 'read) ,stream nil ',eof)))
         ((eql ,elem ',eof) t)
         ,@body)))
```

- Anwendungsbeispiel:

```
(with-open-file (str "foo.lisp" :direction :input)
   (dostream (x str) (print x)))
```

- Wenn Datei keine korrekten Lisp-Objekte enthält, sollte `:line t` angegeben werden, da dann statt `read` komplette Zeilen mit `read-line` gelesen werden.

Makros – Beispiel: Memoizing function values

Eingabe: Funktion, ohne Seiteneffekte

z. B. Reihen, wie Fibonacci, Primzahlen, ...

Ergebnis: Bereits errechnete Funktionswerte sollen in zukünftigen Aufrufen der Funktion ohne Neuberechnung verwendet werden.

$\Longrightarrow$ Aufbau einer Tabelle (Bsp: Fibonacci-Reihe)

Argumentliste $(a_1\ a_2\ \ldots\ a_n)$	Funktionswert f
(1)	1
(2)	2
(3)	3
(4)	5
(5)	8

Idee 1: Tabelle als A-Liste:

```
(((1) . 1) ((2) . 2) ((3) . 3))
```

Idee 2: Tabelle in der Property-Liste des Funktionssymbols ablegen:

```
(setf (get 'func 'calls) a-liste)
(push (cons args func-value)
      (get 'func 'calls))
```

Makros – Beispiel: Memoizing function values

Die Funktion `rec-fib` erzeugt rekursiv die Fibonacci-Zahlenfolge:

```
(defun rec-fib (n)
  (if (<= n 1)
      1
    (+ (rec-fib (- n 1))
       (rec-fib (- n 2)))))
```

Um auf bereits errechnete Funktionswerte zugreifen zu können, muß die Funktion `rec-fib` erweitert werden:

```
(defun rec-fib (n)
  (let ((rec-value (assoc (list n)
                          (get 'rec-fib 'calls)
                          :test #'equal)))
    (if rec-value
        (cdr rec-value)
      (cdr (first
            (push (cons (list n)
                        (if (<= n 1)
                            1
                          (+ (rec-fib (- n 1))
                             (rec-fib (- n 2)))))
                  (get 'rec-fib 'calls)))))))
```

Makros – Beispiel: Memoizing function values

Funktionen, für die man die Funktionswerte abspeichern will, sollen mit Hilfe des Makros `memo` deklariert werden:

```
> (defun rec-fib (n)...)

> (time (rec-fib 20))
; cpu time (total)  1,670 msec user, 170 msec system
; space allocation: 284,578 cons cells,;  0 symbols, 875,680 other bytes
10946

> (memo rec-fib)
REC-FIB

> (time (rec-fib 20))
; cpu time (total)  60 msec user, 0 msec system
; space allocation:  2,433 cons cells,;  84 symbols, 3,376 other bytes
10946

> (time (rec-fib 20))
; cpu time (total)  10 msec user, 0 msec system
; space allocation:  18 cons cells,;  0 symbols, 80 other bytes
10946

> (get 'rec-fib 'calls)
(((20) . 10946) ((19) . 6765) ((18) . 4181) ((17) . 2584) ((16) . 1597)
 ((15) . 987) ((14) . 610) ((13) . 377) ((12) . 233) ((11) . 144) ...)
```

Makros – Beispiel: Memoizing function values

```
(defmacro memo (fn)
  (let* ((fntype (and (fboundp fn)
                      (functionp (symbol-function fn))))
         (fnexpr (when fntype (symbol-function fn))))
    (if fntype
        (if (get fn 'original-function)
            (error "memo: Function already memoized")
          `(progn
             ;; alte Funktion sichern
             (setf (get ',fn 'original-function) ',fnexpr)
             ;; neue Funktion mit "Wrapper" zuweisen
             (setf (symbol-function ',fn)
                 #'(lambda (&rest args)
                     (let ((rec-value (assoc args
                                             (get ',fn 'calls)
                                             :test #'equal)))
                       (if rec-value
                           (cdr rec-value)
                         (cdr
                          (first
                           (push (cons args
                                       (apply ',fnexpr args))
                                 (get ',fn 'calls))))))))
             ',fn))
      (error "memo: Function undefined or of illegal type"))))
```

Makros – Beispiel: Memoizing function values

```
> (macroexpand '(memo rec-fib))

(progn
  (setf (get 'rec-fib 'original-function)
        '#<Interpreted Function REC-FIB>)
  (setf (symbol-function 'rec-fib)
        #'(lambda (&rest args)
            (let ((rec-value
                    (assoc args (get 'rec-fib 'calls) :test #'equal)))
              (if rec-value
                  (cdr rec-value)
                (cdr
                 (first
                  (push (cons args
                              (apply '#<Interpreted Function REC-FIB>
                                     args))
                        (get 'rec-fib 'calls))))))))
  'rec-fib)
```

Teil 8

CLOS – Common Lisp Object System

- Objektorientierte Programmierung
- Grundbegriffe von CLOS
  - Klassendefinition
  - Instanzenerzeugung
  - Generische Funktionen und Methoden
- Weiterführende Konzepte
  - Integration von Klassen in das Lisp-Typsystem
  - Besonderheiten der Vererbung
  - Generic Dispatch
    Before-, After-, Around- und Primary-Methoden
  - Initialisierung von Instanzen

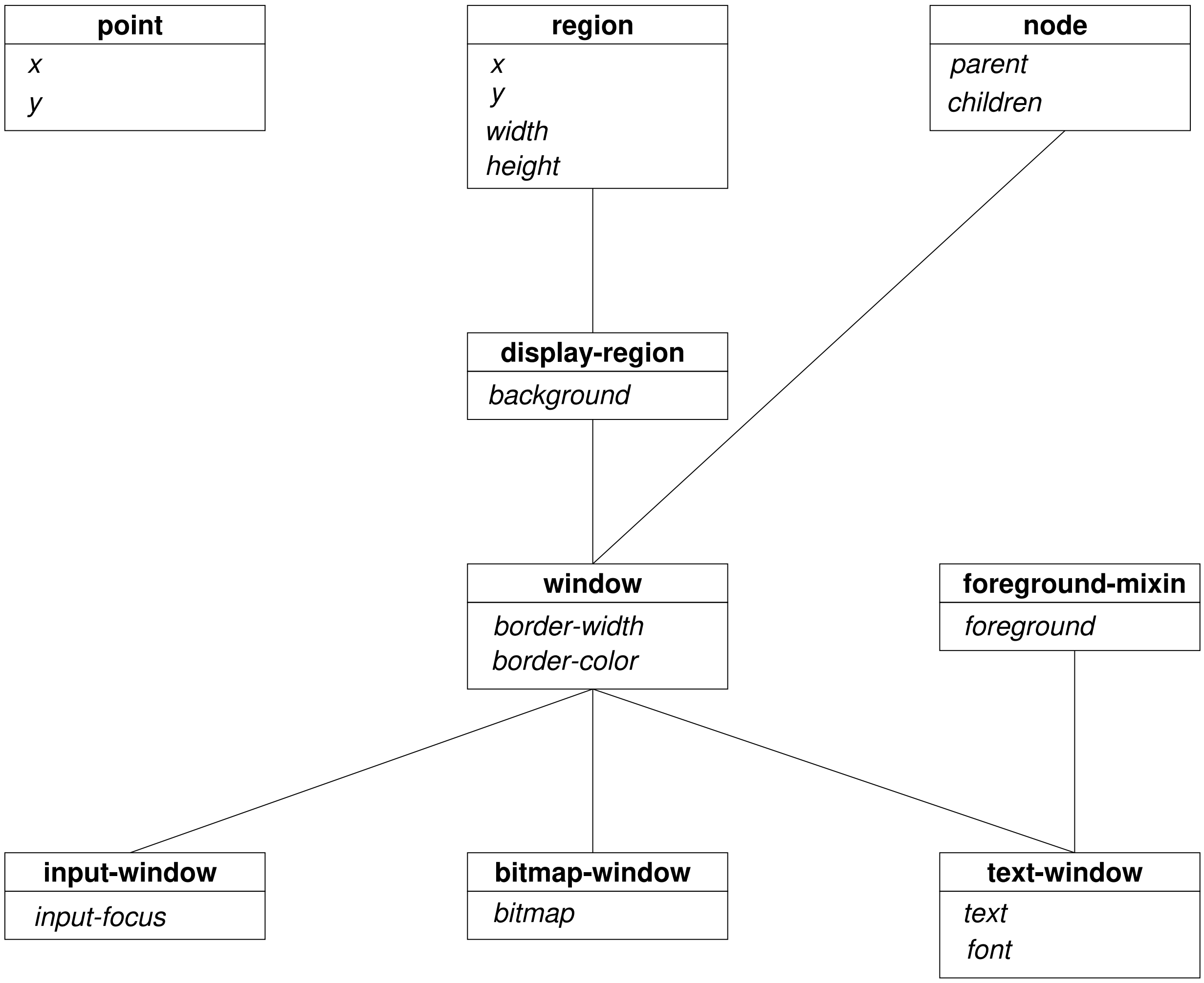
Objektorientierte Programmierung

- Programmieren mit *Objekten*:
  - Erzeugen und Ändern von Objekten
  - Objekte haben assoziertes Verhalten
- Zentrale Begriffe:
  - Klassen
  - Instanzen
  - Slots
  - Methoden
  - Nachrichten/Generische Funktionen
  - Vererbung
- Eigenschaften objektorientierter Programme:
  - Modularität
  - Erweiterbarkeit
  - Hohes Abstraktionsniveau
- CLOS:
  - **C**ommon **L**isp **O**bject **S**ystem
    Object-Oriented Programming in Common-Lisp
  - Vom X3J13 Kommitee in den ANSI Standard
    Common-Lisp aufgenommen.
    In CLtL2 dokumentiert.

Classes and Objects

- Each *object* is an *instance* of a *class*.
- Classes determine the properties (structure and behavior) of their instances.
  - *Structure:* Attributes described by *slots*.
  - *Behavior:* Operations described by *methods*.
- The slots of instances are filled with (individual) values.
- Methods are invoked by *messages* or *generic functions*.
- Classes are organized into an *inheritance hierarchy*, which may be represented by a *directed acyclic graph*.
- In this hierarchy, each class has a number of *direct superclasses* and *direct subclasses*.
  - *Single inheritance:* At most one direct superclass.
  - *Multiple inheritance:* (CLOS) Arbitrary number of direct superclasses.
- The *superclasses* (*subclasses*) of a class are its direct superclasses (subclasses) and their superclasses (subclasses).
- Each class inherits properties (structure and behavior) of its superclasses and spezializes them:
  1. by adding properties,
  2. by specializing/overriding existing properties.

Class Hierarchy (An Example)



Superclass/Subclass Relations

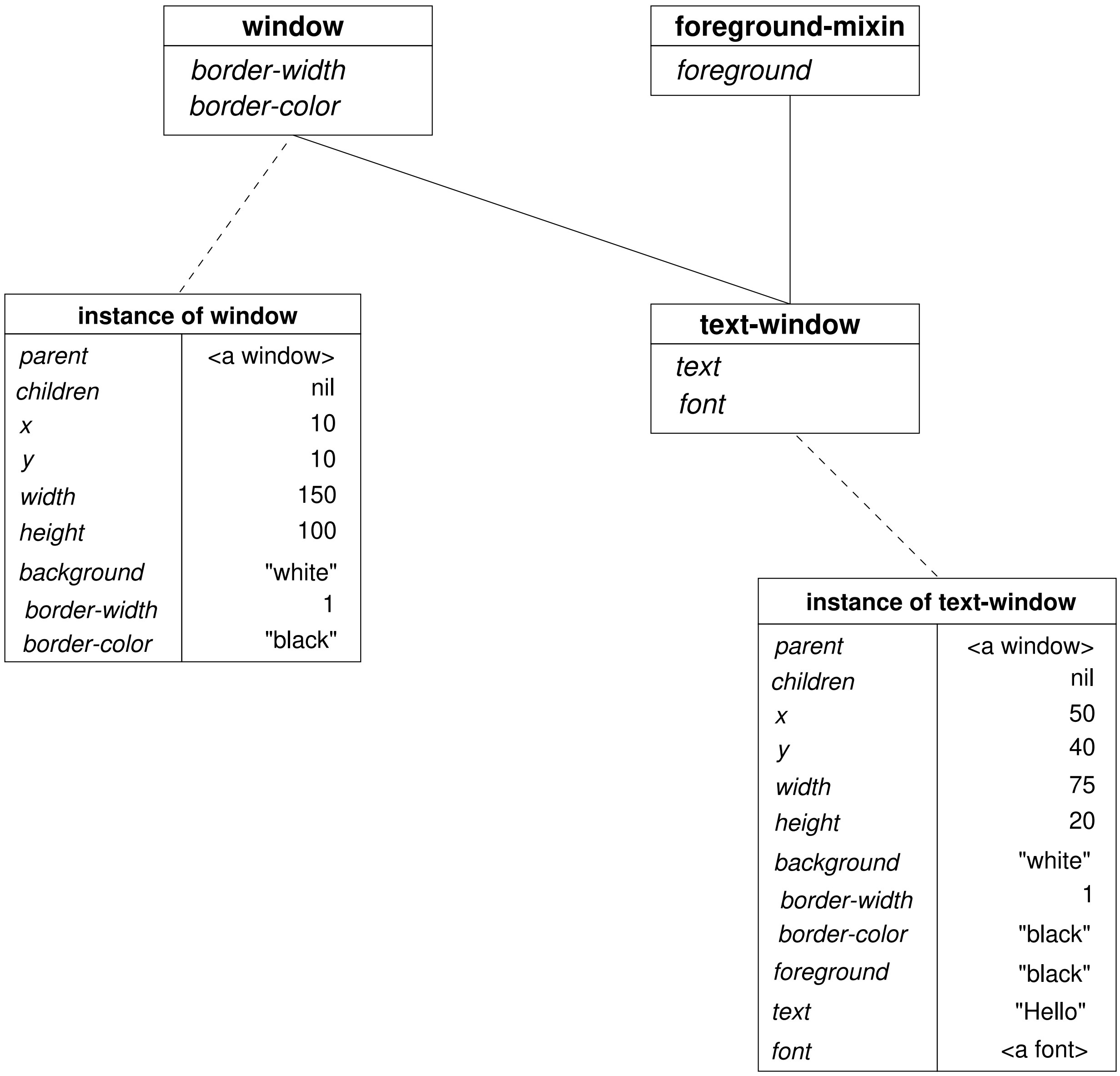
- The class `window` has the direct superclasses `display-region` and `node`.
- The class `region` has the direct subclass `display-region`.
- The class `region` is a superclass of `window`.
- The class `text-window` is a subclass of `node`.

Slots

- The class `window` describes the slots `border-width` and `border-color`.

  It inherits the slots `x`, `y`, `width`, `height`, `background`, `parent`, and `children` from its superclasses.

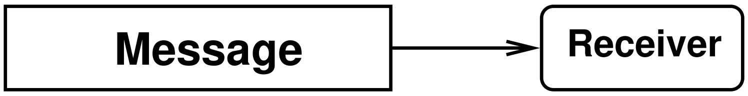
Classes and Instances (An Example)



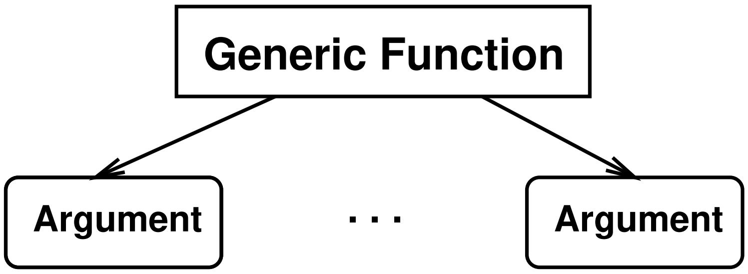
Generic Functions and Methods

- Methods are invoked by

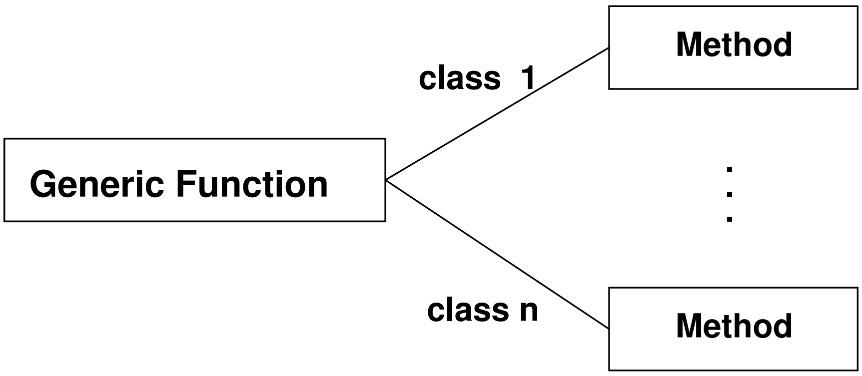
  1. **messages:** The class of **one** recipient object determines, which method is appropriate.



  2. **generic function:** (CLOS) The class**(es)** of the argument object**(s)** determine, which method is appropriate.



- The methods of a message/generic function implement the (class-specific) behavior.



- Generic functions return values, which are computed by the invoked methods.

Defining Classes

Classes are defined by `defclass`.

Examples:

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

(defclass region ()
   ((x :initarg :x :accessor x-pos)
    (y :initarg :y :accessor y-pos)
    (width :initarg :width :accessor width)
    (height :initarg :height :accessor height)))
```

A class definition includes the following items:

- Name of the new class.
- List of direct superclasses.
- Set of *slot specifiers*.
- Set of *class options*:
  - `:default-initargs` default values for initargs (specified by `:initarg` slot options).
  - `:documentation` documentation string for class.

Slot Specifiers

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

A *slot specifier* has one of the following forms:

- A symbol denoting the *slot name* or
- a list containing the *slot name* followed by (zero or more) *slot options*:
  - `:reader`, `:writer`, and `:accessor` automatically generate *slot accessors* for reading and/or writing slot values.
  - `:initarg` defines an initialization argument (initarg) for the slot. It is an argument of the instance constructor `make-instance`.
  - `:initform` provides a default initial value form for the slot if not initialized.
  - `:allocation` controls whether the slot is *local* or *shared*.
  - `:documentation` is a documentation string for the slot.

Class Definition Example

```
(defclass node ()
   ((parent :initarg :parent
            :reader parent)
    (children :initform nil
              :reader children))
   (:documentation "Node in a hierarchy"))

(defclass display-region (region)
   ((background :initarg :background
                :accessor background))
   (:default-initargs :background "white")
   (:documentation "Rectangular region on screen
                    filled with a background-color"))
```

For the `:background` initarg, a default value is given in order to initialize display-regions with a white background, if not specified otherwise.

Example:

By means of the classes `display-region` and `node`, a class for window objects may be defined:

```
(defclass window (display-region node)
   ((border-width :initarg :border-width
                  :accessor border-width)
    (border-color :initarg :border-color
                  :accessor border-color))
   (:default-initargs
      :parent *root-window*
      :x 0
      :y 0
      :border-width 1
      :border-color "black"))
```

- Window objects are display-regions organized in a window hierarchy. They may be framed by a border, specified by the slots `border-width` and `border-color`.
- For the initargs `:parent`, `:x-pos`, `:y-pos`, `:border-width`, and `:border-color` default values are specified.

  `*root-window*` is assumed to be a global variable pointing to the root window of the window hierarchy.

Allocation of a Slot

There are two kinds of slots: *local* and *shared*. The `:allocation` slot option to `defclass` controls whether a local or shared slot is defined.

- A local slot is allocated at each instance (`:allocation :instance`). Each instance has its own value. Modifications affect the instance only.
- A shared slot exists only once for all instances of the class (and its subclasses) (`:allocation :class`). There's only one value for all instances. Modifications so affect all instances.

If no `:allocation` is specified, it defaults to `:instance`.

```
(defclass input-window (window)
    ((input-focus :type (or null input-window)
                  :initform nil
                  :allocation :class
                  :accessor input-focus)))
```

Mixin Classes

```
(defclass foreground-mixin ()
   ((foreground :initarg :foreground
                :accessor foreground))
   (:default-initargs
      :foreground "black"))

(defclass text-window (foreground-mixin window)
   ((text :initarg :text :accessor text)
    (font :initarg :font :accessor font))
   (:default-initargs
      :text ""
      :font '(:family :helvetica
              :face :normal
              :size 12)))

(defclass bitmap-window (window)
   ((bitmap :initarg :bitmap :accessor bitmap)))
```

Classes like `foreground-mixin` are called *mixin classes*. They are intended to be used as superclasses only; no instances are created of them. However, this is only a semantic distinction.

Creating Instances

Instances of classes are created by using `make-instance`, which returns the created instance and has the following arguments:

- Class name,
- Initargs: Initial values can be supplied for all slots which have specified the `:initarg` slot option.

Example:

```
(make-instance 'window
   :x 10 :y 10
   :width 150 :height 100
   :border-width 2)
```

returns a new instance of class `window` with the following slot values:

parent	⟨*toplevel-window*⟩
children	nil
x	10
y	10
width	150
height	100
background	“white”
border-width	2
border-color	“black”

Slot Initialization

There are different mechanisms for initializing slots of newly created instances. The value of a slot is set to

1. the value specified for an initarg supplied to `make-instance`, e.g., slots `x`, `y`, `width`, `height`, and `border-width` in the window example above,
2. the default value specified for the initarg in a `:default-initargs` class option, if no value is supplied to the `make-instance` call, e.g., slots `background`, `border-color`, and `parent` in the window example,
3. the default initial value specified by an `:initform` slot option, if no value is specified for an initarg, e.g., slot `children` in the window example.

If a slot is not initialized by one of these mechanisms, it is **unbound**.

Utility Functions

- `class-of` returns the class of a given instance.

  Syntax: `(class-of` ⟨*instance*⟩`)`

- `describe` prints information about a given object.

  Syntax: `(describe` ⟨*object*⟩`)`

- `slot-boundp` tests whether a slot has a value in a given instance.

  Syntax: `(slot-boundp` ⟨*instance*⟩ ⟨*slot-name*⟩`)`

- `slot-exists-p` tests whether an instance has a slot with a given name.

  Syntax: `(slot-exists-p` ⟨*instance*⟩ ⟨*slot-name*⟩`)`

- The function `class-name` returns the name of a given class object.

  Syntax: `(class-name` ⟨*class*⟩`)`

- The function `find-class` returns the class object for a given symbol.

  Syntax: `(find-class` ⟨*class-name*⟩`)`

Examples:

```
> (setq my-window
     (make-instance 'window :x 10 :y 10
                            :width 150 :height 100
                            :border-width 2))
#<window 124700776>

> (class-of my-window)
#<Standard-Class window>

> (describe my-window)
#<window 124700776> is an instance of class
#<Standard-Class window>:
 The following slots have :INSTANCE allocation:
 children        NIL
 parent          #<window 123527226>
 height          100
 width           150
 y               10
 x               10
 background      "white"
 border-color    "black"
 border-width    2

> (slot-boundp my-window 'parent)
t

> (slot-exists-p my-window 'foreground)
nil
```

Slot Accessors

Slot accessors are defined in the slot specifiers in the class definition:

- `:accessor` defines methods for reading the slot value and writing it with `setf`.

  Example:

  ```
  > (border-width my-window)
  2

  > (setf (border-width my-window) 1)
  1

  > (border-width my-window)
  1
  ```

- `:reader` defines a method for reading the slot value.
- `:writer` defines a method for writing the slot value.

Slot accessors are automatically defined generic functions and methods.

Slots can also be accessed with the `slot-value` function.

Example:

The slot specifier for the border-width of a window

```
(border-width :initarg :border-width
              :accessor border-width)
```

could also be defined as

```
(border-width :initarg :border-width
              :reader border-width
              :writer (setf border-width))
```

`:reader` and `:writer` slot options may also be used in addition to an `:accessor` slot option:

```
(border-width :initarg :border-width
              :accessor border-width
              :reader get-border-width
              :writer set-border-width)
```

```
> (border-width my-window)
1

> (set-border-width 2 my-window)
2

> (get-border-width my-window)
2
```

Slot Access with `slot-value`

To access a slot without invoking slot accessors or to access a slot that has no slot accessors, the `slot-value` function may be used:

Syntax:

(`slot-value` ⟨*instance*⟩ ⟨*slot-name*⟩)

(`setf (slot-value` ⟨*instance*⟩ ⟨*slot-name*⟩`)` ⟨*value*⟩)

Example:

```
> (slot-value my-window 'children)
nil

> (setq my-text-window
        (make-instance 'text-window
                       :x 50 :y 40 :text "Hello"))
#<text-window 126717516>

> (setf (slot-value my-window 'children)
        (list my-text-window))
(#<text-window 126717516>)
```

Note: `slot-value` should be used carefully, since it circumvents the interface defined for slot access.

Macros for Slot Access

with-accessors The slots are accessed with slot accessors. For each slot, a variable name and the accessor name are specified.

Example:

The foreground and background color of a text-window may be switched as follows:

```
(with-accessors ((fg foreground) (bg background))
                my-text-window
     (let ((saved-fg fg))
        (setf fg bg
              bg saved-fg)))
```

with-slots The slots are accessed with the `slot-value` function. For each slot, the slot name (or a variable name and the slot name) is specified and be used as a variable.

Example:

The width and height of a text window may be adjusted according to its text and font as follows:

```
(with-slots (width height text font)
            my-text-window
   (setf width (text-width text font)
         height (text-height text font)))
```

Constructors

For particular classes, *constructors* may be defined, which implement an external (more flexible) interface for instance creation.

Examples:

```
(defun make-point (x y)
   (make-instance 'point :x x :y y))

(defun make-region (x y width height)
   (make-instance 'region
                  :x x :y y
                  :width width :height height))

(defun make-window (x y width height &rest init-args)
   (apply #'make-instance 'window
          :x x :y y :width width :height height
          init-args))
```

A point with coordinates $(0, 0)$, for example, can be created by `(make-point 0 0)`.

The window in the example above may now be created by

```
(make-window 10 10 150 100 :border-width 2)
```

Generic Functions

- *Generic functions* are defined by `defgeneric`.
- Their behaviour depends on the classes of the arguments.
- The behaviour is implemented by *methods*.
- They are used like ordinary lisp functions.
  They return a value, can be used with `#'` ...

```
(defgeneric display (object)
   (:documentation "displays object on the screen"))

(defgeneric move (object destination)
   (:documentation "moves object to destination"))
```

A generic function definition includes the following items:

- Function specifier (symbol or the form `(setf` ⟨*symbol*⟩`)`)
- Lambda-list specifying the arguments of the generic function.
- Set of generic function options, e.g., for documentation.
- Set of method descriptions.
- Definition is done automatically when defining methods, but should be done manually for documentation.

Defining Methods

Methods are defined by `defmethod`, which returns the method object. If the generic function doesn't exist yet, it's automatically defined.

Examples:

```
(defmethod display ((disp-reg display-region))
   (fill-region disp-reg (background disp-reg)))

(defmethod refresh ((disp-reg display-region))
   (display disp-reg))
```

A method definition includes the following items:

- Function specifier (symbol or the form `(setf` ⟨*symbol*⟩`)`).
- *Method qualifiers*, e.g., `:before` and `:after`, which determine the *role* of the method.
- *Specialized lambda-list* specifying the parameters of the method.
- Set of method options, for declaration and documentation.
- Forms implementing the method body.

Method Lambda-Lists

The *specialized lambda-list* of a method specifies its parameters and determines when the method is applicable. It differs from an ordinary lambda-list in that the *required parameters* may be *specialized* in the following form:

(⟨*variable-name*⟩ ⟨*parameter-specializer*⟩)

⟨*parameter-specializer*⟩ is a class name or `(eql` ⟨*form*⟩`)`.

Congruence

The lambda-lists of the generic function and the (specialized) lambda-lists of all its methods must be *congruent*:

- Each lambda-list must have the same number of required parameters.
- Each lambda-list must have the same number of optional parameters.
- If any lambda-list mentions `&rest` or `&key`, each lambda-list must mention one or both of them.
- For keyword parameters the `defgeneric` form states the minimum set of keywords that must be accepted by *all* its methods. The methods can accept these keywords by using `&key`, `&rest` and not `&key`, or by specifying `&allow-other-keys`. Furthermore, each method can name additional `&key` parameters.

Method Selection

- A method is applicable, if all arguments subclasses to the respective parameter specifications.
- If there is more than one method applicable take method with most specific first parameter
- If there are more than one equally specific take method with most specific second parameter
- and so on

specific: defined by *class precedence list*

```
(defmethod eat ((food icecone) (person child))
  ... ) ; eat bottom first

(defmethod eat ((food icecone) (person adult)
  ... ) ; eat top down

(defmethod eat ((food cookie) person)
  ... ) ; bite into edge

(defmethod eat ((food cookie) (person greedy-adult))
  ... ) ; swallow complete

(defmethod eat ((food filled-cookie) (person child))
  ...) ; eat filling, then wafers
```

Integrating Classes and Types

- CLOS maps the space of classes into the Common-Lisp type space. Every class is a Common-Lisp type.
- **User- or system-defined classes:** Defining a named class with `defclass` defines a Common-Lisp type of the same name.
- **Built-in classes:**
  Many of the predefined Common-Lisp types have a corresponding class with the same name as the type:

`array`	`integer`	`sequence`
`bit-vector`	`list`	`string`
`character`	`null`	`symbol`
`complex`	`number`	`t`
`cons`	`ratio`	`vector`
`float`	`rational`	

- Purpose of built-in classes is to spezialize methods on corresponding lisp data types.
- There are restrictions on built-in classes, e.g.:
  - No inheritance from built-in classes
  - No `make-instance`
  - No `slot-value`

Type Functions

A class name or class object can be used as a type specifier.

The Common-Lisp functions `type-of`, `typep`, and `subtypep` can be used to discriminate among classes:

`(type-of` ⟨*instance*⟩`)` returns the class name of *class*, if *instance* is an instance of the user-defined class *class*.

`(typep` ⟨*object*⟩ ⟨*class*⟩`)` returns `t` if the class of *object* is *class* or a subclass of *class*.

`(subtypep` ⟨*class1*⟩ ⟨*class2*⟩`)` returns the values `t t` if *class1* is a subclass of *class2* or they are the same class; otherwise it returns the values `nil t`.

Example: Classes and Types

```
> (type-of my-window)
window

> (typep my-window 'node)
t

> (typep my-window 't)
t

> (subtypep 'window 'node)
t
t

> (subtypep 'window 'display-region)
t
t

> (subtypep 'display-region 'node)
nil
t
```

Class Hierarchy

Two precedence rules:

1. **Hierarchy order:** A class always has precedence over all of its superclasses (partial ordering).

2. **Definition order:** Precedence of direct superclasses of a class follows the (total) order in its `defclass`.

The *class precedence list* of a class is a total ordering on the set of the given class and its superclasses satisfying the two rules. It's ordered from most specific to least specific.

There may exist exactly one, several, or no total orderings that satisfy the rule constraints. In the second case, CLOS chooses an ordering with a “left to right – depth first – up to joins” algorithm. In the third case, an error is signaled.

The class precedence determines inheritance of properties:

- More specific classes can override or modify slots and slot options of less specific classes.

- The method selection and combination process sorts methods by precedence order (most specific method first).

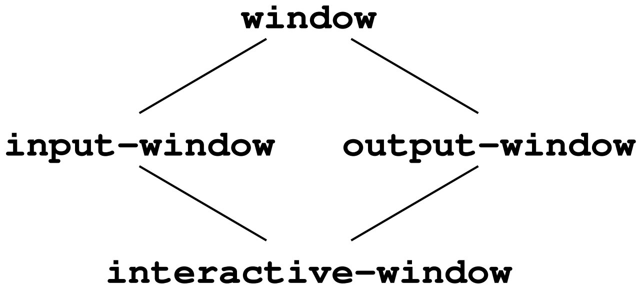
Two distinguished classes:

`t` is superclass of every class except itself. `t` has no superclasses.

`standard-object` is a superclass of all user-defined classes. It defines default behavior e.g. `print-object` and `describe-object`.

Examples: Class Precedence List

Example 1: Exactly one total ordering




```
(defclass window () ())
(defclass input-window (window) ())
(defclass output-window (window) ())

(defclass interactive-window
      (input-window output-window)
      ())
```

The class precedence list for `interactive-window`:

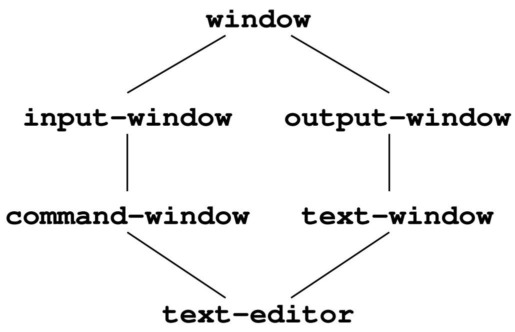
```
(interactive-window input-window output-window
window standard-object t)
```

Note:

- All classes have `t` as the last class in their class precedence list.
- All user-defined classes have `standard-object` as the second-to-last class in their class precedence list.

Examples: Class Precedence List (cont.)

Example 2: Several total orderings




```
(defclass command-window (input-window) ())
(defclass text-window (output-window) ())

(defclass text-editor
      (command-window text-window)
      ())
```

The possible total orderings for `text-editor`:

```
(text-editor command-window input-window
text-window output-window window standard-object
t)

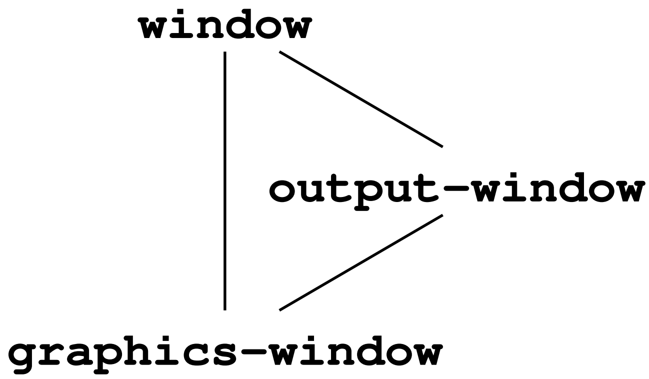
(text-editor command-window text-window
input-window output-window window standard-object
t)

(text-editor command-window text-window
output-window input-window window standard-object
t)
```

CLOS chooses the first total ordering as the class precedence list for `text-editor`.

Examples: Class Precedence List (cont.)

Example 3: No total orderings




```
(defclass graphics-window (window output-window) ())
```

Computing the class precedence list for `graphics-window` signals an error.

Inheritance of Slots

The slots of a class *C* are given by the union of the slots defined by *C* and all its superclasses.

- If only one class among *C* and its superclasses defines a slot with a given name, the slot's characteristics are determined by the defining class.
- If more than one class among *C* and its superclasses defines a slot with the same name, the slot exists only once. Its characteristics are determined by *slot option inheritance*.

Inheritance of Slot Options

`:accessor`, `:reader`, `:writer` – not inherited.

`:allocation` – If the most specific slot specifier does not specify the `:allocation` option, it defaults to `:instance`.

`:documentation` – determined by the most specific slot specifier that specifies this option.

`:initform` – determined by the most specific slot specifier that specifies this option.

`:initarg` – initargs are given by the union of all the `:initarg` slot options specified.

Different Kinds of Methods

Default Method – a method all of whose parameter specializers are the class named `t`. It is always applicable, but may be shadowed by more specific methods.

```
(defmethod eat (food person) ...)
```

Multi-Method – a method that specializes more than one parameter. Multi-methods provide a way to implement operations which truly depend on the type of more than one argument.

```
(defmethod eat ((food cookie)(person child)) ...)
```

Individual Method – a method that specializes one of its parameters on an individual Lisp object (by using the `eql` parameter specializer).

```
(defmethod eat ((food cookie)
                (person (eql *kruemelmonster*))) ...)
```

Setf Method – a method whose function specifier is a list of the form `(setf` ⟨*symbol*⟩`)`. The first element of the lambda-list corresponds to the new value that is given in the call to `setf`.

```
(defmethod (setf position) ((p point)(w window)) ...)
```

Aufruf mit: `(setf (position` ⟨*window*⟩`)` ⟨*point*⟩`)`

Method Roles

The `standard` method combination type supports unqualified methods (*primary methods*) and specific qualified methods (*auxiliary methods*). Four **roles** for methods are recognized based on their qualifiers:

none – primary methods that define the main action to be performed. They compute the return value of the generic function.

By defining a specialized primary method for a class, you *override* the inherited behaviour.

`:before` – auxiliary methods called before primary methods.

`:after` – auxiliary methods called after primary methods.

`:around` – auxiliary methods wrapped around all other methods; may run *instead* of these, but may also call some of these. With around-methods you can *extend* the inherited behavior.

Functions Used in Method Combination

`call-next-method` – calls the next method. For use in primary and around-methods.

`next-method-p` – determines whether a next method exists. For use in primary and around-methods.

Flow of Control in Standard Method Combination

The most specific applicable around-method is called.

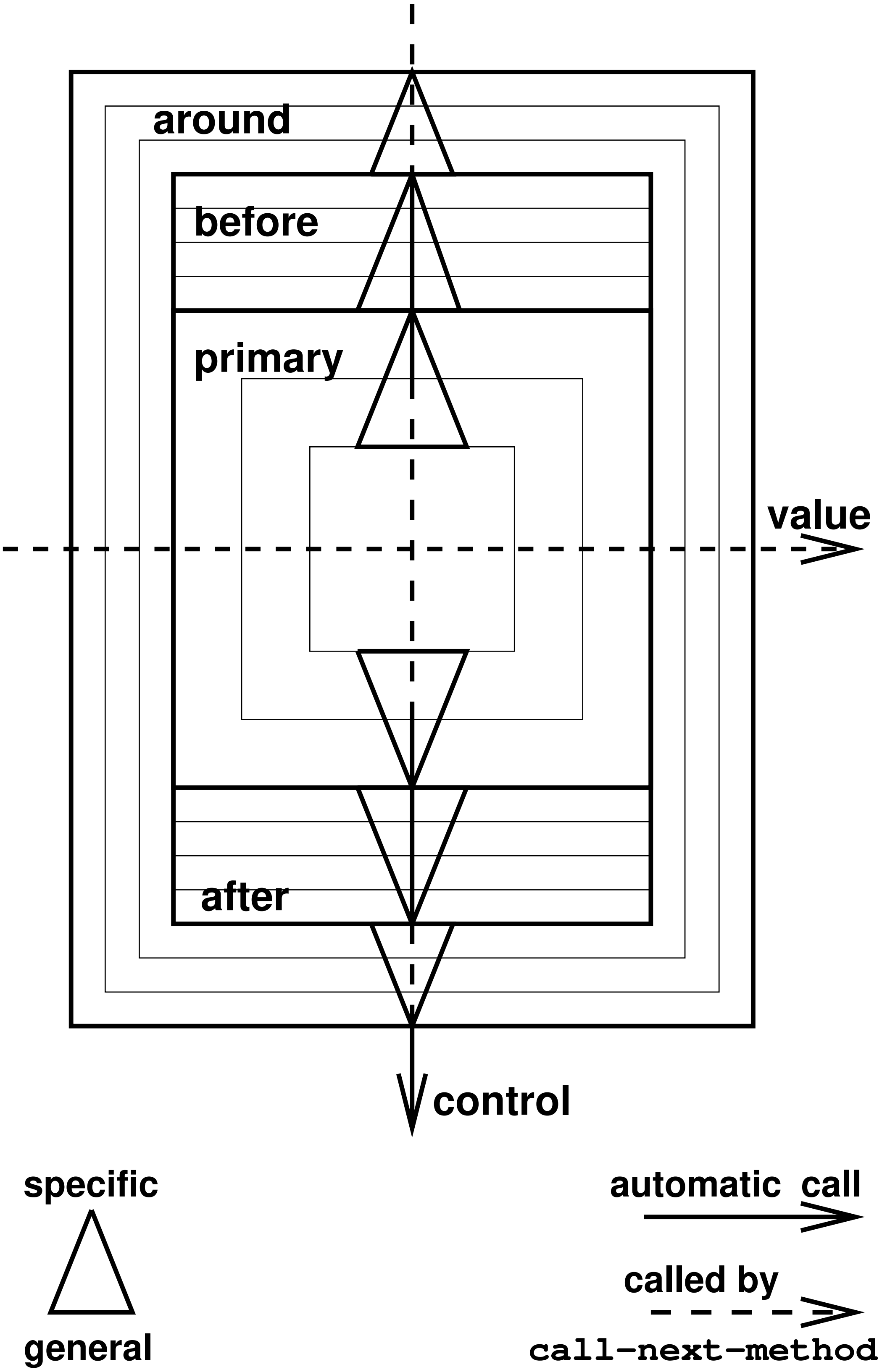
- The values that it returns are the values returned by the generic function.
- Unless it calls `call-next-method`, no other methods are called.
- If `call-next-method` is invoked, the next most specific applicable around-method is called.

If there are no more applicable around-methods

- All the applicable before-methods are called in *most-specific-first* order.
- The most specific applicable primary method is called. The values that it returns are the values returned by the generic function (or by `call-next-method`).

  If it invokes `call-next-method`, the next most specific applicable primary method is called, and so on.
- All the applicable after-methods are called in *least-specific-first* order.

Method Combination Diagram



Initialization of Instances

When `make-instance` is called, it creates an instance and calls the `initialize-instance` generic function to initialize the new instance.

- `initialize-instance` is called with the newly created instance and the defaulted initargs.
- The class `standard-object` supplies a default primary method for `initialize-instance`, which performs slot initialization by filling slots with values according to their initargs and initforms.
- User-defined subclasses typically customize initialization by writing after-methods for `initialize-instance` that do additional work.

Examples: Defining After-Methods for `initialize-instance`

Example 1: Keeping track of windows

The following after-method keeps track of windows by adding newly created windows to a data structure:

```
(defvar *all-windows* ())
(defmethod initialize-instance :after ((win window)
                                       &rest initargs)
  (push win *all-windows*))
```

Example 2: Updating the window hierarchy

Whenever a window (or a node, strictly speaking) has been created, it should be properly linked into the window hierarchy. The following after-method adds the new window to the children list of its parent:

```
(defmethod initialize-instance :after ((node node)
                                       &rest initargs
                                       &key parent)
  (add-child parent node))
```

Teil A

Readermakros (Addendum)

- Lisp-Reader
- Readtable und Makro-Zeichen
- Reader-Makros

Lisp-Reader

Lisp-Writer: Unter dem Begriff *Lisp-Writer* werden alle Methoden zusammengefaßt, die die Aufgabe haben, *Lisp-Objekte* in ihrer *Printform* an einen Stream zu senden.

Lisp-Reader: Unter dem Begriff *Lisp-Reader* werden alle Methoden zusammengefaßt, die die Aufgabe haben, *Lisp-Objekte* in ihrer *Printform* von einem Stream zu lesen.

Der Lisp-Reader

- liest zeichenweise von einem Stream,
- interpretiert dabei eine Folge von Zeichen als Printform eines Lisp-Objekts (lexikalische Analyse),
- erzeugt das entsprechende Objekt und
- gibt dieses Objekt als Funktionswert zurück.

```
> (setq bsp (read)) (a b c)
(A B C)

> bsp
(A B C)
```

Wichtig: Der Lisp-Reader kann *nicht alle* Printformen interpretieren!

Lisp-Reader

Der Lisp-Reader

- führt die *lexikalische Analyse* der Eingabe durch,
- ist als *recursive-decent parser* implementiert,
- ist *konfigurierbar* durch
  - Spezifikation der lexikalischen Bedeutung der einzelnen Zeichen und
  - Definition und Bindung von Funktionen an Makro-Zeichen (sog. Reader-Makros).

Lexikalische Bedeutung: Ein Zeichen kann zu folgenden Zeichentypen gehören:

1. whitespace
2. constituent
3. macro:
   (a) terminating macro (Bsp: `"abc"`, `(a b c)`, `'abc`)
   (b) non-terminating macro (Bsp: `#'`*func*)
4. escape:
   (a) single escape (Bsp: `\#abc`)
   (b) multiple escape (Bsp: `|#abc|`)

Zeichentypen – lexikalische Bedeutung

Readtable:

- Die lexikalische Bedeutung eines Zeichens wird durch eine *Readtable* festgelegt.
- Der Wert von `*readtable*` ist die aktuelle Readtable.
- Es gibt eine Standard-Common-Lisp-Readtable.

Standard-CL-Readtable:

<table>
<tr><td>⟨space⟩, ⟨tab⟩, ⟨page⟩, ⟨newline⟩,
⟨return⟩, ⟨linefeed⟩</td><td>whitespace</td></tr>
<tr><td><code>A...Z a...z 0...9 $ % & * + - . / :</code>
<code>< = > @ ^ _ ~</code> ⟨rubout⟩ ⟨backspace⟩</td><td>constituent</td></tr>
<tr><td><code>! ? [ ] { }</code></td><td>constituent*</td></tr>
<tr><td><code>" ' ( ) , ; `</code></td><td>term. macro</td></tr>
<tr><td><code>#</code></td><td>non term. macro</td></tr>
<tr><td><code>\</code></td><td>single escape</td></tr>
<tr><td><code>|</code></td><td>multiple escape</td></tr>
</table>

Mit * markierte Zeichen können vom Benutzer in ihrer Bedeutung verändert werden.

Escape- und Makro-Zeichen

Single Escape/Multiple Escape:

```
> '\:\,\;
|:,;|

> '(1        2 |(67\8|abc \;)
(1 2 |(678ABC| |;|)
```

Terminierendes/nicht-terminierendes Makro:

```
> (with-input-from-string (s "'ab'dfg")
    (do ((token (read s nil nil) (read s nil nil)))
        ((not token))
      (pprint token)))
'AB
'DFG
NIL

> (with-input-from-string (s "'ab###'dfg")
    (do ((token (read s nil nil) (read s nil nil)))
        ((not token))
      (pprint token)))
'|AB###|
'DFG
NIL
```

Escape- und Makro-Zeichen

Escape-Zeichen sorgen dafür, daß ein oder mehrere nachfolgende Zeichen als Zeichen vom Typ Konstituenten angesehen werden.

Makro-Zeichen sind normalerweise mit einer Funktion verbunden, die aufgerufen wird, wenn das Zeichen gerade zu Beginn eines neuen Scan-Vorganges gelesen werden. Liest jedoch der Lisp-Reader gerade Zeichen für ein Symbol/Zahl, so wird diese Funktion bei nicht-terminierenden Makros unterdrückt.

Single Escape/Multiple Escape:

Ein Single-Escape-Zeichen wie \ liest das direkt nachfolgende Zeichen (und nur dieses) und wandelt es in ein Zeichen vom Typ Konstituent um.

Ein Multiple-Escape-Zeichen wie | liest beliebig viele Zeichen bis zum nächsten Multiple-Escape-Zeichen. Dabei werden alle Zeichen in Zeichen vom Typ Konstituent umgewandelt, Single-Escape-Zeichen werden aber ausgewertet.

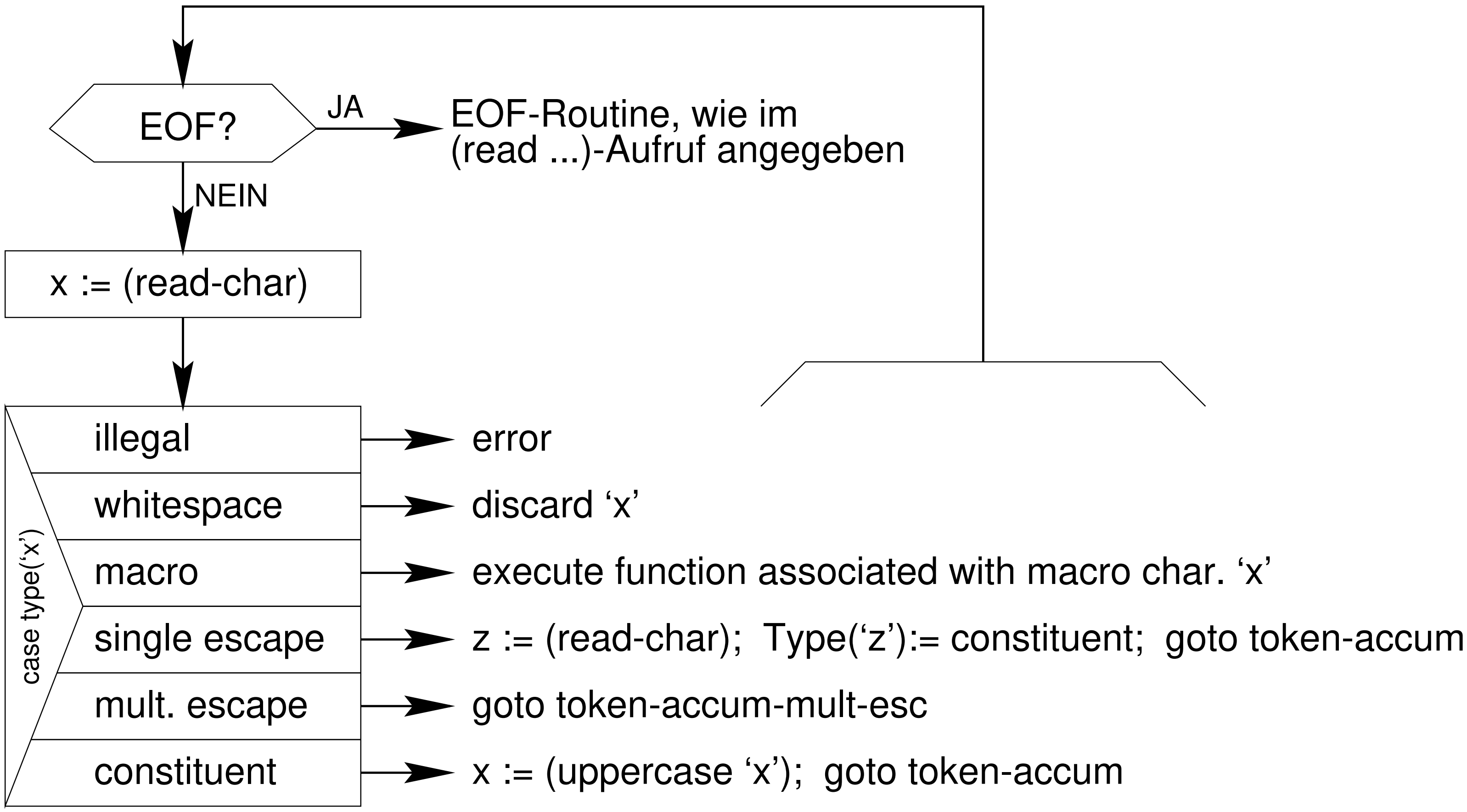
Terminierendes/nicht-terminierendes Makro:

Ein terminierendes Makro-Zeichen wie ' führt dazu, daß das vorherige Symbol vom jetzt folgenden Symbol/Zeichen getrennt wird, also ein neues Symbol/Zeichen begonnen wird.

Ein nicht-terminierendes Makro-Zeichen wie # wird mit in den Symbolnamen einbezogen.

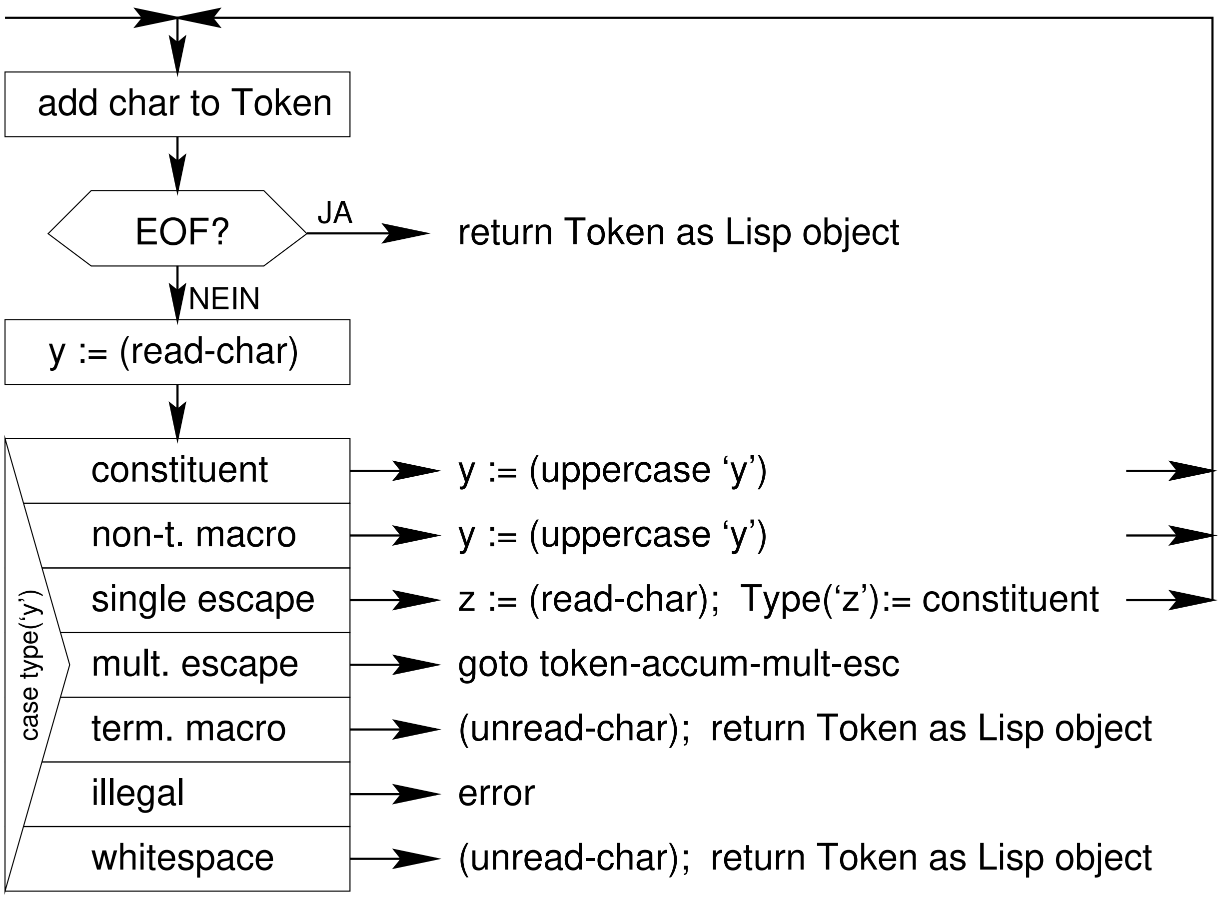
Algorithmus des Lisp-Reader

Hauptschleife:



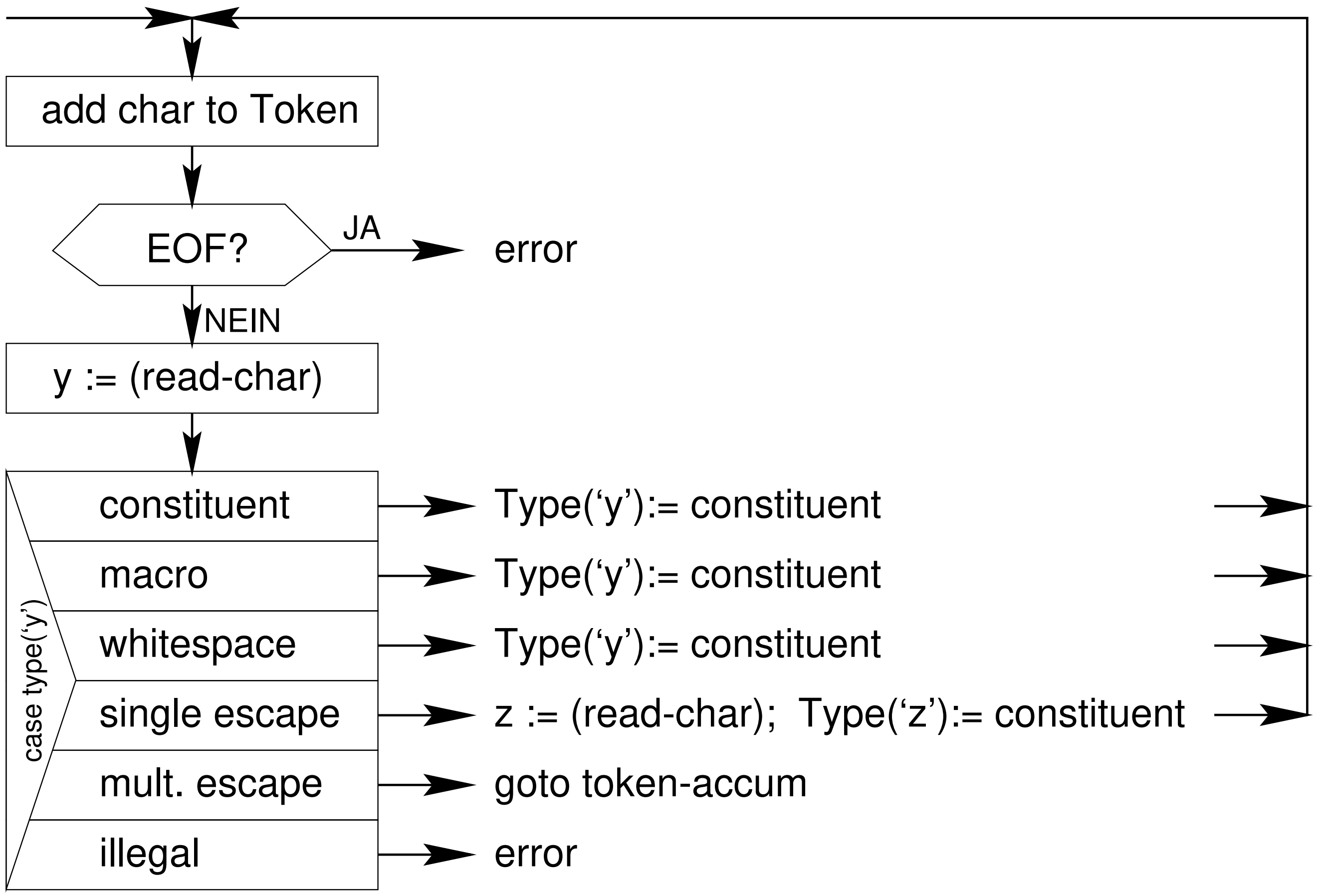
Algorithmus des Lisp-Reader

Teil „token-accum“:



Algorithmus des Lisp-Reader

Teil „token-accum-mult-esc“ :



Makro-Zeichen – Beispiele

(Die Funktion `read` wird rekursiv aufgerufen, um aufeinander folgende Lisp-Objekte zu lesen, bis das Makro-Zeichen `)` gelesen wird. Die Liste von Objekten wird im Speicher aufgebaut und am Ende zurückgegeben.

) darf nur in Verbindung mit dem Makro-Zeichen `(` (oder `#(`) verwendet werden.

' wird als Abkürzung für den Aufruf der Special-Form `quote` benutzt, mit dem Konstanten angegeben werden.

; wird für Kommentare benutzt; alle Zeichen bis zum Zeilenende werden ignoriert.

" führt dazu, daß alle nachfolgenden Zeichen als String interpretiert werden, bis ein `"` gelesen wird.

**`** wird wie das Quote-Zeichen als Abkürzung verwendet, um Konstanten anzugeben, wobei Teile davon (mit vorangestelltem `,`) ausgewertet werden sollen.

, darf nur innerhalb eines Backquote-Ausdrucks verwendet werden.

ist ein *Dispatching-Makro-Zeichen*. Es wird eine optionale Ziffernfolge gelesen, danach ein Zeichen; abhängig von diesem Zeichen wird eine Funktion aufgerufen.

Dispatching-Makro-Zeichen

Standard #-Makro-Syntax (Ausschnitt):

`#'`	`function` abbreviation
`#(`	simple vector
`#0...#9`	used for infix arguments
`#`⟨*n*⟩`=`	label the following object
`#`⟨*n*⟩`#`	reference to #= label
`#:`	uninterned symbol
`#\`	character object
`#|`	balanced comment
`#+`, `#-`	read-time conditional
`#.`	read-time evaluation
`#A`, `#a`	array
`#B`, `#b`	binary
`#C`, `#c`	complex
`#O`, `#o`	octal
`#R`, `#r`	radix-n
`#S`, `#s`	structure
`#X`, `#x`	hexadecimal

Dispatching-Makro-Zeichen (Forts.)

`#o777`

Oktaldarstellung von 511

`#S(Wuerfel Name Wuerfel-A Laenge 8 Farbe Rot)`

bezeichnet eine Instanz einer Struktur `Wuerfel`

`#2A((0 1 5) (4 6 7))`

repräsentiert eine 2×3-Matrix

`#.foo`

$\equiv$ `(eval-when (eval load) foo)`

`#,foo`

$\equiv$ `(eval-when (eval compile) foo)`

`#+sun4 (load "sun4tools")`

`#-sun4 (load "generictools")`

```
> (cons '#2=(p q r) '(x y . #2#))
((P Q R) (X Y (P Q R)))
```

Readtable-Manipulationen

Aktuelle Readtable:

```
*readtable*
```

Anlegen einer neuen Readtable:

```
> (defvar *old-readtable* *readtable*)
*OLD-READTABLE*

> (setq *readtable* (copy-readtable))
#(READTABLE @ ...)
```

Kopieren von Einträgen zwischen Readtables:

```
> (set-syntax-from-char
    #\} #\}
    *readtable* *old-readtable*)
T
```

- Mit der Datenstruktur *Readtable* wird in Lisp der Reader gesteuert. Die Readtable enthält Informationen über die lexikalische Bedeutung der einzelnen Zeichen.

- Mit der Funktion *set-syntax-from-char* ist das Kopieren von Makro-Zeichen-Funktionen von einer Readtable in eine andere möglich.

Readtable-Manipulationen (Forts.)

Makro-Zeichen-Funktion: Das Reader-Makro zur Implementierung des Quote-Zeichens liest das nächste Token und gibt es gequotet zurück:

```
> (defun single-quote-reader (stream char)
    (declare (ignore char))
    `',(read stream t nil t))
SINGLE-QUOTE-READER
```

Das Quote-Zeichen `'` wird mit dieser Funktion assoziiert:

```
> (set-macro-character #\' #'single-quote-reader)
T

> (get-macro-character #\')
#<Interpreted Function SINGLE-QUOTE-READER>
NIL
```

Dollar-Zeichen `$` soll wie Quote-Zeichen sein:

```
> (get-macro-character #\$)
#<Function READ-TOKEN>
T

> (set-macro-character #\$ #'single-quote-reader)
T

> (get-macro-character #\$)
#<Interpreted Function SINGLE-QUOTE-READER>
NIL
```

Readtable-Manipulationen (Forts.)

Dispatching-Makro-Zeichen-Funktion: Das Dispatching-Makro #{ soll aus der nachfolgenden, durch das Zeichen } begrenzten Liste von Tokens eine Liste aller möglichen Tokenpaare erzeugen.

```
(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  `',(mapcon #'(lambda (x)
                 (mapcar #'(lambda (y)
                             (list (car x) y))
                         (cdr x)))
             (read-delimited-list #\} stream t))))
```

```
> (set-dispatch-macro-character #\# #\{
                                #'|#{-reader|)
#<Interpreted Function #{-reader>
```

```
> #{p q z a  }
((P Q) (P Z) (P A) (Q Z) (Q A) (Z A))
```

```
> #{p q z a}
;; man wartet ewig, da "a}" ein Token ist!
```

Abhilfe: Terminierendes Makro-Zeichen } einführen:

```
> (set-macro-character #\} (get-macro-character #\)))
T
```

```
> #{p q z a}
((P Q) (P Z) (P A) (Q Z) (Q A) (Z A))
```

Zusammenfassung Lisp-Reader

- Der Lisp-Reader besteht aus 4 Teilen:
  1. Zeichenselektierung (lexikalische Bedeutung/Readtable)
  2. Makroexpansion
  3. Tokenakkumulierung

     sammelt Zeichen/Konstituenten, die Teile eines Tokens bilden
  4. Tokeninterpretation
- Der Benutzer kann den Lisp-Reader mit Hilfe der Funktionen
  - `set-syntax-from-char`
  - `set-macro-character`, `get-macro-character`
  - `make-dispatch-macro-character`
  - `set-dispatch-macro-character`, `get-dispatch-macro-character`

  verändern

Zusammenfassung Lisp-Reader (Forts.)

- Mit `set-macro-character` wird mit dem spezifizierten Makro-Zeichen eine Funktion assoziiert. Wenn der Lisp-Reader dieses Zeichen gelesen hat, wird die assoziierte Funktion ausgeführt.
- `get-macro-character` liefert die dem spezifizierten Makro-Zeichen assoziierte Funktion.
- Mit `set-dispatch-macro-character` wird mit dem spezifizierten Zeichenpaar eine Funktion assoziiert; das Zeichenpaar besteht aus einem zuvor deklarierten Dispatch-Makro-Zeichen und einem weiteren Zeichen.
- `get-dispatch-macro-character` liefert die dem spezifierten Zeichenpaar assoziierte Funktion.