Einführung in LISP

Helmar Gust   Armin Tischler   Tobias Thelen
Timo Steffens

März 2000

Inhaltsverzeichnis

1 Historischer Abriß **5**

2 Bedienung des VirtCampus **6**

2.1 Die Tutorumgebung 6

2.2 Darstellung von Beispielen 7

3 Grundbegriffe **8**

3.1 Atome, Listen und S-Expressions 8

3.2 Listen 11

3.3 Wertzuweisung 13

3.4 Funktionsdefinition 14

3.5 Exkurs: Der Unterschied zwischen Funktionen und Prozeduren 15

3.6 Übungen 15

4 Kontrollstrukturen **16**

4.1 Tests 16

4.2 Rekursion 18

4.3 Übungen: 20

5 Listenverwaltung **21**

5.1 Zusammenfügen 22

5.2 Zitieren 23

5.3 Übungen: 25

6 Prozeduren und Funktionen (mit verschiedene Parametertypen) **27**

7 Schleifenkonstrukte und Iteration **32**

7.1 Lokale Variablen und sequentielle Ausführung 32

7.2 Schleifen 36

7.3 Die `map`-Famile 39

7.4 Übungen: 44

8 Destruktive Listenprozeduren **45**

8.1 Einfache destruktive Prozeduren 46

8.2 place forms 50

8.3 Differenzlisten 51

8.4 Exkurs: Vorteile seiteneffektfreier Programmierung 55

8.5 Übungen 57

9 Weitere Datenstrukturen **58**

9.1 Assoziationslisten 58

9.2 Übungen: 61

9.3 Eigenschaftslisten (property lists) 63

9.4 Übungen: 65

9.5 Arrays 68

10 Funktionsdefinitionen **69**

10.1 Variablenbindungen 69

10.2 Funktionsdefinition als Variablenbindung 70

10.3 Makros 73

10.4 Übungen: 75

11 Ein- und Ausgabe **75**

11.1 Dateien 77

11.2 Übungen: 79

12 Objektorientierung mit CLOS **79**

12.1 Definition von Klassen und Objekterzeugung 80

12.2 Zugriff auf Instanzvariablen 80

12.3 Initialisierungsausdrücke 82

12.4 Methoden 83

12.5 Vererbung 89

12.6 Zugriffsbeschränkungen 93

12.7 Weitere Features von CLOS 94

13 Pattern Matching und Unifikation **95**

13.1 Pattern Matching 95

14 Prologinterpreter **98**

15 Ausschnitte aus dem *Jargon-File* **105**

Literatur **108**

Index **109**

Übersicht vordefinierter Funktionen und Prozeduren **113**

1 Historischer Abriß

Lisp ist eine Programmiersprache, die in ihren Anfängen aus den späten Fünfziger Jahren stammt und somit neben Fortran die älteste noch benutzte 'Hochsprache' ist. Der Name 'Lisp' ist eine Abkürzung für 'LISt Processing', was auf das Haupteinsatzgebiet hinweist: Im Gegensatz zu Sprachen wie Fortran, die vor allem zur Programmierung mathematischer Berechnungen entwickelt wurden, ist Lisp für Symbolverarbeitung im generelleren – also nicht auf Zahlen beschränkten – Sinne geeignet und bietet dafür mächtige dynamische Datenstrukturen, nämlich Listen. Lisp war von Anfang an die Sprache der Künstlichen Intelligenz, wurde am MIT AI-Lab[1] entwickelt und verbreitete sich in den 60er und 70er Jahren von dort in verschiedensten Dialekten und als Maschinensprache einer ganzen Rechnergeneration, den Lispmaschinen. Weitere Einblicke in die Entstehung und Entwickelung von Lisp, der frühen KI[2] und der heutigen Computerlandschaft bietet das 'Jargon-File', das in Kapitel 15, S.105 mit einigen Zitaten vorgestellt wird.

Trotz dieses Alters ist Lisp auch heute noch die wichtigste KI-Programmiersprache (obwohl Prolog, Smalltalk und neuerdings auch C++ aufholen), allerdings war die fehlende Standardisierung ein großes Problem, das durch den De-Fakto-Standard *Common Lisp* zum größten Teil gelöst wurde. *Common Lisp* ist ein sehr umfangreiches Lisp-System (und enthält z.B. mit dem *Clos* (Common Lisp Object System) eine sehr mächtige objektorientierte Erweiterung), so daß sich viele Implementationen auf wesentliche Kernbereiche von *Common-Lisp* beschränken oder, wie *Scheme*, ein 'abgespecktes' Lisp als eigene Programmiersprache darstellen. Die *Common-Lisp* Spezifikation umfaßt mehr als 1300 Seiten, während *Scheme* mit nur ca. 50 auskommt, weniger als der *Common Lisp* Index.

Lisp ist der Urvater der sogenannten funktionalen Sprachen, die neben den prozeduralen, logischen und ojektorientierten Sprachen stehen und sich von

[1] Artificial Intelligence Laboratory am *Massachussets Institute of Technology*

[2] KI bzw. AI als Abkürzungen für 'Künstliche Intelligenz' bzw. 'Artificial Intelligence' sind so weit verbreitet, daß sie auch in diesem Text durchgehend benutzt werden.

diesen zum einen durch konsequente Verwendung von Funktionen als einzigem oder zumindest hauptsächlichen[3] Mechanismus abgrenzen und zum anderen eine theoretische und mathematisch abgesicherte Fundierung in dem von Alonso Church entwickelten λ-Kalkül haben (Church 1932, 1933).

Lisp hat einige besondere Eigenschaften, die es von anderen Programmiersprachen abheben:

- dynamische Datenstrukturen, der Programmierer muß sich nicht um Speichermanagement, Zeiger u.ä. kümmern und kann so auf einer 'höheren' Ebene arbeiten
- schwaches Typenkonzept, d.h. eine Variable muß nicht einen Wert eines bestimmten Typs haben, sondern kann an beliebige Werte, Strukturen oder gar Funktionen gebunden werden
- es gibt keinen prinzipiellen Unterschied zwischen Programm und Daten
- unterstützt modularen Programmaufbau durch viele Prozeduren bzw. Funktionen

2 Bedienung des VirtCampus

2.1 Die Tutorumgebung

Die meisten Lispimplementationen treten mit dem Benutzer über ein einfaches Interface, den 'read-eval-loop', in Kontakt: Ein auswertbarer Ausdruck wird eingelesen, ausgewertet und das Ergebnis ausgegeben. Dieser Vorgang wiederholt sich, bis der Benutzer diese Schleife (loop) abbricht. Zum schnellen Ausprobieren von kleinen Beispielen ist dieses Verfahren meist ausreichend, Probleme ergeben sich aufgrund begrenzter Editiermöglichkeiten jedoch schnell. Meist kann aus einer solchen textbasierten Lispumgebung ein

[3] Lisp ist keine pure funktionale Programmiersprache wie z.B. die allerdings weniger verbreiteten Sprachen LML, HOPE, Haskell und Miranda.

Editor aufgerufen werden und das (als Textdatei) veränderte Lispprogramm wird dann erneut geladen. Im Virtcampus tritt an die Stelle einer textbasierten 'read-eval-loop' die grafische Tutorumgebung: Es steht ein Editorfenster zur Verfügung, in dem Text eingegeben, geändert, geladen und gespeichert werden kann. Der Benutzer arbeitet zunächst nur mit der Oberfläche und teilt dem System explizit mit, wann die Eingabe bewertet werden soll. Die beiden Fenster sind das Editor- und das Execute-Fenster. Im Execute-Fenster können einzelne kleinere Eingaben vorgenommen werden (z.B. einzelne Funktionsaufrufe), während das Editor-Fenster dafür gedacht ist, größere Programmteile in einer Datei zu verwalten und zu pflegen. So kann man über ein Menü Dateioperationen wie Laden und Speichern, und Editieroperationen wie Kopieren und Ausschneiden ausführen.

Die einfachen, hier vorgestellten Beispiele können zunächst alle im Execute-Fenster ausprobiert werden. Dazu wird der hinter `>` stehende Text eingetippt und danach entweder die Eingabe durch Return abgeschlossen oder der Execute-Knopf betätigt.

Desweiteren steht eine History-Funktion zur Verfügung, durch die bereits eingegebene Aufrufe (ggf. verändert) erneut ausgeführt werden können. Die Trennung von Ein- und Ausgabe hat weitreichende Implikationen für die Struktur interaktiver Programme: Dem Benutzer bleiben mehr Freiheiten in der Programmbedienung, das Programm wartet nicht einfach auf eine Tastatureingabe, sondern reagiert auf Ereignisse.

2.2 Darstellung von Beispielen

Ein weiterer wichtiger Aspekt ist die Darstellung von Beispielen. Den Eingaben wird immer `>` vorangestellt. Diese so dargestellten Ausdrücke werden vom Benutzer entweder einzeln im Executefenster eingegeben oder aber gebündelt (oder auch einzeln) im Editorfenster. Wird übers Execute-Fenster eingegeben, so wird im Ausgabefenster der entsprechende Returnwert (z.B. der Funktionsname) ausgegeben, was bei der Eingabe über das Editor-Fenster entfällt. In letzterem Fall muß das Programm nämlich explizit interpretiert und dann aufgerufen werden, um die Ergebnisse zu sehen. Desweiteren kommt es häufiger vor, daß der Returnwert, weil er an dieser Stelle uninteressant ist, weggelassen wird. Z.B. wenn der Returnwert der Wert einer Printanweisung ist. In diesem Falle würde im Ausgabefenster zweimal die gleiche Ausgabe erscheinen. Der Ausgabewert erscheint in dieser Dokumentation unter der Eingabe ohne Einrückung.

Alle Lispausdrücke werden in einer anderen Schriftart (`courier`) dargestellt, oder andersherum, alles was in `Schreibmaschinenschrift` erscheint, kann so eingetippt und ausprobiert werden. Ausnahmen sind entweder gekennzeichnet oder offensichtlich (so z.B. daß `>` natürlich nicht mit eingegeben wird).

Bei der Eingabe können sowohl Groß- als auch Kleinbuchstaben verwendet werden. Die Ausgabe erfolgt jedoch, abgesehen von `nil` und Strings, in Großbuchstaben, da intern alle Klein- auf Großbuchstaben abgebildet werden.

3 Grundbegriffe

Die Lispsyntax ist sehr einfach, obwohl auf den ersten Blick sehr viele Klammern den Eindruck unnötiger Komplexität vermitteln. Die wichtigsten Gundbegriffe, die sich in allen höheren Lispkonstrukten wiederfinden, werden in diesem Kapitel vorgestellt.

3.1 Atome, Listen und S-Expressions

Die Grundstruktur für Programme und Daten in Lisp bilden nur wenige verschiedene Arten von Ausdrücken, genauer:

- *Atome*, die grundlegenden Objekte in LISP, z.B. Zahlen oder Zeichenketten,

- oder Gruppen solcher Atome, die *Listen* genannt werden; Listen selbst können wiederum Listen als Elemente besitzen, was für Lisp von fundamentaler Bedeutung ist. Listen werden von runden Klammern eingeschlossen, die einzelnen Elemente werden durch Leerzeichen voneinander getrennt.

Beispiele für Lispausdrücke sind demnach:

```
a
1
langes-ausdrucks-beispiel
(1 2 3)
(liste mit atomen)
(atom (noch eine liste))
(1 1 (2 2 (3 3) 2 (3 (4))) 1 1)
```

Das letzte Beispiel zeigt, daß Listen auch geschachtelt werden können, so daß komplexere Strukturen sowohl für Daten als auch die Programme selbst aufgebaut werden können. Von der äußeren Form her sind Daten und Programme identisch, nämlich aus Atomen und Listen aufgebaut. Der Unterschied liegt in der Bearbeitung, in der *Auswertung*. Ausdrücke, die ausgewertet werden können, werden S-Expressions[4] genannt.

Bei der Auswertung von Audrücken sind folgende Fälle zu unterscheiden:

- *Zahlen* „evaluieren zu sich selbst“, d.h. wird eine Zahl ausgewertet, kommt wieder diese Zahl heraus
- *Strings,* z.B. `"Hallo"`, evaluieren ebenfalls zu sich selbst
- `t` und `nil` stehen für *wahr*(`t`) bzw. *falsch*(`nil`) und evaluieren auch zu sich selbst

[4] Der Name S-Expression stammt von dem Namen „Symbolic Expression“, im Gegensatz zu M-Expression („Meta-Expression“), eine heute unübliche Bezeichnung.

- *Listen*, das ist alles, was in runde Klammern eingeschlossen ist, werden als Funktionsaufruf aufgefasst (siehe unten)
- *Symbole,* das ist alles andere, können an einen Wert gebunden sein. Z.B. könnte das Symbol `tag` den Wert `montag` haben, so daß `tag` ausgewertet `montag` ergäbe.

Listen werden nach dem Muster

```
(funktionsname arg-1 ... arg-n)
```

als Funktionsaufruf aufgefasst und ausgewertet (d.h. die Listen werden durch den Wert der Funktion „ersetzt“).

`(funktionsname arg-1 ... arg-n)` würde so verwendet einen Funktionsaufruf darstellen, wobei 'funktionsname' der Name der Funktion und 'arg-1' bis 'arg-n' die Argumente sind. Anders ausgedrückt: Findet der Lispinterpreter eine S-Expression in Listenform, so versucht er, eine Funktion aufzurufen, deren Name das erste (oder einzige) Element des Ausdrucks ist. Alle weiteren Elemente werden wiederum als S-Expressions interpretiert, ausgewertet und die Ergebnisse der Funktion als Parameter übergeben.[5]

Ein einfacher Funktionsaufruf ist beispielsweise die Addition zweier Zahlen:

```
>  (+ 3 4)
7
```

Zu beachten ist hier, daß in Lisp die Prozedurnamen immer an erster Stelle stehen, auch bei arithmetischen Ausdrücken. Diese sogenannte Präfixnotation gewährleistet eine einheitliche Form, unabhängig von der Anzahl der beteiligten Argumente. Ein komplexeres Beispiel macht deutlich, daß keine Prioritätsregeln, bzw. Klammerung notwendig ist, die nicht für die Listenstruktur erforderlich ist:

[5] Das Prinzip, daß die Argumente ausgewertet und dann übergeben werden, gilt leider nicht so universell: Es gibt spezielle Funktionen, deren Argumente nicht ausgewertet werden. Im Regelfall darf aber angenommen werden, daß die Argumente vor Auswertung der Funktion ausgewertet werden.

```
Infix-Notation:    (3 + 4) * (5 + 6)
Praefix-Notation:  (* (+ 3 4) (+ 5 6))
```

3.2 Listen

Listen sind die zentrale Datenstruktur in Lisp. Im vorhergehenden Abschnitt wurde erwähnt, daß der Lispinterpreter jede S-Expression auszuwerten versucht. Betrachtet man Listen allerdings als Datenstruktur, so sollen sie in der Regel nicht ausgewertet, d.h. als Funktionsaufruf oder Wertersetzung aufgefasst werden. Eine Liste von Namen, wie z.B.

```
(bernd michael maria),
```

soll natürlich nicht als Funktionsaufruf interpretiert werden, so daß die Funktion `bernd` mit den Argumenten `michael` und `maria` aufgerufen würden, sondern die Liste mit den Namen soll, so wie sie ist, benutzbar sein. Dazu ist es nötig die Liste vor der Auswertung, sprich der Interpretation als Funktion, zu schützen. Dieses Schützen geschieht durch „zitieren“, oder englisch 'quote':

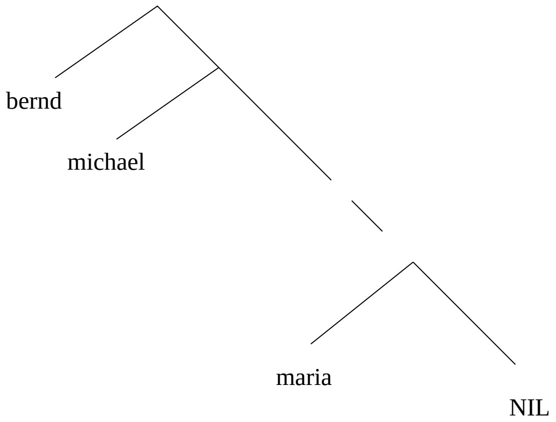
```
>  (bernd michael maria)
error - unbound variable BERND
>  (quote (bernd michael maria))
(bernd michael maria)
```

`quote` ist eine besondere Funktion die ihre Argumente unausgewertet zurückgibt. Weil `quote` sehr häufig benutzt wird, gibt es eine abkürzende Schreibweise, die fast ausschließlich verwendet wird: [6]

```
'(bernd michael maria)
```

Listen können allgemein auch als Binärbäume dargestellt werden, wie z.B.:

[6] Diese Schreibweise ist tatsächlich nur eine Abkürzung, die bereits von der Einleseroutine wieder in `(quote (bernd michael maria))` umgewandelt wird.



`(bernd michael maria)` ist nun also als Liste von drei Daten verwendbar. Doch wie läßt sich auf die Komponenten dieser Liste zugreifen? Wie aus der obigen Darstellung als Binärbaum ersichtlich, werden Listen als aus zwei Komponenten bestehend betrachtet, dem ersten Element (steht im linken Teilbaum) und dem Rest der Liste (steht im rechten Teilbaum). Es gibt zwei built-in Funktionen, die diese Komponenten liefern, `car` ('contents of address part of register') und `cdr` ('contents of decrement part of register'). Diese Bezeichnungen beziehen sich auf die erste Maschine, auf der Lisp implementiert wurde (s.a. Kapitel 15, S.106).

```
>  (car '(bernd michael maria))
BERND
>  (cdr '(bernd michael maria))
(MICHAEL MARIA)
>  (car (cdr '(bernd michael maria)))
MICHAEL
```

Das letzte Beispiel zeigt, daß Funktionsaufrufe auch geschachtelt werden dürfen, zu lesen sind solche Ausdrücke leicht von innen nach außen: Der Rest (`cdr`) der Liste `(bernd michael maria)` ist die Liste `(michael maria)`, das erste Element (`car`) davon ist das Atom `michael`.

Für den Zugriff auf Listen gibt es zunächst einmal nur diese beiden Zugriffsoperationen: Auf das erste Element und auf den Rest. Was zunächst etwas unflexibel und wenig aussieht, erweist sich aber als mächtiger und leistungsfähiger Mechanismus.[7]

3.3 Wertzuweisung

Mit der Prozedur `setq` können Symbolen Werte zugewiesen werden:

```
>  (setq a 10)
10
```

weist dem Symbol `a`, das als globale Variable betrachtet werden kann, den Wert `10` zu. `setq` steht für 'SET Quoted', d.h. das erste Argument wird automatisch als nichtauszuwertender Ausdruck betrachtet und muß daher nicht explizit zitiert werden. Das zweite Argument hingegen darf natürlich eine beliebige S-Expression sein, z.B.: `(setq a (+ 4 6))`. Die Auswertung eines Symbols liefert seinen Wert, d.h. um den Wert eines Symbols ausgeben zu lassen (z.B. um Bindungen zu überprüfen) kann es einfach alleine auf der Eingabezeile stehen. Wird versucht, Symbole auszuwerten, die an keinen Wert gebunden sind, wird eine Fehlermeldung ausgegeben. Eine Sonderstellung nehmen numerische Symbole (Zahlen) und Strings ein; sie evaluieren immer zu sich selbst, d.h. der Wert von `3` ist immer 3 und kann auch nicht verändert werden (`(setq 3 4)` führt ebenfalls zu einer Fehlermeldung).[8] Dasselbe gilt für die reservierten Symbole `t` und `nil`.

```
>  a
10
>  4
4
>  b
```

[7] Bei dem aus prozeduralen Programmiersprachen bekannten Konzept der linearen Listen ist es nicht anders: Das erste (aktuelle) Element kann erfragt werden oder der Zeiger auf das aktuelle Element auf das nächste (den Rest der Liste) weitergesetzt werden.

[8] Es gibt in der Tat Programmiersprachen, in denen so etwas möglich ist, z.B. Tcl.

```
error: unbound variable - B
> (setq b a)
10
> (setq b 'a)
a
```

3.4 Funktionsdefinition

Bisher wurden nur vordefinierte Funktionen wie `car, cdr, +, setq` verwendet, wie lassen sich aber Funktionen neu definieren? Da in Lisp alle Ausdrücke S-Expressions sind, werden Funktionen auch nach diesem Schema definiert:

```
(defun name argumente rumpf)
```

Die Prozedur `defun` sorgt dafür, daß an das Symbol `name` die Funktionsdefinition gehängt wird. Wie dieses technisch geschieht, darauf wird in Kapitel 10 noch genauer eingegangen. Eine Funktion, die das zweite Element einer Liste ausgibt, könnte so definiert werden:

```
> (defun zweites (liste)
      (car (cdr liste)))
ZWEITES
```

Der Wert des Ausdrucks, der Funktion `defun` ist nicht das interessante, sondern das was bei der Auswertung *passiert*: Die Definition der Funktion `zweites`.

Eine Beispieleingabe zum Testen der Funktion könnte folgendermaßen aussehen:

```
> (zweites '(a b c))
B
```

Eine neu definierte Funktion kann natürlich beliebig komplex sein und kann nach der Definition wie eine interne Funktion benutzt werden, d.h. kann an jeder Stelle stehen, an der auch der Name einer vordefinierten Funktion stehen kann.

3.5 Exkurs: Der Unterschied zwischen Funktionen und Prozeduren

Auch wenn es eventuell vorkommen wird, daß die Begriffe *Funktion* und *Prozedur* synonym verwendet werden, so sind sie streng genommen zu unterscheiden. Eine *Prozedur* ist eine grundlegende Einheit in Lisp, die spezifiziert, wie etwas getan werden soll. Eine *Prozedur* kann vom Benutzer definiert oder vom Lisp-System zur Verfügung gestellt werden, dann wird sie auch *Grundprozedur* genannt. Bewirkt eine Prozedur Dinge, die weiter gelten, wenn die Prozedur bereits beendet ist, so spricht man von *Nebeneffekten* oder auch *Seiteneffekten.* Eine Prozedur, die nur ihren Wert und zwar ausschließlich mit Hilfe ihrer Argumente ermittelt, heißt *Funktion.* Prozeduren sind also keine Funktionen, wenn sie Nebeneffekte haben. Demnach ist `setq` eine Prozedur, aber keine Funktion. `car` und `cdr` sind dagegen Funktionen.[9]

3.6 Übungen

1. Wie läßt sich das Symbol `mittwoch` aus den folgenden Ausdrücken unter Verwendung von `car` und `cdr` herausholen?

```
(montag dienstag mittwoch donnerstag)
((montag dienstag) (mittwoch donnerstag))
(montag (dienstag) ((mittwoch)) (((donnerstag))))
((((montag))) ((dienstag)) (mittwoch) donnerstag)
(((montag) (dienstag) (mittwoch) (donnerstag)))
((((montag) dienstag) mittwoch) donnerstag))
```

[9] An dieser Stelle zeigt sich, daß Lisp keine pure funktionale Sprache ist, denn dann wären Prozeduren in diesem Sinne nicht erlaubt. So gibt es in puren funktionalen Sprachen z.B. keine globalen Variablen, denn Variablen werden dort ausschließlich in einem intuitiv aus mathematischen Formeln bekannten Sinne verwendet: als eventuell noch unbekannte Konstanten.

4 Kontrollstrukturen

4.1 Tests

Oft ist es notwendig, daß im Programmverlauf an einer Stelle abhängig vom Ausgang eines Testes unterschiedliche Aktionen als nächstes durchgeführt werden sollen, bzw. welcher Wert im Sinne einer mathematischen Fallunterscheidung zurückgeliefert werden soll. Diese Auswahl wird häufig mit Hilfe von Prädikaten in Verbindung mit `cond` ausgeführt. Unter einem Prädikat versteht man eine Funktion, deren Wert entweder wahr oder falsch ist, wobei falsch immer durch `nil` angezeigt wird und wahr durch `t` oder irgendeinen von `nil` verschiedenen Wert. Die Prozedur `cond` hat folgende Syntax:

```
(cond (<bedingung-1> <aktion-1> ...)
      (<bedingung-2> <aktion-2> ...)
      ...
      (<bedingung-n> <aktion-n> ...))
```

Jede der Listen `(<bedingung> <aktion> ...)` wird Klausel genannt. Die Idee dabei ist: Es werden nacheinander die Bedingungsteile ausgeführt, bis ein Wert ungleich `nil` erreicht ist, d.h. eine erfüllte Bedingung gefunden worden ist. Dann wird die Aktion, die zu der Bedingung gehört, ausgeführt. Der Wert des letzten Termes der Aktion wird als Wert der gesamten `cond`-Prozedur zurückgegeben. Pro Bedingung kann also nicht nur eine Aktion ausgeführt werden sondern auch mehrere. Eine Funktion `minimum`, die die kleinste von zwei Zahlen ausgibt, könnte folgendermaßen aussehen:

```
>   (defun minimum (x y)
        (cond ((< x y) x)   x < y --> x
              (T y)))       sonst --> y
MINIMUM

>  (minimum 3 4)
3
```

Die letzte Klausel des obigen Bedingungstermes beginnt mit `t`. Falls alle anderen Bedingungen der vorausgehenden Klausen `nil` liefern, wird die Aktion dieser Klausel ausgeführt. Der Test 'fällt durch' und wird am Schluß abgefangen. Es ist sinnvoll, diese Konvention in den meisten Fällen einzuhalten, auch wenn die letzte Klausel 'eigentlich' nicht erreicht werden kann, so wird das Programm robuster, Fehler und unsinnige Werte können abgefangen werden.[10]

Häufig liegt ein Fall wie in obigem Beispiel vor, daß zwei alternative Aktionen abhängig von einem Testergebnis ausgewählt werden müssen. Dazu gibt es eine Prozedur `if`, die genau für diesen Sachverhalt geeignet ist. Bei Verwendung von `if` anstelle von `cond` würde obiges Beispiel wie folgt aussehen:

```
>  (defun minimum (x y)
        (if (< x y) x y))
MINIMUM
```

Größere Programme lassen sich nur mit einem streng durchgehaltenen sinnvollen Programmierstil erstellen und warten. Dabei ist es vernünftig zwei 'Regeln' zu beachten. Zum einen ist es hilfreich, die Klammern in einer Zeile ausgewogen zu halten, wie in der Zeile mit dem `if`. Ist dieses nicht möglich, da sonst die Zeile zu lang wird, so sollte man dafür sorgen, daß unter einer öffnenden Klammer die entsprechenden Zeilen tiefer eine schließende steht.

Die letzte Regel wird von erfahreneren Lispprogrammierern in den allermeisten Fällen nicht eingehalten, sondern durch die Regel ersetzt, daß niemals ausschließlich Klammern in einer Zeile stehen dürfen. Viele Editoren haben zudem spezielle Lispmodi bzw. zumindest die Möglichkeit, Klammerung zu überprüfen.

[10] Die `cond`-Prozedur ist am ehesten mit dem CASE-Konstrukt aus prozeduralen Programmiersprachen vergleichbar, auch dort ist es ratsam am Ende einen *default* Fall anzufügen, um Fehler abfangen zu können.

4.2 Rekursion

Ein häufig verwendetes Prädikat ist `null`, das testet, ob eine Liste leer ist. Das folgende einfache Beispiel geht rekursiv eine Liste durch und gibt die Elemente mit `print`[11] aus. Rekursives Abarbeiten von Listen ist ein geradezu notwendiges Verfahren in Lisp und die meisten Funktionen arbeiten nach demselben Muster:

```
>  (defun test (x)
       (cond ((null x) nil)  ; Abbruch, wenn Liste leer
             (T (print (car x)) (test (cdr x)))))

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

An diesem Beispiel sieht man auch, daß in einem Bedingungsteil mehrere Ausdrücke stehen können, aber nur der letzte Teil den Returnwert liefert. In diesem Fall ist `nil` der Returnwert, der durch alle Ebenen der Rekursion durchgereicht wird.

Wer mit dem Konzept der Rekursion noch nicht ausreichend vertraut ist, sollte sich genau vor Augen führen, wie die obige Funktion arbeitet. Das allgemeine Schema dabei ist:

```
FUNKTION foo (liste)
      1. Abbruchbedingung erreicht? (Ist die Liste leer?) --> Ende
      2. Mit erstem Element der Liste etwas tun (z.B. ausgeben)
      3. foo mit dem Rest der Liste erneut aufrufen
```

[11] Zu den Ausgabefunktionen, s. Kapitel 11, S.75

Erstaunlich scheint, daß *dieselbe* Funktion wieder aufgerufen wird. Das ist auch der Fall, allerdings wird dabei jedesmal neuer Platz für die lokalen Variablen reserviert, so daß die verschiedenen Instanzen derselben Funktion sich nicht behindern. Eine andere rekursive Funktion ist z.B. das Aufsummieren aller Elemente einer Liste, die so definiert werden könnte:
Die Summe aller Elemente einer Liste ist das erste Element plus die Summe aller Elemente der Restliste.

```
(defun summe (liste)
    (cond ((null liste) 0)
          (T (+ (car liste) (summe (cdr liste))))))
```

Wird `summe` mit `(summe '(1 2 3 4))` aufgerufen, ergibt sich folgende Funktionsschachtelung:

```
(summe '(1 2 3 4))
  --> (+ 1 (summe '(2 3 4)))
      --> (+ 2 (summe '(3 4)))
          --> (+ 3 (summe '(4)))
              --> (+ 4 (summe nil))
                  (+ 4 0)
              <--- 4
              (+ 3 4)
          <--- 7
          (+ 2 7)
      <--- 9
      (+ 1 9)
  <--- 10
10
```

Die Variable `liste` wird dabei bei jedem Funktionsaufruf in der aufgerufenen Funktion anders belegt, bleibt aber in der aufrufenden Funktion (mit ursprünglichem Wert) erhalten. Wenn die Rekursionstiefe zu groß wird (oft durch einen Fehler, wenn z.B. der Rekursionsabbruch vergessen wird), wird immer wieder Speicher für die lokalen Variablen (dazu zählen hier insbesondere auch die Argumente) angelegt, bis der Speicher für lokale Variablen 'überläuft' und eine Fehlermeldung ausgegeben wird.

Mit dem Prädikat `equal` können Ausdrücke auf Gleichheit überprüfen. (Es gibt noch weitere speziellere Vergleichsoperatoren, s. dazu Kapitel 7.3.)

```
>  (setq x 'a)
A
>  (equal 'a x)
T
```

Soll an einer Stelle überprüft werden, ob mehrere Ausdrücke nicht `nil` liefern, so läßt sich dieses mit `and` durchführen. `and` wertet seine Argumentausdrücke der Reihe nach aus. Liefert ein Ausdruck `nil` so wird `and` abgebrochen und der Returnwert ist ebenfalls `nil`. Das Prädikat `or` arbeitet ganz analog zu `and`. Es wertet seine Argumente solange der Reihe nach aus, bis eines einen Wert ungleich `nil` liefert. Dann wird die Auswertung gestoppt und der Wert zurückgegeben.

4.3 Übungen:

1. Definiere mit `defun` eine Funktion `mein-element`, die angibt, ob ein Element in einer Liste vorhanden ist. Wenn ja, so soll der Rest der Liste beginnend mit dem gesuchten Element ausgegeben werden, andernfalls `nil`. Die Funktion soll mit zwei Argumenten aufgerufen werden. Beispiel:

   ```
   >  (mein-element 'a '(s a d f))
   (A D F)
   ```

2. Definiere analog zu 1. eine Funktion `paarelement`, die feststellt, ob zwei Elemente aufeinanderfolgend in einer Liste vorhanden sind. Beispiel:

```
>  (paarelement 'a 'd '(s a d f))
(A D F)
```

3. Schreibe eine Funktion, die die Fibonnaccizahlen berechnet. Die Fibonaccizahl von 0 ist 1, die von 1 ist ebenfalls 1 und die nächste Fibonnaccizahl ergibt sich jeweils aus der Summe ihrer beiden unmittelbaren Vorgänger. Die Zahlen lauten also:
   1 1 2 3 5 8 13 21 34 55 89 ...
   (Zahlen lassen sich mit dem Prädikat `=` vergleichen.)

4. Schreibe eine Funktion, die die Anzahl der Elemente einer Liste ermittelt. Die Elemente innerhalb geschachtelter Schleifen werden nicht gezählt, d.h. nur die Anzahl der Elemente der obersten Ebene ist relevant.

5. Die Funktion `n-tes` soll mit zwei Argumenten aufgerufen werden, wobei das erste Argument eine natürliche Zahl und das zweite Argument eine Liste sein soll. Als Returnwert soll das n-te Element der Liste geliefert werden. Sollte dieses nicht möglich sein, weil z.B. die Liste weniger als `n` Elemente enthält, so soll das Ergebnis `nil` sein. Beispiel:

```
>  (n-tes 3 '(a b c d e f))
C
```

5 Listenverwaltung

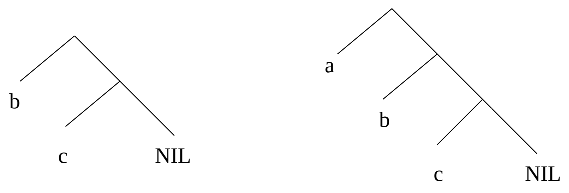
Die beiden grundlegenden Funktionen zum Listenzugriff wurden bereits in Kapitel 3.2 vorgestellt: `car` und `cdr`. Es fehlen noch Möglichkeiten zur Konstruktion und weiteren Manipulation von Listen. Außerdem bietet Lisp noch ausgefeiltere Möglichkeiten des Zitierens als `quote`.

5.1 Zusammenfügen

Um ein Element an eine Liste vorne anzuhängen, gibt es die Grundprozedur `cons`.

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

Was dabei passiert ist, daß der Baum so erweitert wird wie unten angedeutet:



Damit erklärt sich auch das Ergebnis des nächsten Beispiels:

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

Denn die Liste `(a b)` wird zum ersten Element der Ergebnisliste. Das Ergebnis einer `cons`-Operation[12] ist eine 'cons-Zelle'. Eine 'cons-Zelle' ist ein Listenelement bestehend aus zwei Feldern: Einem Feld mit Dateninhalt und einem mit einem Zeiger auf das folgende Element der Liste. Auf das erste Feld einer 'cons-Zelle' wird mit `car` zugegriffen, auf das zweite mit `cdr`. `cons` baut eine neue 'cons-Zelle' mit dem ersten Argument als Datum im ersten Feld und einem Zeiger auf das zweite Argument im zweiten Feld. In gewissem Sinne stellt `cons` eine 'inverse' Beziehung zu `car` und `cdr` her, d.h.

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

[12]Der Name `cons` stammt von *construct* oder *constructor*

Im Gegensatz zu `cons` kann `list` mit einer beliebigen Anzahl von Argumenten aufgerufen werden. Jedoch fügt `list` die Elemente auf andere Weise zusammen als `cons`.

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

`list` erzeugt eine Liste seiner Argumente.

```
>  (list '(a b) '(c d))
((A B) (C D))
```

Eine dritte Grundprozedur, die Ausdrücke zusammenfügt, ist `append`. Diese kann ebenfalls beliebig viele Argumente (Listen) haben. Sie fügt die *Elemente* ihrer Argumente zusammen, was sie von `cons` und auch von `list` unterscheidet:

```
>  (append '(a b) '(c d))
(A B C D)
```

Die Elemente selbst werden von `append` nicht verändert:

```
>  (append '((a) (b)) '((c) (d)))
((A) (B) (C) (D))
```

5.2 Zitieren

Weitere nützliche Werkzeuge im Umgang mit Listen, sind `backquote` (‘), `comma`(,) und `comma-at`(,@). Das `backquote` (Quasi-Apostroph) erleichtert das Ausfüllen von Schablonen, womit gemeint ist, Ausdrücke zu erzeugen, in denen das meiste fixiert ist und nur wenige variable Details enthalten sind. Das `backquote`(‘) unterdrückt wie das `quote`(’) die Auswertung des folgenden Ausdrucks, mit dem Unterschied jedoch, daß jedes `comma` in diesem Ausdruck bewirkt, daß der nach ihm folgende Ausdruck ausgewertet wird. Beispiel:

```
>  (setq variable 'beispiel)
BEISPIEL
>  `(Dies ist ein ,variable)
(DIES IST EIN BEISPIEL)
```

Das `comma-at`(,@) in einer Liste, der ein `backquote` vorangestellt ist, bewirkt, daß die Elemente der ihm folgenden Liste an die Stelle der Liste eingefügt werden, an der sich das `comma-at`(,@) befindet.

```
>  (setq variable '(etwas schwieriges))
(ETWAS SCHWIERIGES)

>  `(dies ist ein ,@variable beispiel)
(DIES IST EIN ETWAS SCHWIERIGES BEISPIEL)
```

Der Unterschied zwischen ',' und ',@' wird klar, wenn man sich folgendes Beispiel anschaut:

```
>  `(dies ist ein ,variable beispiel)
(DIES IST EIN (ETWAS SCHWIERIGES) BEISPIEL)
```

Das ',@' erhält also die Struktur der Subliste nicht.

Weitere Funktionen, die den Umgang mit Listen erleichtern, sind `length`, `reverse`, `subst` und `last`. `length` zählt die Anzahl der Elemente der obersten Ebene einer Liste. `reverse` dreht die Reihenfolge der Elemente der obersten Ebene einer Liste um. Beide erwarten als Argument eine Liste, deren Elemente entweder Atome oder Listen sind. Sie verändern jedoch solche Sublisten nicht.

```
>  (length '(a b))
2
>  (length '((a b) (a b)))
```

```
2
>  (length (append '(a b) '(a b)))
4

>  (reverse '(a b c))
(C B A)
>  (reverse '((a b)(c d)))
((C D) (A B))
>  (reverse (append '(a b) '(a b)))
(B A B A)
```

`subst` führt die Ersetzung aller Vorkommen eines Atoms in einem Ausdruck durch. Die Syntax dabei lautet:

```
(subst <neuer-Ausdruck> <altes-Atom> <zu-aendernder-Ausdruck>)
```

```
>  (subst 'a 'x '(+ (* x y) x))
(+ (* A Y) A)
>  (subst '(+ a b) 'x '(* x y))
(* (+ A B) Y)
```

`last` liefert eine Liste, die nur das letzte Element einer als Argument übergebenen Liste enthält.

```
>  (last '(a b c))
(C)
>  (last '((a b) (c d)))
((C D))
```

5.3 Übungen:

1. Weise unter Verwendung von `setq` dem Symbol `x` den Wert `a` und dem Symbol `y` den Wert `b` zu `(setq symbol 'wert)`. Wie lassen sich

mit `list` aus `x` und `y` die Listen `(a b)`, `(x b)`, `(a y)` bzw. `(x y)` erzeugen?

2. Die Symbole haben die gleichen Werte wie in 1. Was liefert `(cons x y)` ?

3. Betrachte folgendes:

   ```
   (setq x '(a b))
   (setq y '(c d))
   ```

   Wie lassen sich mittels `cons` die Listen `((a b) c d)` und `(x c d)` bzw. die Punktpaare `((A B) . Y)` und `(X . Y)` erzeugen?

4. Schreibe die Funktion `lösche`, die alle Vorkommen eines angegebenen Elementes aus einer Liste löscht. Der Returnwert der Funktion soll die Eingabeliste ohne das gelöschte Element sein. Sollte das angegebene Element nicht in der Liste vorkommen, so soll die ursprüngliche Liste unverändert zurückgegeben werden.

   ```
   > (loesche 'a '(d a s a r t w))
   (D S R T W)
   ```

5. Die Funktion `menge` soll aus einer Liste von Elementen eine Menge in folgendem Sinne machen. Alle Elemente, die in der Liste mehrfach vorkommen, sollen in der zurückgegebenen Liste nur noch einfach vorkommen. Verwende bei der Definition der Funktion `menge` eine Funktion wie `lösche`.

   ```
   > (menge '(a s d a s d f g d h))
   (A S D F G H)
   ```

6. Definiere eine Funktion `verkette`. Denke dabei an das Prädikat `append` in Prolog.

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

7. Die Funktion `vereinigung` soll aus zwei einzugebenden Mengen wieder eine Menge erzeugen, die alle Elemente der beiden Mengen enthalt, wobei jedes Element, wie für eine Menge üblich, nur einmal vorkommt. Benutze bei der Definition von `vereinigung` möglichst Funktionen aus diesem bungsblock.

```
> (vereinigung '(a b c) '(d a f))
(A B C D F)
```

8. Schreibe eine Funktion `invertieren`, die wie der Name schon sagt, die Reihenfolge der Elemente einer Liste invertieren soll. Die Funktion `verkette` kann dabei sehr nützlich sein.

```
> (invertieren '(a b c d))
(D C B A)
```

9. Die Funktion `durchschnitt` soll aus zwei einzugebenden Mengen die Schnittmenge bilden.

```
> (durchschnitt '(a b c) '(d a f c))
(A C)
```

10. Schreibe erneut die Funktion `invertieren`, diesmal jedoch nur unter der Verwendung der einfachen Grundprozeduren `defun`, `equal`, `car`, `cdr` usw. Es sollen keinerlei Funktionen zum Verketten von Listen verwendet werden.

6 Prozeduren und Funktionen (mit verschiedene Parametertypen)

Bislang wurden nur Funktionen definiert, die eine feste Anzahl von Parametern haben, die immer ausgewertet werden. Nun werden wir sehen, daß es auch andere Typen von Parametern und Prozeduren gibt, die mit variabler Anzahl an Argumenten aufgerufen werden können. Die zentrale Rolle bei der Verwaltung der Argumente spielen sogenannte Lambdalisten. Eine Lambdaliste definiert die Argumente, die von einer Funktion akzeptiert werden. (Genaueres zum λ-Kalkül siehe Kap. 10.2, S. 70.)

Lambdalisten beginnen mit den zwingend benötigten Argumenten, d.h. diese Argumente müssen bei jedem Aufruf der Funktion angegeben werden. Dieses sind genau die Argumente, die wir bisher kennengelernt haben. Anschließend werden die optionalen Argumente aufgeführt. Sie können, wie der Name schon sagt, beim Aufruf der Funktion angegeben werden, müssen es aber nicht. Bei der Definition von optionalen Argumenten können Initialwerte angegeben werden. Wird beim Aufruf der Funktion das optionale Argument nicht mit angegeben, so erhält es diesen Wert. Sollte kein Initialwert definiert worden sein, so wird ein beim Aufruf nicht angegebenes optionales Argument mit `nil` initialisiert. Optionale Parameter werden definiert, indem man ihnen `&optional` voranstellt.

Wir schauen uns nun ein Beispiel an, in dem ein optionaler Parameter benutzt wird. Es wird eine rekursive Funktion definiert, die die Reihenfolge der Elemente einer Liste invertiert.

```
>  (defun s-rev (liste &optional erg)
       (cond ((null liste) erg)
             (T (s-rev (cdr liste)
                       (cons (car liste) erg)))))
S-REV

>  (s-rev '(a s d f g h j))
(J H G F D S A)
```

Die Funktion `s-rev` benutzt den optionalen Parameter `erg`, um das Ergebnis aufzubauen. Beim Aufruf der Funktion mit der umzudrehenden Liste wurde das optionale Argument nicht mit angegeben, wodurch `erg` mit `nil` initialisiert wird. In der `t`-Klausel von `cond` ruft sich `s-rev` selbst auf (Rekursion).

Diesmal werden der Funktion zwei Parameter übergeben, der erste ist der `cdr` der Liste, womit der Rekursionsabbruch gewährleistet ist, denn irgendwann ist das Listenende erreicht und `(null liste)` wird wahr. Das zweite Argument besteht aus der bisherigen Ergebnisliste, an die hier der `car` der Liste mit `cons` vorne angefügt wird. Auf diese Weise wird die inverse Ergebnisliste erzeugt.

Ersetzt man die erste Zeile der Funktionsdefinition durch

```
(defun s-rev (liste &optional (erg '(ende)))),
```

so würde der obige Aufruf `(j h g f d s a ende)` liefern, denn dann wäre `erg` anfangs nicht NIL, sondern die Liste `(ende)` gewesen, an die vorne mit `cons` die Werte angehängt werden.

Eine andere typische Verwendung von optionalen Argumenten sind 'default'-Werte, eine Funktion die z.B. den Flächeninhalt eines Rechtecks berechnen soll, aber nur einen Wert bekommt, könnte dann den Flächeninhalt eines entsprechenden Quadrates liefern:

```
(defun flaeche (x &optional (y x))
    (* x y))
```

Den optionalen Argumenten folgt das `&rest` Argument. An das `&rest` Argument wird die Liste der Argumente gebunden, die in der Argumentenliste noch übrigbleiben, wenn die zwingend benötigten und die optionalen Argumente bereits entfernt worden sind. Sie werden z.B. benutzt, wenn von vornherein nicht klar ist, mit wievielen Argumenten eine Funktion aufgerufen wird. `setq` ist ein Beispiel für diese Technik: Normalerweise wird es mit zwei Argumenten benutzt, kann aber beliebig viele haben (z.B. `(setq a 10 b 20 c 30)`). Generell läßt sich ein &rest Argument auch als optionaler Parameter, der dann als Liste ausgewertet wird, implementieren. Dann müßten die Argumente aber schon beim Aufruf in eine Liste 'verpackt' werden. Obiges `setq`-Beispiel würde dann so aussehen: `(setq '(a 10 b 20 c 30))`. Das `&rest`-Argument kann also als abkürzende Schreibweise verwendet werden.

Ein weitere Typ von Parametern sind die Hilfsparameter. Diese werden durch Voranstellen von `'&aux'` definiert. Mit ihnen lassen sich Parameter definieren, die als lokale Hilfsvariablen benutzt werden. Solche Hilfsparameter werden benutzt, wenn beim Ablauf der Funktion Zwischenwerte gespeichert werden müssen. Daß die Parameter lokal sind, ist hier wichtig, denn globale Variablen bergen immer die Gefahr in sich, daß ihr aktueller Wert unbeabsichtigter Weise durch eine andere Prozedur überschrieben wird, was zu unerwünschten Nebeneffekten führen kann, deren Ursachen nur sehr schwierig oder gar nicht auszumachen sind. Hilfsparametern kann ebenfalls ein Initialwert zugewiesen werden. Wir betrachten nun eine Version des Invertierens einer Liste, die von einem solchen Hilfsparameter Gebrauch macht.

```
>  (defun inv (liste &aux hilf)
       (cond ((null liste) liste)
             ((null (cdr liste)) liste)
             (T (setq hilf (inv (cdr liste)))
                (cons (car hilf)
                      (inv (cons (car liste)
                                 (inv (cdr hilf))))))))
```

Die Wirkungsweise dieser Funktion wird deutlicher anhand der Liste `(1 2 3 4 5 6 7)`. Mit dem `setq` erhält `hilf` den Wert `(7 6 5 4 3 2)`, nämlich das Inverse des `cdr`. Diese Liste wird zwischengespeichert, da sie zweimal benötigt wird und die Berechnung recht aufwendig ist. Danach wird mit dem `cons` die 7 mit dem Inversen von `(1 2 3 4 5 6)` verbunden, aber dieses wird gerade von `(inv (cons (car liste) (inv (cdr hilf))))` geliefert. Denn (cdr hilf) ist `(6 5 4 3 2)`. Diese Liste wird also invertiert und vorne der `car` von `liste` angehängt, was ja gerade das erste Element ist. Also wird das Inverse von `(1 2 3 4 5 6)` gebildet, an das vorne die 7 gehängt wird. Mit der Liste `(1 2 3 4 5 6)` verfährt die Funktion genauso…

Den `&rest` Argumenten können `&key` Argumente folgen. Wenn ein Schlüsselwort der Funktion übergeben wird, erscheint ein Paar von Werten in der Argumentenliste. Der erste Ausdruck des Paares sollte ein Schlüsselwortsymbol sein (d.h. ein Symbol, das mit : beginnt). Der Wert des zweiten Ausdrucks ist dann der Wert des Schlüsselwortsymboles. Dadurch werden Argumente nicht mehr durch ihre Position unterschieden, sondern durch Schlüsselwortsymbole.

Beispiel:

```
> (defun keylist (a &key x y z)
(list a x y z))
> (keylist 1 :y 2 )
(1 NIL 2 NIL)
> (keylist 1 :y 3 :x 2)
(1 2 3 NIL)
```

Die komplette Syntax für Lambdalisten:

```
(<arg>...
[&optional [<oarg> | (<oarg> [<init>] [<svar>]])]...]
[&rest <rarg>]
[&key
[<karg> | ([<karg> | (<key> <karg>)]
[<init> [<svar>]])]
... &allow-other-keys]
[&aux
[<aux> | (<aux> [<init>])]
...])
```

wobei:

<`arg`> erforderliches Argumentsymbol
<`oarg`> &optional Argumentsymbol
<`rarg`> &rest Argumentsymbol
<`karg`> &key Argumentsymbol
<`key`> Schlüsselwortsymbol
<`aux`> Hilfsvariablensymbol

<`init`> Initialisierungsausdruck
<`svar`> ist ein supplied-p Variablensymbol

Die Reihenfolge der verschiedenen Parametertypen muß auch bei der Definition von Prozeduren und Funktionen eingehalten werden. Das bedeutet z.B.: die zwingend anzugebenen Parameter stehen immer am Anfang, die Hilfsvariablen werden immer als letztes in die Parameterliste einer Funktionsdefinition eingetragen, usw.

Einige Grundprozeduren besitzen noch eine ganz andere Art von Parametern. Und zwar kann man ihnen mittels eines Schlüsselwortes mitteilen, welche Prozedur sie als Vergleichstest verwenden. Im Normalfall wird `eql` als Vergleichstest benutzt, möchte man allerdings einen anderen Test benutzen, so kann man dieses der Prozedur mittels des Schlüsselwortes `:test` mitteilen.

```
>  (member '(a b) '((a b) (c d)))
nil
>  (member '(a b) '((a b) (c d)) :test 'equal)
((A B) (C D))
```

Die Möglichkeit, den Vergleichsoperator anzugeben, besteht bei einer Reihe weiterer Grundprozeduren wie `delete`, `remove`, `assoc` usw.

7 Schleifenkonstrukte und Iteration

Häufig ist es in einem Programm notwendig, Iterationen durchzuführen. Lisp stellt eine ganze Reihe von Prozeduren zur Verfügung, Probleme auch iterativ und nicht nur rekursiv zu bearbeiten.

7.1 Lokale Variablen und sequentielle Ausführung

Eine einfache Möglichkeit, lokale Variablen zu initialisieren und mehrere Funktionen an einer Stelle nacheinander aufzurufen, bietet `let`. Die Funktionsweise machen wir uns an einem Beispiel klar. Angenommen wir bekommen ein Element und einen Beutel und sollen das Element in den Beutel packen, sofern es dort nicht schon vorhanden ist. Nur wissen wir nicht, welcher der beiden Parameter nun der Beutel und welcher das Element ist. Das Problem lösen wir nun mit `let` auf recht 'elegante' Art.

```
(defun hinzufuegen1 (x y)
    (let ((element x)      ; element binden und init.
          (beutel y))      ; beutel binden und init.
      (cond ((and (atom beutel) (listp element))
             (setq element y)
             (setq beutel x)))
      (cond ((member element beutel) beutel)
            (T (cons element beutel)))))
```

Unter einem Beutel (`bag`) wollen wir hier eine 'Multimenge' verstehen, d.h. eine Menge, die aus Elementen besteht, wobei die Reihenfolge, in der die Elemente in den Beutel gelangen, nicht interessiert. Der Unterschied zu einer Menge besteht nun darin, daß Elemente mehrfach vorkommen können.

Zuerst werden die lokalen Variablen `element` und `beutel` gebunden und mit `x` bzw. `y` initialisiert. Danach wird getestet, ob die Zuweisung falsch war. Denn `atom` prüft, ob `beutel` ein Atom ist und `listp` ('list predicate') prüft, ob `element` eine Liste ist. Liefern beide `t`, so liefert auch `and` `t`. Dann wurde aber gerade die Zuweisung von `beutel` und `element` falsch gemacht und wir vertauschen die Zuweisungen mittels `setq`. Im zweiten `cond` wird mit `member` getestet, ob sich `element` schon im `beutel` befindet, falls ja, wird der `beutel` so zurückgegeben, falls nein, wird das `element` zuerst in den `beutel` gepackt. In `let` werden die Parameter parallel gebunden, d.h. in obigem Beispiel könnte `element` nicht dazu benutzt werden, `beutel` zu definieren. Wollte man dieses aber dennoch machen. so müßte man `let*` verwenden. Diese Grundprozedur funktioniert genauso wie `let`, jedoch mit dem Unterschied, daß die Parameter in `let*` seriell zugewiesen werden. Seriell bedeutet hier, daß eine Variable, der schon ein Wert zugewiesen worden ist, dazu benutzt werden kann, einer nach ihr stehenden Variablen einen Wert zuzuweisen. Bei paralleler Zuweisung ist dieses nicht möglich. Im folgenden nun ein Beispiel, das den Unterschied verdeutlicht:

```
(defun test (x)
    (let ((x (+ x 2))
          (y (+ x 1)))
         y))

> (test 3)
4
```

```
(defun test* (x)
    (let* ((x (+ x 2))
           (y (+ x 1)))
          y))

> (test* 3)
6
```

Funktionen, die `t` oder `nil` liefern, werden auch *Prädikate* genannt. Solche Prädikate haben in Lisp häufig die Endung `p`, wie z.B. `listp`. Jedoch ist diese Endung der Prädikatsnamen nicht konsequent durchgehalten worden. So gibt es, wie bereits gesehen, das Prädikat `atom`, welches konsequenterweise `atomp` heißen müßte.

Das erstes Beispiel mit dem Beutel und dem Element kann auch gelösen werden, indem sofort sichergestellt wird, daß die Variablenzuweisungen richtig erfolgen.

```
(defun hinzufuegen2 (x y)
    (let ((element (cond ((and (atom x) (listp y)) x)
                         (T y)))
          (beutel  (cond ((and (atom x) (listp y)) y)
                         (T x))))
         (cond ((member element beutel) beutel)
               (T (cons element beutel)))))
```

Nun scheint es überflüssig zu sein, zweimal exakt die gleiche `cond`-Abfrage zu machen und nur `x` und `y` zu vertauschen. Möchte man nun das Ergebnis des ersten Vergleichs nutzen, so muß man `let*` anstelle von `let` verwenden. Denn `let*` behandelt seine Variablen nacheinander. Folgende Version von `hinzufuegen` leistet also das gleiche:

```
(defun hinzufuegen3 (x y)
    (let* ((element (cond ((and (atom x) (listp y)) x)
                          (T y)))
           (beutel  (cond ((equal element x) y)
                          (T x))))
          (cond ((member element beutel) beutel)
                (T (cons element beutel)))))
```

`equal` ist eine der Grundprozedur, die testen, ob die angebenen Ausdrücke gleich sind. Da `let*` zuerst den Initialausdruck auswertet und an `element` zuweist, kann man im Initialausdruck von `beutel` bereits `element` benutzen.

Bei `defun` und `let` werden alle Ausdrücke, bis auf den letzten, ausschließlich zum Erzeugen von Nebeneffekten ausgewertet. Möchte man die Reihenfolge der Auswertung beibehalten, jedoch den Wert des ersten Ausdrucks zurückgeben, so kann man dieses mit der Grundprozedur `prog1` erreichen.

```
>  (prog1 (setq a 1) (setq b 2) (setq c 3))
1
```

Analog dazu liefert `prog2` den Wert seines zweiten Ausdruckes.

```
>  (prog2 (setq a 1) (setq b 2) (setq c 3))
2
```

Und `progn` liefert den Wert seines letzten Ausdruckes, was allerdings selten benutzt wird, da sich dieser Effekt auch mit `let` erzielen läßt.

Auf die Verwendung von `prog` sollte man nach Möglichkeit vollkommen verzichten, denn es läßt sich eigentlich immer durch Verwendung anderer Prädikate ersetzen, die es teilweise zur Zeit der Entstehung von `prog` noch nicht gegeben hat.

7.2 Schleifen

Häufig verwendete Schleifenkonstrukte sind `do`-Schleifen. Die allgemeine Syntax für `do`-Schleifen lautet:

```
(do ((<varible-1> <initialwert-1> <aktualisierungsregel-1>)
    (<varible-2> <initialwert-2> <aktualisierungsregel-2>)
    ...
    (<varible-n> <initialwert-n> <aktualisierungsregel-n>))
    (<test> [Zwischenterme] <resultat>)
    <rumpf>
)
```

Alle Initialisierungsausdrücke (`initialwert`) werden vor der Parameterbindung und der Wertzuweisung ausgewertet. Entsprechend werden alle Aktualisierungsausdrücke (`aktualisierungsregel`) vor erneuter Wertzuweisung ausgewertet. `do` behandelt seine Variablen parallel.

Ein typisches Beispiel für die Verwendung von `do` ist beispielsweise die Berechnung der y-ten Potenz einer Zahl x.

```
(defun exponent1 (x y)
    (do ((resultat 1)           ; binden und init.
         (exp y))               ; binden und init.
        ((zerop exp) resultat)  ; Abbruchbedingung
        (setq resultat (* resultat x))
        (setq exp (- exp 1))))
>  (exponent1 3 4)
81
```

In diesem Fall haben wir die Hauptarbeit in den Rumpf der Schleife gelegt. Nutzt man allerdings die Möglichkeit aus, Aktualisierungsregeln anzugeben, so kann man die Funktion auch mit komplett leerem Schleifenrumpf schreiben.

```
(defun exponent2 (x y)
    (do ((resultat 1 (* resultat x))
         (exp y (- exp 1)))
        ((zerop exp) resultat)))  ; Abbruchbedingung
```

Zuerst wird hier `resultat` mit 1 initialisiert und `exp` mit `y`. Wird die Abbruchbedingung nicht erfüllt, so wird im nachsten Schleifendurchlauf `resultat` auf den Wert von sich selbst multipliziert mit `x` gesetzt. Analog wird `exp` um 1 reduziert. Dieser Vorgang wird solange wiederholt, bis die Abbruchbedingung erfüllt ist, d.h. `exp` null ist. Dann wird `resultat` ausgegeben. `zerop` ist ein Prädikat, das `t` liefert, wenn sein Argument Null ist und sonst `nil`.

Eine weitere hier zu erwähnende Grundprozedur ist `do*`. `do*` verhält sich zu `do` wie `let*` zu `let`.

Oft möchte man eine Liste Element für Element durchgehen oder eine Aktion mit bestimmter vorgegebener Häufigkeit wiederholen. Dieses könnte man sicherlich mit einer geeigneten Anwendung von `do` erreichen, aber einfacher geht es noch, wenn man die Grundprozeduren `dolist` bzw. `dotimes` verwendet.

```
(defun ausgabe (liste)
    (dolist (e liste (terpri))
       (prin1 e)
       (princ " "))) ; String, bestehend aus einem
                     ; Leerzeichen

>  (ausgabe '(a c v d s))
A C V D S       ; Ergebnis von 'AUSGABE'
nil             ; Returnwert von 'TERPRI'
```

Der Hilfsvariable `e` wird in der `dolist`-Schleife nacheinander jedes Element von `liste` zugewiesen, welches dann mit `prin1` ausgegeben wird. Ist das Ende der Liste erreicht, so wird `fertig` ausgegeben. Die Verwendung von `prin1` bewirkt nach der Ausgabe seines Argumentes im Gegensatz zu `print` keinen Zeilenvorschub. `princ` funktioniert wie `prin1` mit dem Unterschied jedoch, daß es die Doppelanführungszeichen bei der Ausgabe von Strings wegläßt. Ersetzt man in obigem `princ` durch `prin1`, so sieht die Ausgabe folgendermaßen aus:

```
"A" "C" "V" "D" "S" "FERTIG"
```

`dotimes` hat eine sehr ähnliche Syntax wie `dolist`.

```
(defun zahlen (bis)
    (dotimes (x bis 'fertig)
      (prin1 x)
      (princ " ")))  ; Leerzeichen

>  (zahlen 10)
0 1 2 3 4 5 6 7 8 9
nil
```

Wie man an dem Beispiel sieht, werden der Hilfsvariablen `x` (hier definiert als Hilfsparameter) nacheinander die Werte 0. . . 9 zugewiesen und nicht 1. . . 10, wie man vielleicht auch vermuten könnte.

Eine etwas altertümliche Art der Iteration bietet die Grundprozedur `prog`. Die Funktionsweise von `prog` betrachten wir an einem weiteren Beispiel zur Berechnung der Potenz.

```
(defun exponent4 (x y)
    (prog (resultat exp)     ; Parameter mit NIL binden
          (setq resultat 1)  ; initialisieren
          (setq exp y)       ; initialisieren
      loop                   ; Sprungmarke
          (cond ((zerop exp)          ; Ende-Test
                 (return resultat)))  ; Abbruch
```

```
          (setq resultat (* x resultat))
          (setq exp (- exp 1))
          (go loop)))                  ; Wiederholung
```

Es tauchen bei dieser Definition zwei neue Grundprozeduren auf. `go` sorgt dafür, daß die Auswertung nach der angegebenen Marke (label) fortgesetzt wird. Mit `return` wird das `prog`-Konstrukt mit dem Returnwert `resultat` verlassen. Weil danach auch die Prozedur fertig ist, wird `resultat` zum Returnwert von `exponent4`. Der Name der Sprungmarke kann frei gewählt werden, nur vom System benutzte Namen sollte man nicht verwenden. `go` kann die Kontrolle an eine Sprungmarke in einem lexikalisch übergeordneten Schleifenkonstrukt (`prog`, `do`, `dotimes` usw.) übergeben, aber nicht an eine Marke in einem lexikalisch untergeordneten. `return`, `go` und Sprungmarken können also in den oben genannten Schleifenkonstrukten verwendet werden, jedoch nicht außerhalb solcher Konstrukte. Die Verwendung von `go` und `return` ist in *allen* Fällen überflüssig und kann durch Schleifenkonstrukte ersetzt werden, es gilt dasselbe wie für `goto` in Pascal oder C.

7.3 Die map-Famile

Möchte man eine Funktion immer wieder auf eine ganze Reihe von Objekten anwenden, so läßt sich diese Art der Iteration auf einfache Weise mit `mapcar` durchführen. `mapcar` erhält als erstes Argument die Funktion und als weitere Argumente die Liste der Objekte, die übergeben werden sollen.

```
>  (mapcar print '(a b c))
A
B
C
(A B C)    Wert der Funktion mapcar

>  (mapcar '+ '(1 2 3) '(5 4 6) '(2 5 9))
(8 11 18)
```

Für die von `mapcar` zu verwendende Funktion kann ein Symbol, dessen Wert die Funktionsdefinition ist, gequotet oder ungequotet angegeben werden, oder aber die Funktionsdefinition in Form eines Lambdaausdruckes selbst (siehe Kapitel 10.2, S. 70). `mapcar` veranlaßt die wiederholte Anwendung von `'+'`. Und zwar werden zuerst die jeweils ersten Zahlen der drei Listen addiert, danach dann die jeweils mittleren und zum Schluß die jeweils letzten Zahlen. Die einzelnen Ergebnisse werden zusammen in Form einer Liste ausgegeben. Die Anzahl der Funktionsanwendungen wird bestimmt durch die kürzeste Liste; unterschiedlich lange Listen führen nicht zu einer Fehlermeldung.

```
>  (mapcar '+ '(1 2 3) '(6) '(2 5 9))
(9)
```

Ganz Analog zu `mapcar` funktioniert `mapc`, nur mit dem Unterschied, daß `mapc` nicht die Liste der Einzelergebnisse zurückgibt, sondern sein zweites Argument (die Liste mit den ersten Argumenten der anzuwendenden Funktion).

```
>  (mapc '+ '(1 2 3) '(5 4 6) '(2 5 9))
(1 2 3)
```

Eine ähnlich wie `mapcar` arbeitende Grundprozedur ist `maplist`. Sie wendet jedoch eine Funktion sukzessive auf die jeweiligen Restlisten und nicht auf die einzelnen Elemente der angegebenen Listen an.

```
>  (maplist print '(a b c))
(A B C)
(B C)
(C)
((A B C) (B C) (C))
```

Analog zu `mapc` funktioniert `mapl`, denn `mapl` gibt ebenfalls die erste Argumentenliste zurück.

```
>  (mapl print '(a b c))
(A B C)
(B C)
(C)
(A B C)
```

Wie eine Kombination der Prozeduren `append` und `mapcar` wirkt `mapcan` (mapcan ...) $\equiv$ (append(mapcar ...)). Betrachtet man folgendes Beispiel, so ist die Ergebnisliste vielleicht nicht in der Form wie man sie gerne haben möchte.

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

Anstelle einer Liste von Listen, möchte man z.B. eine Liste bestehend aus Atomen als Rückgabewert erhalten. Dieses erreicht man hier durch die Anwendung von `mapcan` anstelle von `mapcar`.

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

Möchte man die Elemente einer Liste als Argumente für eine Funktion verwenden, so ist dieses nicht ohne weiteres möglich. Aber es gibt die Grundprozedur `apply`, welche genau dieses leistet. Sie erhält als erstes Argument die anzuwendende Funktion und als zweites Argument eine Liste. Diese Liste besteht aus den Argumenten, auf die die Funktion angewendet werden soll.

```
>  (apply '+ '(1 2 3 4 5))
15
```

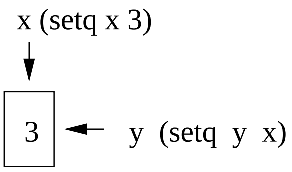
Gleichheitsstests

Bevor wir nun zu den Übungen übergehen, sollten wir noch die Unterschiede zwischen den verschiedenen Gleichheitstest klären. In Lisp gibt es vier verschiedene Gleichheitstests. Nämlich `equal`, `eql` und `eq` und zum Vergleichen von Zahlen noch `=`. Wie schon erwähnt ist `equal` der am häufigsten gebrauchte. Aber auch für die anderen gibt es durchaus Anwendungen. Als nächstes nun ein paar Beispiele, die die unterschiedlichen Wirkungsweisen verdeutlichen.

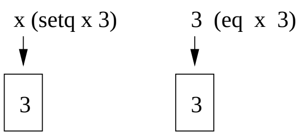
```
>  (setq x '(a b c) y '(d e f))
(D E F)
>  (eq (append x y) (append x y))
nil
>  (eql (append x y) (append x y))
nil
>  (equal (append x y) (append x y))
T

>  (eq 4 (+ 2 2))
nil
>  (eql 4 (+ 2 2))
T
>  (equal 4 (+ 2 2))
T

>  (setq x 3)
3
>  (setq y x)
3
>  (eq x y)
T
>  (eq x 3)
nil
>  (eql x 3)
T
>  (equal x 3)
```

`T`

`eq` vergleicht einfach nur die Zeiger seiner Argumente. Das erklart, warum beim ersten Beispiel `nil` herauskommt. Denn jeder Aufruf von `append` erzeugt eine Kopie seiner Argumente und verknüpft diese. Deshalb sind die Zeiger, die von den `append` Aufrufen zurückgegeben werden, verschieden und das Ergebnis des Vergleichs mit `eq` liefert `nil`. `eql` unterscheidet sich nur beim Vergleich von Atomen von `eq`, ansonsten verhält es sich beim Vergleich von komplexen Strukturen (keinen Atomen) wie `eq`. Daher liefert `eq` auch `T`. Nur `equal` liefert beim ersten Beispiel `t`, da es die Zeiger verfolgt und die Strukturen explizit miteinander vergleicht. Beim zweiten und dritten Beispiel wird der Unterschied zwischen `eq` und `eql` verdeutlicht. Die Zuweisung kann man sich als Schaubild vorstellen.



An dem Bild sieht man, daß `x` und `y` auf exakt diesselbe Speicherstelle zeigen. Daher liefert `(eq x y) t` (und ebenso natürlich auch `eql` und `equal`). Bei `(eq x 3)` sieht das Ergebnis allerdings etwas anders aus. Da `3` ein Atom ist, wird beim Vergleich eine andere Speicherstelle benutzt.



Die Speicherstelle, auf die das `x` zeigt und diejenige, welche durch `(eq x 3)` für das Atom `3` erzeugt wird, sind nicht dieselben, daher liefert `eq` hier `nil`. Im Gegensatz dazu liefert `eql t`, da es nicht für jedes Atom einen neuen Speicherplatz benutzt. Ganz krass erscheint das Beispiel:

```
>  (eq 3 3)
nil
```

Denn hier werden für die beiden Atome `3` vom System zwei unterschiedliche Speicherstellen benutzt. Da `eq` nur die Zeiger vergleicht, benötigt es in vielen Fällen am wenigsten und `equal` am meisten Laufzeit. In einigen Fällen ist das Laufzeitverhalten bei allen drei Vergleichsprozeduren gleich. Und zwar dann, wenn die Zeiger der zu vergleichenden Objekte gleich sind. Viele Grundprozeduren benutzen einen Vergleichstest bei ihrer Ausführung. Ist nichts anderes angegeben, so wird immer `eql` genommen. Dieses reicht meist aus und hat den Vorteil, daß es in vielen Fällen schneller ist als `equal`.

7.4 Übungen:

1. Schreibe unter Verwendung von `do` eine iterative Version der Funktion `mein-member`.

2. Zwei Listen mit unbekannter Länge sollen destruktiv (`setf`) miteinander verknüpft werden, und zwar so, daß stets an die längere Liste die kürzere Liste hinten angehängt wird. Schreibe zur Lösung dieser Aufgabe eine Funktion, die `let` für die Zuweisung und `if` für die Bedingungsabfragen verwendet. In der Funktion sollen die Längen der Listen nur einmal ermittelt werden.

   ```
   >  (appendlk '(a b c) '(1 2))
   (A B C 1 2)
   >  (appendlk '(1 2) '(a b c))
   (A B C 1 2)
   ```

   Hinweis: Die Funktion `lastlist` könnte bei der Lösung sehr hilfreich sein.

3. Selbes Problem wie unter 2., jedoch soll diesmal die Wertzuweisung auf jeden Fall nur einmal, also gleich richtig erfolgen. Das bedeutet, daß

schon bei (bzw. vor) der Parameterübergabe abgefragt werden muß, welche Liste welcher Variablen zuzuweisen ist. Auch in diesem Fall sollen die Längen der Listen nur ein einziges Mal bestimmt werden.

4. Schreibe eine Funktion, die als Eingabe eine Liste bekommt und dazu alle möglichen Punktpaare erzeugt, die sich aus der Kombination von je zwei Listenelementen erzeugen lassen.

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

(Auf die Reihenfolge der Punktpaare kommt es dabei nicht an.)

5. Es sollen zwei (gleichlange) Listen von Zahlen miteinander verglichen werden. berlege, wie sich diese Aufgabe in Lisp 'möglichst' einfach lösen läßt. Als Eingabe sollen zwei Zahlenlisten der folgenden Art dienen:

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

Und die Ausgabe soll in etwa folgendermaßen aussehen:

```
(3 IST GROESSER ALS 1)
(2 IST GLEICH 2)
(4 IST GROESSER ALS 3)
(5 IST GROESSER ALS 4)
```

8 Destruktive Listenprozeduren

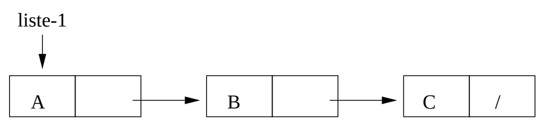
Bisher wurden Funktionen vorgestellt, die eine oder mehrere Listen als Argumente akzeptieren und eine *neue* Liste zurückliefern: `reverse` z.B. liefert eine Liste, in der die Elemente der obersten Ebene in umgekehrter Reihenfolge erscheinen. Dadurch wird die alte Liste aber nicht verändert!

```
>  (setq a '(1 2 3 4))
(1 2 3 4)
>  (reverse a)
(4 3 2 1)
>  a
(1 2 3 4)
```

8.1 Einfache destruktive Prozeduren

Nun kann es Situationen geben, in denen dieser Effekt unerwünscht ist und eine Prozedur[13] die Speicherrepräsentation ihrer Argumente ändern soll.

Um zu verstehen, wie eine solche Veränderung geschieht, muß man sich über die interne Speicherrepräsenation von Listen im Klaren sein. Das schon früher angesprochene Konzept der 'cons-Zellen' (Kap.5.1,S. 22) wird hier sehr wichtig. Wir betrachten die Liste `(a b c)` mit dem Namen `Liste-1`. Die interne Zellenorganisation des Speichers kann man sich folgendermaßen vorstellen:



('/' steht für `nil` und zeigt das Ende der Liste an)

Dabei ist `liste-1` der Name der Liste hinter dem sich ein Zeiger verbirgt, der auf die erste cons-Zelle zeigt, denn `liste-1` ist nichts anderes als ein Symbol, dessen Wert eine Liste ist. So eine Zelle besteht aus zwei Teilen, nämlich einem Teil, der den Wert, im ersten Fall also `a` enthält, und einem Teil, der einen Zeiger auf die nächste Zelle enthält, die genauso aufgebaut ist. Das Ende einer so dargestellten Liste ist erreicht, wenn der Teil einer Zelle, der sonst einen Zeiger auf die nächste Zelle enthält, einen bestimmten Wert,

[13]Keine Funktion kann dies leisten, weil die Veränderung der Speicherrepräsentation ein Seiteneffekt ist. Also müssen destruktive Operationen auf Listen Prozeduren sein.

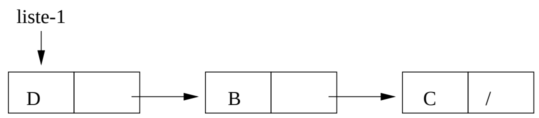
nämlich `nil`, enthält, wodurch das System erkennt, daß hier die Liste zuende ist.

Mit den bisher vorgestellten Funktionen und Prozeduren war es nicht möglich, einzelne Inhalte von cons-Zellen zu *verändern*, es konnten lediglich neue cons-Zellen und Listen erzeugt werden. Die destruktiven Listenprozeduren übernehmen die Aufgabe, Teile von Listen, d.h. Inhalte von cons-Zellen, direkt zu verändern (und dabei die alten Inhalte zu zerstören, daher 'destruktiv').

Um z.B. das `a` in `liste-1` zu ersetzen, kann man die Prozedur `rplaca` verwenden ('replace car'). Diese erhält als erstes Element die zu verändernde Liste und als zweites den neuen Ausdruck. Sie liefert als Ergebnis die als Nebeneffekt dauerhaft veränderte Liste.

```
>  liste-1
(A B C)
>  (rplaca liste-1 'd)
(D B C)
>  liste-1
(D B C)
```

Wie man anhand der letzten Eingabe sieht, ist nicht etwa eine Kopie der `liste-1` verändert und zurückgegeben worden, sondern es ist tatsächlich die `liste-1` selbst verändert worden. Sie sieht also nun folgendermaßen aus:



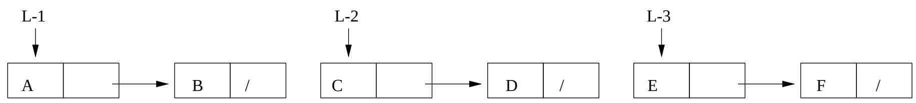
Die Prozedur `rplacd` ('replace cdr') funktioniert analog zu `rplaca`, jedoch ersetzt sie den `cdr` einer Liste.

```
>  (rplacd liste-1 '(e f))
(D E F)
```

Eine weitere Prozedur, die bei oberfächlicher Anwendung zu überraschenden Ergebnissen führen kann, ist `nconc`. Diese Prozedur verkettet Listen, wobei die Speicherrepräsentation der Listen verändert wird. Um uns die Wirkungsweise dieser Prozedur näher ansehen zu können, erzeugen wir uns als erstes drei Listen.

```
>  (setq L-1 '(a b) L-2 '(c d) L-3 '(e f))
(E F)
```

Mit dieser Eingabe haben wir uns die drei Listen `L-1`, `L-2`, und `L-3` erzeugt, denn `setq` betrachtet seine Argumente jeweils paarweise. Wir können uns die Speicherrepräsentation also wieder folgendermaßen vorstellen:



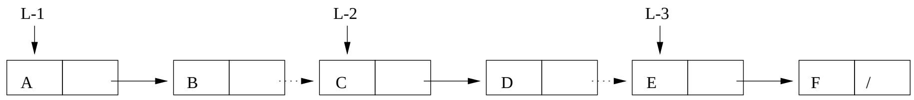
Diese drei Listen werden wir nun mit `nconc` verknüpfen.

```
>  (nconc L-1 L-2 L-3)
(A B C D E F)
```

Die Ausgabe ist nicht weiter verwunderlich. Sie würde ebenfalls so aussehen, wenn wir `append` auf die Listen angewendet hatten. Nun schauen wir uns die ursprünglichen Listen an:

```
>  L-1
(A B C D E F)

>  L-2
(C D E F)

>  L-3
(E F)
```

Wie wir sehen, sind die Listen, zumindest `L-1` und `L-2`, verändert worden. Dieses wäre bei der Verwendung von `append` nicht passiert, da `append` mit Kopien der Originallisten arbeitet. Was nun wirklich passiert ist, sieht man am besten an einem neuen cons-Zellenbild:



In diesem Bild sind im Gegensatz zum obigen Bild die gestrichelten Pfeile neu hinzugekommen. `nconc` bewirkt, daß der Teil der letzten Zelle der ersten Liste, der vorher `nil` enthielt, nun einen Zeiger auf die erste Zelle der zweiten Liste besitzt. Und analog dazu besitzt nun der entsprechende Teil der letzten Zelle einen Zeiger auf die erste Zelle der Liste `L-3`. Dieses erklärt nun auch die Veränderungen der Listen `L-1` und `L-2`, und auch, warum `L-3` unverändert geblieben ist. Verfolgt man nun den Zeiger von `L-1`, so zeigt dieser auf die Zelle, die `A` enthält. Deren Zeiger zeigt auf die Zelle, die `B` enthält, usw. Dieses Verfolgen des Zeigers bricht erst ab, wenn ein `nil` (/) erreicht wird, d.h. für `L-1` wird also der Zeiger von der Zelle mit `A` bis zu der mit `F` verfolgt, für `L-2` von `C` bis `F` und für `L-3` von `E` bis `F`.

Eine weitere mit Vorsicht zu genießende Prozedur ist `delete`. Diese erhält als erstes das zu löschende Element und als zweites die Liste, in der das Element gelöscht werden soll. `liste` habe den Wert `(a a d a s a e)` und daraus soll das `a` gelöscht werden.

```
>  (delete 'a liste)
(D S E)
```

Das 'delete' liefert also die richtige Ergebnisliste, die 'liste' selber allerdings sieht folgendermaßen aus:

```
>  liste
(A A D S E)
```

Die Vorkommen von 'a' werden also destruktiv aus 'liste' gelöscht, jedoch nicht die am Anfang der Liste stehenden. Diese werden bei der Ausgabe einfach übersprungen und sind daher in 'liste' noch immer vorhanden.

8.2 place forms

Die allgemeinste Prozedur, mit der man Strukturen verändern kann, ist `setf`. Sie erhält als erstes Argument eine Zugriffsform (place form) und als zweites einen Wert bzw. einen Ausdruck, der den Wert liefert. Unter einer Zugriffsform versteht man eine Prozedur bzw. Funktion, mit der man auf einen bestimmten Teil einer Struktur zugreifen kann. Häufig werden hierbei Funktionen wie `car`, `cdr` oder `nth` verwendet. Als Zugriffsform können noch wesentlich mehr Grundprozeduren verwendet werden, die wir später noch kennenlernen werden. Hier nun einige Beispiele für die Verwendung von `setf`:

```
> (setq l '(a b c))
(A B C)
> (setf (car l) 5))
5
> l
(5 B C)
```

In obigem Beispiel ist `(car l)` die Zugriffsform und `5` der Wert. Genauso funktioniert auch

```
> (setf (cdr l) '(6 7))
(6 7)
> l
(5 6 7)
```

Ein weiterer Punkt, der bei der Verwendung von destruktiven Prozeduren beachtet werden muß, ist die Erzeugung von zirkulären Listen. Das sind Listenstrukturen, die so verändert worden sind, daß sie Zeiger auf sich selbst enthalten, d.h. sich also selbst 'in den Schwanz beißen'. Das folgende Beispiel sollte nicht ausprobiert werden, sofern man das System nicht neu starten möchte.

```
> (setf (cdr l) l)
laeuft ununterbrochen
```

8.3 Differenzlisten

Bei der Verwendung von großen Listen, stellt man sehr schnell fest, daß Funktionen, die nicht auf den `car` bzw. `cdr` einer Liste zugreifen, sehr viel Rechenzeit benötigen. Ein Beispiel dafür ist `append`. Um zwei Listen miteinander zu verketten, ist es nötig, das Ende, d.h. das letzte Element der Liste zu kennen. Dazu muß man aber die Liste Element für Element durchgehen und testen, ob das Ende bereits erreicht ist, wodurch gerade der große Rechenaufwand entsteht. Will man also ein solches Problem effizienter lösen, so bietet sich die Verwendung von *Differenzlisten* an.

Eine Differenzliste kann man sich als eine Liste bestehend aus zwei Listen vorstellen, wobei die Differenz der beiden Listen die eigentliche Liste darstellt. Wir wollen die Liste `(a b c)` als Differenzliste darstellen. Mögliche Darstellungsweisen dafür sind:

$$\begin{bmatrix} \texttt{(a b c d e f)} \\ \texttt{(d e f)} \end{bmatrix}, \begin{bmatrix} \texttt{(a b c d)} \\ \texttt{(d)} \end{bmatrix} \mathit{oder} \begin{bmatrix} \texttt{(a b c)} \\ \texttt{()} \end{bmatrix}$$

(Dieses ist allerdings noch kein Lispcode)

Hat man nun noch eine zweite Liste `(1 2 3)`, kann man sie folgendermaßen als Differenzliste darstellen:

$$\begin{bmatrix} \texttt{(1 2 3)} \\ \texttt{()} \end{bmatrix}$$

Um nun die zwei Differenzlisten zu verketten, identifiziert man den zweiten Teil (die Restliste) der ersten Differenzliste mit dem ersten Teil der zweiten. Dann bildet der erste Teil der ersten Differenzliste und der zweite Teil der zweiten Differenzliste die neue verkettete Differenzliste. Was also passiert, ist das folgende:

$$dliste-1: \begin{bmatrix} \texttt{(a b c)} \\ \hfill \texttt{()} \end{bmatrix} dliste-2: \begin{bmatrix} \texttt{(1 2 3)} \\ \hfill \texttt{()} \end{bmatrix}$$

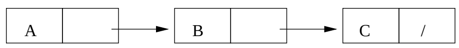
So passen die beiden Listen noch nicht zusammen, denn `()` von `liste-1` kann man nicht mit `(1 2 3)` von `liste-2` identifizieren. Ändert man aber die Darstellung der ersten Liste wie folgt, so lassen sich die beiden Differenzlisten sehr leicht verketten:

$$dliste-1: \begin{bmatrix} \texttt{(a b c 1 2 3)} \\ \hfill \texttt{(1 2 3)} \end{bmatrix} dliste-2: \begin{bmatrix} \texttt{(1 2 3)} \\ \hfill \texttt{()} \end{bmatrix}$$

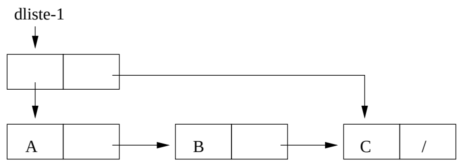
$$dliste-3: \begin{bmatrix} \texttt{(a b c 1 2 3)} \\ \hfill \texttt{()} \end{bmatrix}$$

Die `dliste-1` repräsentiert nun immer noch die Liste '(a b c)', allerdings lassen sich nun der zweite Teil von `dliste-1` und der erste Teil von `dliste-2` identifizieren und es ergibt sich die Ergebnisliste `dliste-3`.

Insofern ist in Lisp die Vorstellung für Differenzlisten analog zu der in Prolog. Allerdings lassen sich Differenzlisten in Lisp nicht ganz so, wie oben dargestellt, verwalten. Warum das so ist, das machen wir uns wieder an einem cons-Zellenbild klar. Wir möchten Liste `(a b c)` als Differenzliste darstellen und zwar so, daß der zweite Teil der Differenzliste leer ist. Dann sieht der erste Teil so aus



und der zweite Teil der Differenzliste sollte aus der zweiten cons-Zelle bestehen. Auf diese Weise hat man dann die Möglichkeit das `nil` der letzten `cons-Zelle` durch einen Zeiger auf eine andere Zelle zu ersetzen und so die Liste zu verlängern. Das ist aber gerade die Ursache dafür, daß der zweite Teil der Differenzliste nicht die leere Liste `()` sein kann, denn diese ist keine cons-Zelle und kann somit nicht direkt angesprochen werden, d.h. es kann kein Zeiger direkt auf das `nil` dieser Liste zeigen. Um dieses Problem zu lösen, nimmt man als den zweiten Teil der Differenzliste eine Liste, die nur aus der letzten cons-Zelle der darzustellenden Liste besteht. Genauer gesagt man nimmt keine Kopie, sondern genau den Zeiger, der auf die entsprechende Speicherstelle zeigt (der gestrichelte Pfeil). Die `dliste-1` kann man sich also folgendermaßen vorstellen:



Dadurch hat man die Möglichkeit gewonnen, sehr schnell etwas an eine Liste anzuhängen. Denn nun ist es nicht mehr nötig, alle Listenelemente abzuklappern, um das letzte Element zu finden. In Lisp würde eine solche Liste etwa so aussehen `((a b c) (c))`.

Aber natürlich kann man nicht einfach eine solche Liste eintippen, um sich eine Differenliste zu erzeugen, denn die beiden `c` einer so eingetippten Liste sind zwar die gleichen Buchstaben, aber sie stehen nicht an exakt derselben Speicherstelle.

Wir schreiben uns nun eine Funktion, die aus einer Liste eine Differenzliste macht.

```
(defun dlist (liste)
  (cond ((null liste) nil)                ; Liste leer?
        (T `(,liste ,(lastlist liste))))) ; Differenzliste bauen
```

Als nächstes müssen wir noch die Funktion `lastlist` definieren, welche das letzte Element der angegebenen Liste liefern soll und nicht eine Kopie davon.

```
(defun lastlist (liste)
    (cond ((null (cdr liste)) liste)   ; Listenende erreicht?
          (T (lastlist (cdr liste))))) ; mit Restliste fortfahren
```

Nun können wir aus der Liste `(a b c)` eine Differenzliste machen.

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

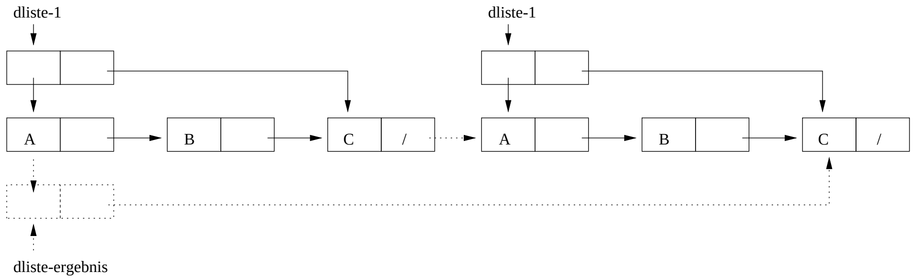
Aber der Aufwand zur Erzeugung der Differenzlisten ware völlig unnütz, wenn man nicht auch den Vorteil, den sie bringen, ausnützen würde. Also schreiben wir eine Prozedur `dappend`, mit der sich Differenzlisten 'schnell' verketten lassen.

```
(defun dappend (dliste-1 dliste-2)
  (cond ((null dliste-1) dliste-2)
        ((null dliste-2) dliste-1)
        (T (setf (cdadr dliste-1) (car dliste-2))
              `(,(car dliste-1) ,(cadr dliste-2)))))
```

Unsere Prozedur `dappend` schluckt nur Differenzlisten als Eingabe, daher testen wir sie folgendermaßen:

```
>  (dappend (dlist '(a b c)) (dlist '(1 2 3)))
((A B C 1 2 3) (3))
```

Die Funktionsweise von `dappend` kann man sich am besten wieder an einem Bild verdeutlichen. Das gestrichelt dargestellte wird durch das `dappend` erzeugt.



An dieser Stelle wollen wir uns noch einmal in Erinnerung rufen, was *Seiteneffekte* sind. Denn insbesondere auch die in diesem Kapitel eingeführten destruktiven Listenprozeduren erzeugen Seiteneffekte. Unter Seiteneffekten versteht man Dinge, die noch weiterhin gelten, auch wenn die Prozedur, die sie verursacht hat, schon längst abgearbeitet ist.

Aus dem Blickpunkt des funktionalen Programmierstils sind Seiteneffekte unsauber. Gründe, warum man trotzdem destruktive Listenprozeduren benutzt sind vor allem Effizienzgewinn (wie bei Differenzlisten) und – seltener – Übersichtlichkeit. In allen anderen Fällen sollte genau überlegt werden, ob nicht auch seiteneffektfreie Programmierung denselben Zweck erfüllen kann.

8.4 Exkurs: Vorteile seiteneffektfreier Programmierung

Durch das Fehlen von Seiteneffekten bezeichnet derselbe Ausdruck *immer* denselben Wert. Diese Eigenschaft wird *referentielle Transparenz (refential transparacy)* genannt. Die folgenden zwei Beispiele verdeutlichen diesen Umstand:

```
(defun foo1 (x)
    (cond ((and (bar x) (baz x)) (+ x 1))
          (T x)))

(defun foo2 (x)
    (cond ((and (baz x) (bar x)) (+ x 1))
```

```
            (T x)))
```

Falls `bar` und `baz` Nebeneffekte enthalten, sind die Funktionen `foo1` und `foo2` nicht identisch. Jenachdem ob `bar` oder `baz` zuerst auf `x` angewendet wird, können sich im zweiten Teil der `and`-Verknüpfung unterschiedlich Ergebnisse ergeben. Hier lauert eine nicht zu unterschätzende Gefahr! Man sieht `bar` und `baz` von außen nicht an, ob sie `x` destruktiv verändern, so daß eine der beiden `foo`-Funktionen für den angestrebten Zweck falsch sein könnte. Daraus resultierende Fehler sind sehr schwierig zu entdecken!

Referentiell transparente Programme haben den Vorteil mit mathematisch sauberen Methoden verifizierbar zu sein[14]. Diese Methoden sind *Substitution* und *Induktion*.

Beispielhaft soll hier die Korrektheit einer Funktion bewiesen werden, die die Fakultät einer Zahl n (n!) berechnet:

```
(defun fak (n)
   (cond ((= n 0) 1)                              || [1]
         ((> n 0) (* (+ n 1) (fak (- n 1))))   || [2]
         (t (error "Fehler: Argument < 0 !"))))
```

Der Beweis erfolgt durch Induktion: Zuerst muß gezeigt werden, daß die Funktion die Fakultät für den Wert $n = 0$ berechnet (Induktionsanfang, Schritt 1), dann, daß unter der Annahme, daß sie die Falkultät von n korrekt berechnet, sie dies auch für $n + 1$ tut (Induktionschritt, Schritt 2).

Schritt 1: `(fak 0)` $= 1$ Durch Regel [1]

[14]Unter dem Begriff *Verifikation* aus der theoretischen Informatik versteht man den mathematischen Beweis, daß ein Programm korrekt ist, d.h. genau das tut, was es tun soll. Außerdem können mit Verifikationsmethoden Schwachstellen (wie Ineffizienz) in Programmen aufgespürt werden. Je 'sauberer' die mathematischen Grundlagen einer Programmiersprache sind, desto einfacher ist es, z.B. automatische Verifikationsverfahren auf Programme anzusetzen. (Siehe dazu auch das Vorlesungsskript *Grundlagen der theoretischen Informatik* von Prof. Sperschneider.)

	$1 = 0!$	Definition der Fakultät
Schritt 2:	Annahme: `(fak n)` $= n!$ für $n > 0$	Induktionsannahme
	`(fak (+ n 1)) = (* (+ n 1) (fak n))`	
		Durch Regel [2], $n + 1 > 0$
	`(fak (+ n 1))` $= (n + 1) * n!$	Durch Induktionsannahme
	`(fak (+ n 1))` $= (n + 1)!$	Definition der Fakultät

Der Beweis setzt, wie in der Mathematik üblich voraus, daß Funktionen nur auf Werte angewendet werden, für die sie definiert sind. In unserem Programm wird der Fall falscher Argumente durch eine Fehlermeldung abgefangen.

Der obige Beweis ist nur dann korrekt, wenn keine der beteiligten Funktionen Seiteneffekte hat in der Art, daß `n` verändert wird. Würde das geschehen (insbesondere durch `(> x 0)`), dann wäre ein derart einfacher und eleganter Beweis nicht möglich.

Generell besteht zwischen rekursiver Definition und induktivem Beweis einer Funktion ein starker Zusammenhang: Es wird unterschieden zwischen speziellen Fällen (Induktionsanfang) und dem generellen Fall (Induktionsschritt).

8.5 Übungen

1. Formuliere äquivalente Ausdrücke für `rplaca` und `rplacd` unter Verwendung von `setf`.

2. Schreibe eine Funktion, die aus einer Differenzliste eine 'normale' Liste macht.

3. Schreibe eine Prozedur `addlast`, welche an eine Differenzliste ein Element anhängt.

9 Weitere Datenstrukturen

9.1 Assoziationslisten

Eine Assoziationsliste ist eine Liste, die bestimmte Konventionen erfüllt. Sie besteht aus eingebetteten Unterlisten, deren erstes Element jeweils einen (Zugriffs-)Schlüssel darstellt. Die allgemeine Struktur einer Assoziationsliste lautet:

```
((schluessel-1 wert-1)
 (schluessel-2 wert-2)
 ...
 (schluessel-n wert-n))
```

Assoziationslisten werden in vielerlei Hinsicht benutzt. Eine häufige Art der Anwendung ist das Speichern von Daten. Möchte man z.B. zu einem Buch verschiedene Daten wie Titel, Autor, Verlag usw. halten, so kann man dieses recht leicht mittels einer Assoziationliste machen.

```
>  (setq buch '((titel lisp)
                (autor Winston/Horn)
                (verlag addison-wesley)
                (erscheinungsjahr 1987)
                (auflage 2)))
((TITEL LISP)
 (AUTOR WINSTON/HORN)
 (VERLAG ADDISON-WESLEY)
 (ERSCHEINUNGSJAHR 1987)
 (AUFLAGE 2))
```

Auf diese Weise haben wir dem Symbol `buch` eine Assoziationsliste zugewiesen, die die Schlüssel `titel`, `autor`. `verlag`, `erscheinungsjahr` und `auflage` besitzt. Um nun eine bestimmte Eigenschaft abfragen zu können, gibt es die Grundprozedur `assoc`. Diese verlangt als erstes Argument den Schlüssel (Eigenschaftsnamen) und als zweites die Assoziationsliste bzw. ein Symbol, dessen Wert eine Assoziationsliste ist. Möchte wir nun erfragen, wer der Autor unseres Buches ist, so können wir dieses mit einen Aufruf von `assoc` erreichen.

```
>  (assoc 'autor buch)
(AUTOR WINSTON/HORN)
```

Analog lassen sich auch alle anderen Schlüssel abfragen.

```
>  (assoc 'verlag buch)
(VERLAG ADDISON-WESLEY)
>  (assoc 'auflage buch)
(AUFLAGE 2)
```

Es ist darauf zu achten, daß `assoc` immer Schlüssel und Wert zusammen zurückgibt, bzw. `nil`, wenn ein Schlüssel nicht gefunden wird.

```
>  (assoc 'preis buch)
nil
```

Der Grund für die Rückgabe der gesamten Unterliste im Erfolgsfall, Schlüssel und Wert, ist die Vermeidung von Zweideutigkeit. Würde nur der Wert, also der `cdr` der Unterliste, zurückgegeben, könnte man zwei Fälle nicht unterscheiden. Nämlich den Fall, daß der Schlüssel den Wert `nil` hat, und den Fall, daß der Schlüssel nicht gefunden wurde.

Sicherlich wird man sich nicht die Mühe machen, eine Assoziationsliste anzulegen, wenn man nur ein Buch verwalten möchte, sondern man möchte eine Literaturliste anlegen, in der man beispielsweise seinen gesamten Bücherbestand ablegt. Dazu erzeugt man zu jedem Buch eine Assoziationsliste und hält die Symbolnamen in einer 'literaturliste'. Um die Arbeit im Umgang mit der Literaturliste zu vereinfachen, definiert man sich Funktionen, die das Einfügen, Löschen, Suchen usw. erledigen.

```
(setq literaturliste nil)     ; leere Liste erzeugen

(defun buch-einfuegen (buch)
   (setq literaturliste (cons buch literaturliste)))
   ; Buch in Literaturliste einfuegen

(defun suchen (schluessel buch)
   (let (hilf erg)           ; binden und mit NIL init.
     (dolist (hilf literaturliste erg)
       (cond ((equal hilf buch)
                (return (assoc schluessel (eval buch))))))))
      ; Wert eines bestimmten Schluessels des angegebenen
      ; Buches ausgeben
```

Durch diese Funktionen hat man nun einen leichteren Zugriff auf seine Daten. Die Liste der hier nützlichen Funktionen läßt sich natürlich noch beliebig erweitern. Zu erwähnen ist noch, daß `assoc` immer den ersten passenden Schlüssel nimmt und die entsprechende Unterliste ausgibt. Einen Schlüssel in einer Assoziationsliste mehrfach vorkommen zu lassen, kann dennoch recht sinnvoll sein. Hat man beispielsweise die Adressen seiner Bekannten mittels Assoziationslisten gespeichert und es fährt jemand für ein paar Wochen weg, so fügt man die neue Adresse einfach am Anfang der entsprechenden Assoziationsliste an. Läßt man sich nun die Adresse ausgeben, erhalt man die neue. Ist der Bekannten aus dem Urlaub wieder zurück, so löscht man das erste Vorkommen von Adresse in der Assoziationsliste wieder. Nun erhält man bei Abfrage der Adresse wieder die alte, ohne sie jemals gelöscht oder geändert haben zu müssen.

Die oben verwendete Grundprozedur `eval` sorgt dafür, daß ihr Argument ausgewertet wird.

```
>  (setq buch 'buch1)
>  (setq buch1 '((titel lisp)))
```

```
>  buch
BUCH1
>  (eval buch)
((TITEL LISP))
```

Laßt man in der Funktion `suchen` das `eval` weg, so gibt das System die Fehlermeldung `error: bad argument type - BUCH1` aus. Das liegt daran, daß `assoc` die Variable `buch` einmal auswertet, also ihren Wert `buch1` erhält, was aber keine Assoziationsliste ist. Um an die Assoziationsliste zu kommen, müßte `buch1` nochmal ausgewertet werden. Um die zweifache Auswertung zu erzwingen, muß also an dieser Stelle das `eval` stehen. `eval` ist eine sehr mächtige Prozedur, mit ihrer Hilfe ist es z.B. trivial, einen Lispinterpreter in Lisp zu schreiben (nicht umsonst heißt es read-*eval*-loop). Zur Laufzeit können Ausdrücke zusammengesetzt werden (Daten) und dann mittels `eval` ausgewertet bzw. ausgeführt werden (Programm). Eine strikte Trennung wie in Compilersprachen wie Pascal oder C gibt es in Lisp somit nicht!

9.2 Übungen:

1. Schreibe eine Funktion, welche zu der folgenden Assoziationsliste ausgibt, welche Person im angegebenen Verwandtschaftsverhältnis zu `peter` steht:

   ```
   (setq peter '((vater . michael) (mutter . maria)
                 (bruder . markus) (schwester . maria)))

   >  (wer 'mutter)
   (MUTTER . MARIA)
   ```

2. Betrachte die folgende Tabelle:

Mutter	Vater	Kind
Renate	Ingo	Maria
Renate	Ingo	Manfred
Susanne	Manfred	Klaus
Heide	Hans	Susanne
–	–	Ingo
–	–	Renate

berlege, wie sich obige Tabelle mittels Assoziationslisten darstellen läßt. Schreibe Funktionen, die zu einer angegebenen Person die Eltern, die Großmutter und den Großvater angeben. Zum Testen kannst Du die genannten Verwandtschaftsbeziehungen benutzen. Du solltest darauf achten, daß Du auch 'vernünftige' Ergebnisse erhälst, wenn die Angaben nicht vollständig sind.

```
>  (parents_of maria)
((VATER INGO) (MUTTER RENATE))

>  (grandpa klaus)
(INGO HANS)
```

3. Meyers planen ein Picknick im Grünen. Dazu benutzen sie die folgende Wetterprognose für die nachste Woche:

```
Montag wolkenlos 30 Grad.
Dienstag regnerisch 18 Grad.
Mittwoch bedeckt 26 Grad.
Donnerstag sonnig 30 Grad.
Freitag unbestaendig 29 Grad.
Samstag wolkenlos 30 Grad.
Sonntag sonnig 27 Grad.
```

Sie möchten für das Picknick einen Tag finden, an dem alle frei haben und das Wetter schön ist. Das Wetter ist schön, wenn es wolkenlos ist oder die Sonne scheint. Da Hans, der Vater, ein Astronom ist, arbeitet er, wenn der Himmel wolkenlos ist. An anderen Tagen hat er frei. Fritz, der Sohn, geht von Montag bis Samstag in die Schule, hat aber schulfrei, wenn die Temperaturen auf über 28 Grad steigt. Eva, die Mutter, hat Sonntagsdienst im Krankenhaus, sonst aber frei.

berlege Dir eine geeignete Lisp-Darstellung (Assoziationslisten) für die obige Wetterprognose.

Schreibe Lisp-Funktionen, mit denen Du in einem Aufruf feststellen kannst, an welchen Tagen Meyers ihr Picknick machen können. Unverbindlicher Tip: Entwickle zuerst Funktionen, die für jede einzelne Person feststellen, ob sie an einem angegebenen Tag frei hat. Benutze diese Funktionen wiederum für eine neue Funktion, die ermittelt, ob an einem angegebenen Tag alle frei haben und das Wetter schön ist. Auf diese Funktion kannst Du dann die Funktion `mapcar` anwenden, sofern Du die Tage in geeigneter Weise dargestellt hast.

9.3 Eigenschaftslisten (property lists)

Ein Symbol ist ein Konstrukt, daß aus verschiedenen Teilen bestehen kann. Es kann zum einen einen Wert haben `(setq symbol 'x)` oder an ihm kann eine Funktionsbindung hängen `(defun symbol ...)`. Eigenschaftslisten sind ein weiterer Teil eines Symbols. Denn jedes Symbol hat eine Eigenschaftsliste, nur wird sie in den meisten Fällen nicht angewendet, d.h. wird ein Symbol erzeugt, so ist die Eigenschaftsliste leer (`nil`) und das wird bei den meisten Symbolen auch nicht verändert. Die Anwendung ist jedoch jederzeit möglich. Eine Eigenschaftsliste ist von der Form:

```
X-(eigenschaft-1 wert-1 eigenschaft-2 wert-2  ....
   eigenschaft-n wert-n)
```

Das `X` steht für irgendeinen Symbolnamen, der Bindestrich soll verdeutlichen, daß die Eigenschaftsliste an sein Symbol gehängt ist.

Um Eigenschaften abzufragen, verwendet man `get`. Der Ausdruck

```
(get symbol eigenschaft)
```

liefert den Wert der erfragten Eigenschaft des angegebenen Symbols. Im Gegensatz zu `assoc` liefert `get` nur den Wert und nicht Eigenschaft und Wert. Deshalb sollte man Eigenschaften nicht unbedingt den Wert `nil` geben. Denn dann läßt sich nicht unterscheiden, ob eine Eigenschaft den Wert `nil` hat oder ob sie nicht gefunden wurde.

Um eine Eigenschaft zu setzen, verwendet man `setf`. Als Zugriffsform benutzt man hier `get`. Mit

```
(setf (get symbol eigenschaft) wert)
```

wird der Wert einer Eigenschaft gesetzt.

Nun wollen wir die Daten eines Buches mittels einer Eigenschaftsliste speichern.

```
> (setf (get 'buch1 'titel) 'lisp)
LISP
> (setf (get 'buch1 'autor) 'winston/horn)
WINSTON/HORN
```

`symbol-plist` gibt die Eigenschaftsliste zu einem Symbol aus.

```
> (symbol-plist 'buch1)
(AUTOR WINSTON/HORN TITEL LISP)
```

`putprop` leistet dasselbe wie die Kombination aus `setf` und `get`. Allerdings ist hierbei die Reihenfolge der Argumente ein wenig verdreht, d.h. `(putprop symbol wert eigenschaft)` leistet das Gewünschte.

```
> (putprop 'buch1 'addison-wesley 'verlag)
ADDISON-WESLEY
> (get 'buch1 'verlag)
ADDISON-WESLEY
```

Möchte man eine Eigenschaft entfernen, so geht dieses mit `remprop` ('remove property').

```
> (remprop 'buch1 'autor)
nil
> (symbol-plist 'buch1)
(VERLAG ADDISON-WESLEY TITEL LISP)
```

Eigenschaftslisten können jede Eigenschaft nur einmal besitzen, da das Verändern der Werte der Eigenschaften destruktiv geschieht. Daher eignen sich Eigenschaftslisten auch nicht für die Verwendung als Stack, im Gegensatz zu Assoziationslisten. Das ist auch ein Grund, warum Variablenbindungen in Common Lisp in Assoziationslisten gespeichert werden. Auf diesen Punkt kommen wir gleich wieder zurück.

9.4 Übungen:

1. Stelle Dir vor, Du möchtest die Telefonnummern Deiner Bekannten im Rechner verwalten. Dann könntest Du dazu eine Eigenschaftsliste verwenden, wie z.B. die folgende:

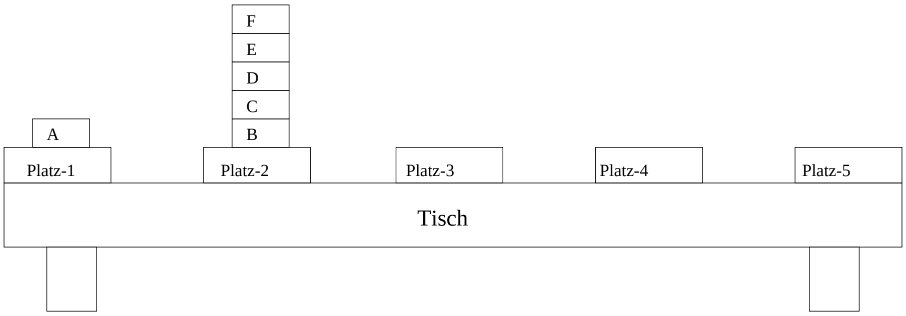
   ```
   (PAUL 34556 PETER 35353 MARIA 95644 MARK 36747)
   ```

   Erzeuge dazu die nötige Eigenschaftsliste mit dem Namen `telefonliste'`. Verwende dazu die Funktionen `setf` und `get` bzw. die Funktion `putprop`. Ob Du die Eigenschaftsliste richtig angelegt hast, kannst Du mit

   ```
   (symbol-plist 'telefonliste)
   ```

   überprüfen. Wie erhält man zu einem Namen die zugehörige Telefonnummer?

2. Betrachte die folgende Klötzchenwelt:



Zur Realisierung der Klötzchenwelt schreibe folgende Funktionen:

Mit der Funktion `def-object` sollen die Objekte in der Klötzchenwelt eingefügt werden. berlege Dir dazu, welche Informationen Du zu jedem Klotz wissen mußt, d.h. welche Attribute ein Objekt besitzen muß. Die Klötzchenwelt sollte durch eine entsprechende Anzahl von Aufrufen der Funktion `def-object` erstellt werden. Gleichzeitig sollte in irgendeiner Form gespeichert werden, welche Objekte bereits existieren.

Damit die Attribute eines Objektes gesetzt werden können, solltest Du eine Funktion `set-att` definieren, mit der Du jedem Attribut jedes Objektes einen Wert zuweisen kannst.

Ebenso benötigst Du eine Funktion `get-att`, mit der Du die Werte der Attribute der einzelnen Objekte erhälst.

Um einem Klotz eine Position zu geben, ist es notwendig, vorher zu testen, ob der Platz frei ist. Definiere dazu die Funktion `is-clear`. Schreibe ebenfalls eine Funktion `on-top`, welche zu einem angegebenen Objekt ausgibt, welches Objekt sich auf ihm befindet, bzw. `nil`, falls sich auf ihm kein Objekt befindet.

Um die Klötzchenwelt von außen 'betrachten' zu können, sollte es möglich sein, sich anzeigen zu lassen, wo sich welche Klötze befinden. Schreibe dazu eine Funktion `wo`, die Dir dieses ausgibt.

3. Betrachte die Klötzchenwelt von Aufgabe 2. Es soll nun das bereits bestehende Programm soweit erweitert werden, daß Klötze bewegt werden können. Der Dialog soll dabei erst einmal über einen Funktionsaufruf

aus dem Eingabefenster heraus stattfinden. Dazu sind folgende drei Funktionen zu schreiben:

Die Funktion `find-space` soll einfach einen Platz liefern, der frei ist. Dazu kann man alle in Frage kommenden Plätze (d.h einen Klotz bzw. einen Platz auf dem Tisch) betrachten und den nächst möglichen zurückgeben. Jedoch ist darauf zu achten, daß nicht jeder freie Platz in einer bestimmten Situation benutzt werden kann. Betrachte das folgende Beispiel:

```
(move-to 'klotz-a 'platz-2)
```

Wenn man `platz-2` freimachen will, um `klotz-a` darauf legen zu können, so ist es sicherlich nicht sinnvoll, einen Klotz, der auf `platz-2` liegt in dieser Situation auf `klotz-a` zu legen. Dieses gilt natürlich auch im umgekehrten Fall.

Die Funktion `make-clear` soll dafür sorgen, daß ein angegebener Platz (bzw. Klotz) freigeräumt wird, sofern er es nicht schon ist. Auch hierbei muß darauf geachtet werden, daß dabei nicht jeder Platz benutzt werden darf. Tip: Es kann durchaus sinnvoll sein, daß `make-clear` von `move-to` Gebrauch macht.

Die Funktion `move-to` soll das Bewegen eines Klotzes mit allen sich daraus ergebenden Konsequenzen steuern. Sie sollte mindestens zwei Parameter besitzen, so daß sie mit dem zu bewegenden Klotz und dem Ziel des Klotzes aufgerufen werden kann. Mindestens deshalb, weil eventuell noch ein zusätzlicher (optionaler) Parameter notwendig wird, über dem man der Funktion mitteilen kann, welche Plätze bei der jeweiligen Klotzbewegung nicht benutzt werden dürfen. Die Funktion `move-to` kann folgendermaßen funktionieren:

```
>  (move-to 'klotz-a 'platz-2)
```

Als erstes müssen nun zwei Aufrufe der Funktion `make-clear` stattfinden, die dafür sorgen, daß sowohl das Ziel (`platz-2`) als auch der zu setzende Klotz (klotz-a) freigeräumt werden. Danach muß noch ausgegeben werden, welcher Klotz wohin bewegt werden soll. Die entsprechenden Attribute der Objekte, für die sich etwas geändert hat, müssen nun die aktuellen Werte erhalten, was mittels geeigneter Aufrufe von set-att (s. 2.) geschehen kann.

9.5 Arrays

Lisp stellt neben den traditionell verwendeten – und auch meist bevorzugten – Listen andere Datentypen, wie z.B. Arrays zur Verfügung. Arrays werden wie Listen fester Länge benutzt, wobei nicht – wie mit car und cdr – auf das jeweils erste Element oder den Rest zugegriffen wird, sondern mittels eines Indexes auf einzelne Elemente direkt. Hier liegt auch der Vorteil von Arrays Listen gegenüber: Wenn häufig auf exakte Positionen zugegriffen werden soll, läßt sich das mit weniger Operationen bewerkstelligen als bei Listen. Listen haben den Vorteil höherer Flexibilität, insbesonderer den beliebiger, dynamisch änderbarer Länge.

Die Benutzung und Verwaltung von Arrays geschieht über nur drei Befehle zum kreieren ganzer Arrays, abfragen und setzen einzelner Elemente. Ein neues Array der Länge x wird mit make-array x angelegt. Mehrdimensionale Arrays sind nicht vorgesehen. Die Funktion (aref <array> <pos>) liefert den Wert des Arrays <array> an Position <pos>, wobei die Zählung bei 0 beginnt. Wird auf ein nicht innerhalb der Arraygrenzen liegendes Element zugegriffen, wird eine Fehlermeldung ausgegeben. Zum Setzen einzelner Elemente wird das oben erwähnte Verfahren mit setf angewendet: (setf (aref <array> <pos>) <new-value>) setzt Position <pos> im Array <array> neu mit dem Wert <new-value>.

Beispiele:

```
> (setq feld (make-array 4))
#(nil nil nil nil)
> (setf (aref feld 0) 1)
```

```
1
> (setf (aref feld 1) 2)
2
> (aref feld 0)
1
> feld
#(1 2 nil nil)
```

10 Funktionsdefinitionen

10.1 Variablenbindungen

Zunachst möchten wir klären, auf welche Art und Weise in LISP Variablenbindungen verwaltet werden. Das ist wichtig zu wissen, wenn Funktionsdefinitionen geschachtelt vorkommen. Dazu betrachten wir ein einfaches Beispiel:

```
>  (setq y 3)
3
>  (defun funktion-1 ()
      (let ((y 5)) (defun funktion-2 () y)))
FUNKTION-1
```

Durch den Aufruf von `funktion-1` wird nun auch `funktion-2` definiert. Die Frage, die sich nun stellt, ist die: Welcher Wert wird ausgegeben, wenn `funktion-2` aufgerufen wird, 5 oder 3?

```
>  (funktion-2)
5
```

Die Ausgabe hier bezieht sich auf die meisten modernen LISP-Systeme, die mit 'lexikalischer' Bindung (Scoperegeln) arbeiten, wie es in COMMON LISP definiert ist. Aber es gibt durchaus auch Systeme, die hier den Wert 5 ergeben hätten. Solche Systeme arbeiten mit dynamischen Bindungen. Der Unterschied zwischen den beiden Bindungsarten besteht darin, daß die lexikalische Bindung dafür sorgt, daß der Wert abhängig davon ist, wo die Funktion definiert worden ist, im Gegensatz zur dynamischen Bindung, wo der Wert davon abhängt, in welcher Umgebung die Funktion aufgerufen wird. Die Variablenbindungen in LISP werden in Assoziationslisten gespeichert. Für das Beispiel oben würde das folgende Liste bedeuten:

```
                  ((y . 5) (y . 3))
Definition von Funktion-2 | durch das setq
```

Wie wir schon gesehen haben, liefert ein `assoc` immmer den Schlüssel und den Wert des ersten Vorkommens des gesuchten Schlüssels. Daher wird in diesem Fall beim Aufruf von `funktion-2` auch `5` und nicht `3` ausgegeben. Läßt man die `(y 5)` in obigem Beispiel weg, so liefert `funktion-2` den Returnwert `3`. Denn dann besteht die Variablenliste nur aus `((y . 3))` und somit wird dieser Wert für `y` ausgegeben.

10.2 Funktionsdefinition als Variablenbindung

Was macht das `defun` (define function) ?

Funktionsdefinitionen in Lisp sind nichts anderes als Lisp-Datenstrukturen, was Lisp deutlich von anderen Programmiersprachen unterscheidet.

```
(defun name (parameterliste) <funktionsrumpf>)
                     =
(setq name #'(lambda (parameterliste) <funktionsrumpf>))
```

Dabei passiert folgendes: Einem Symbol `name` wird als Wert eine Funktionsdefinition zugewiesen. (`#'`<`ausdr`> ≡ `(function` <`ausdr`>`)`) Der Name `lambda` leitet sich von dem von Church entwickelten Lambdakalkül in der Logik ab.

Beispiel: $\lambda x : x + 5 = f$ (Funktionsdefinition von f)
$f(3) = (\lambda : x + 5)(3)$ (ersetze alle freien Vorkommen von 'x' durch 3 in dem Ausdruck (x + 5))

Lisp unterscheidet sich vom Lambdakalkül dadurch, daß in einer Funktion mehrere Ausdrücke nacheinander stehen können, aber nur der letzte Ausdruck liefert den Wert der Funktion, wohingegen die anderen Ausdrücke 'nur' Nebeneffekte liefern. Die Nebeneffekte sind logisch gesehen 'Schmutz'. Aber in vielen Fällen interessieren nur die Nebeneffekte und nicht der eigentliche Rückgabewert einer Funktion. (Z.B. `defun`, dabei interessiert nicht der Name der Funktion, der ausgegeben wird, sondern, daß an den Funktionsnamen die Funktionsdefinition gebunden wird.)

```
(defun f-1 ()
    (let ((x 1)(y 2))
      (defun f-2 ()
          (let ((x 3))
            (print (* x y))))))
>  f-1
#f((LAMBDA nil  (LET  ((X 1) (Y 2))
   (DEFUN F-2 nil (LET ((X 3)) (PRINT (* X Y)))))))
```

Gibt man im Eingabefenster nur den Funktionsnamen an, so gibt das System die Funktionsdefinition aus. Das `#f` bedeutet, daß es sich um einen Funktionsabschluß handelt. Die beiden `nil` zeigen an, daß sowohl bei `f-1` als auch bei `f-2` keine Parameter definiert worden sind. Ansonsten wird hier die Funktionsdefinition wiedergegeben.

```
>  (f-1)
 F-2
>   f-2
#f((LAMBDA nil (LET ((X 3))
   (PRINT (* X Y)))) (Y . 2) (X . 1))
```

Die Ausgabe ist analog zu der obigen zu verstehen. Was hier neu hinzukommt ist die Variablenbindung `(Y . 2) (X . 1)`. Das sind die Werte, die `x` und `y` durch das `let` in `F-1` erhalten haben. Alle Variablen, die hier nicht wie `x` bzw. `y` aufgeführt sind, werden global betrachtet.

Man kann nicht nur mit der Prozedur `defun`, sondern auch mit der Prozedur `lambda` Funktionen definieren. Dieses funktioniert, wie man aus den Vorbemerkungen schließen kann, ganz analog zu der Definition mit `defun`, nur mit dem Unterschied, daß so definierte Funktionen keinen Namen erhalten. Aber das kann durchaus sinnvoll sein. Wenn man eine Funktion nur an einer Stelle benutzt, so braucht man sich keinen extra Namen zu überlegen.

```
>  (mapcar #'(lambda (x) (* x x)) '(1 2 3 4 5 6 7 8 9))
(1 4 9 16 25 36 49 64 81)
```

Durch das `lambda` wird eine Funktion definiert, die zu einer gegeben Zahl das Quadrat berechnet. Es könnte hier statt `#'` auch einfach nur `'` stehen. Das Ergebnis würde sich in diesem Fall nicht andern. Die Form `#'<lambda ausdruck>` unterscheidet sich von `'<lambda ausdruck>` dadurch, daß sie mit einem lexikalischen Abschluß (lexical closure) verbunden ist. Wird ein mit `#'` gekennzeichneter Ausdruck ausgewertet, so gelten lexikalische Bindungsregeln (Scoperegeln). Sollen dagegen dynamische Bindungsregeln gelten, so muß man Quote(`'`) verwenden.

```
>  (setq x 4)
     (defun test ()
         (let ((x 1))
           (setq name '(lambda () (* x x)))))
>  (test)
(LAMBDA nil (* X X))
```

Wie man an dem fehlenden `#f` schon sieht, ist hier kein Funktionsabschluß nach lexikalischen Bindungsregeln gebildet worden.

```
> (name)
16
```

Dies zeigt, daß nicht die Definitionsumgebung entscheidend für das Ergebnis ist, sondern die Aufrufumgebung. Es werden hier dynamische Bindungsregel angewendet. Probiert man die gleiche Funktionsdefinition aus, nur unter Verwendung von `#'` vor dem Lambdaausdruck, so liefert `(name)` den Wert `1`.

```
>   ...
    (setq name #'(lambda () (* x x)) )
    ...

> (test)
#f((LAMBDA nil (* X X)) (X . 1))
```

Denn so wird der lexikalische Abschluß gebildet und dabei `1` an das Symbol `x` gebunden.

10.3 Makros

Nicht alle Aufgaben lassen sich mit Funktionen bzw. Prozeduren der herkömmlichen Art elegant lösen. Ein weiteres von Lisp zur Verfügung gestelltes Mittel sind Makros. Diese werden analog zu Funktionen nur mit der Prozedur `defmacro` anstelle von `defun` definiert. Makros werten ihre Argumente nicht aus, sondern erzeugen mit ihnen erst einen Lisp-Code, der dann ausgewertet wird. Ein kleines Beispiel macht den Unterschied zwischen `defun` und `defmacro` klar.

```
> (setq test '(a b c d))
(A B C D)

> (defmacro m-print (x)   > (defun f-print (x)
```

```
       (print x))                  (print x))
M-PRINT                     F-PRINT

>  (m-print test)           >  (f-print test)
TEST                        (A B C D)
(A B C D)                   (A B C D)
```

An den Testaufrufen zeigt sich, daß das Makro `m-print` sein Argument nicht auswertet. Daher gibt `print` auch `test` aus und nicht den Wert von `test`. Der Lisp-Code, den der Makro erzeugt, besteht aus `(print test)`. Erst dieser wird dann ausgewertet und ergibt den Returnwert `(A B C D)`.

Betrachten wir nun eine typische Anwendung von Makros. Sie werden häufig im Zusammenspiel mit ‘ und `,` bzw. `,@` verwendet. Wir möchten nun ein Makro schreiben, das die Grundprozedur `if` simuliert. Die Funktionsweise von `if`

```
(if <test> <aktion-1, wenn test pos.> <aktion-2 sonst>)
```

möchten wir dabei durch eine entsprechende Form von `cond` simulieren.

```
(cond (<test> <aktion-1, wenn test pos.>)
      (T     <aktion-2 sonst>))
```

Wir möchten die oben dargestellte Schablone mit den entsprechenden Daten (Test und Aktionen) füllen und danach auswerten.

```
>  (defmacro mein-if (test aktion-1 &optional aktion-2)
       `(cond (,test ,aktion-1)
        (T ,aktion-2)))
MEIN-IF

>  (mein-if (< 3 4) (print 3) (print 4))
3
3
```

Der optionale Parameter bei der Makrodefinition steht für den optionalen Else-Teil des `if`. Die Wirkungsweise unseres Makros beim Aufruf besteht darin, daß er die `cond`-Schablone mit Werten füllt, d.h. beim Aufruf von `mein-if` wird folgender Lisp-Code (Zwischenform) gebildet:

```
(cond ((< 3 4) (print 3))
      (T       (print 4)))
```

Dieser LISP-Code wird erst dann (also nach dem Ausfüllen der Schablone) ausgewertet. Makros nutzen die Tatsache aus, daß Lisp-Pozeduren und Lisp-Daten die gleiche Form haben.

10.4 Übungen:

1. Schreibe die Funktionen `when` und `unless`, die folgendes leisten, als Makros:

   (when <test> <ausdrücke>) ≡ (cond (<test> <ausdrücke>))

   (unless <test> <ausdrücke>) ≡ (cond ((not <test>) <ausdrücke>))

   ```
   >  (when (< 4 13) (print 5))
   5
   >  (unless (> 4 13) (print 15))
   15
   ```

11 Ein- und Ausgabe

Ein wesentlicher Aspekt beim Programmieren, von dem wir auch schon stillschweigend Gebrauch gemacht haben, ist die Ein- und Ausgabe. Insbesondere ist die Schnittstelle zum Benutzer nicht zu unterschätzen.

Die Prozeduren `print`, `princ` und `prin1` haben wir bereits benutzt. Um einfach einen Zeilenvorschub zu bewirken, gibt es die Funktion `terpri`.

```
>  (defun ausgabe ()
(princ '(1 2 3)) (terpri) (princ '(4 5 6)))

>  (ausgabe)
(1 2 3)
(4 5 6)(4 5 6)
```

Wäre `terpri` oben nicht vorhanden, so würden die Listen hintereinander ausgegeben.

```
(1 2 3)(4 5 6)(4 5 6)
```

Um nur ein Zeichen zu schreiben bzw. zu lesen, gibt es die Funktionen `write-char` und `read-char`.

```
>  (write-char #\A)
A65
```

Das `A` ist die Ausgabe von `write-char` und die 65 der Returnwert. Analog liest `read-char` den Charakter an der aktuellen Cursorposition und gibt seinen Integerwert (ASCII) zurück.

Möchte man bei der Ausgabe bestimmte Formate einhalten z.B. einen Rand, so ist es wichtig, vor dem Aufruf einer Print-Funktion zu wissen, wie lang der dort auszugebende Ausdruck ist, um gegebenenfalls vorher einen Zeilenvorschub zu machen. Da es zwei verschiedene Printfunktionen gibt, die selbst keinen Zeilenvorschub veranlassen, aber ihre Argumente unterschiedlich ausgeben, gibt es auch zwei Funktionen, die die Länge eines Ausdruckes unterschiedlich bestimmen.

```
>  (setq x (a b "c" " " d))
(A B "c" " " D)
>  (prin1 x)
(A B "c" " " D)
```

```
> (princ x)
(A B c   D)
> (flatsize x)
15
> (flatc x)
11
```

`flatsize` gibt also die Länge eines Ausdrucks an, wenn er mit `prin1` ausgegeben wird und `flatc`, wenn mit `princ` ausgegeben wird.

11.1 Dateien

Eine andere Art der Ein- und Ausgabe ist die Benutzung von Dateien. Auch hierbei gibt es diverse Varianten in den unterschiedlichen Lisp-Systemen. Momentan wird das Verhalten von Dateien auf dem Virtuellen Campus noch implementiert, daher folgt hier eine Darstellung des Zugriffs auf Dateien, wie es im ANSI Common LISP definiert ist. Die Anwendung auf dem Virtuellen Campus wird nur minimal davon abweichen.

Möchte man eine Datei öffnen, so gibt es dafür zwei Gründe. Entweder möchte man daraus lesen oder darein schreiben. In beiden Fällen öffnet man zunächst einen Stream. Standardmäßig wird die Eingabe von dem Stream `*standard-input*` gelesen, die Ausgabe von `*standard-output*`. Voreingestellt entspricht beides dem Toplevel. Dies haben wir implizit genutzt, als wir die Funktionen `read` und `format` verwendet haben. Erstere Funktion hat als optionales Argument einen Stream, dessen Default `*standard-input*` ist, letzere Funktion kann ebenfalls einen Stream erhalten, wenn dieses Argument aber `t` ist, wird `*standard-output*` benutzt. Dateien können wir also nun verwenden, indem wir die Streams explizit übergeben.

Eine Funktion, um Dateien zu öffnen ist `open`. Sie erhält einen Pfadnamen und eine Anzahl optionaler Schlüsselwort-Argumente. Ist die Funktion erfolgreich, liefert sie einen Stream zurück, der auf die Datei zeigt.

Statt des Pfadnamens kann man auch einen String verwenden, dies ist aber nicht mehr portabel. Daher muß man mit einen Pfadnamen erzeugen, indem man mit Schlüsselwort-Argumenten die Komponenten `host`, `device`, `directory`, `name`, `type` und `version` angibt. Im einfachsten Fall reicht die Komponente `name`.

```
> (setf path (make-pathname :name "mmyfile"))
> (setf str (open path :direction :output :if-exists :supersede))
```

Die ausgegebene Repräsentation des Streams ist von der Implementation abhängig. In jedem Fall aber ist es wichtig, den Stream in einer Variablen zu sichern.

Mit `open` spezifiziert man außerdem, wie man den Stream nutzen möchte. Das Argument :direction zeigt an, ob man schreiben (`:output`), lesen (`:input`) oder beides (`:io`) möchte. Wenn der Stream für die Ausgabe benutzt wird, kann man mit `if-exists` bestimmen, was passieren soll, wenn die Datei schon existiert. Normalerweise ist dies `supersede`, das besagt, daß die Datei überschrieben werden soll.

Wenn wir diesen Stream nun als erstes Argument an `format` üebergeben, wird die Ausgabe nicht mehr auf den Toplevel, sondern auf den Stream geschrieben:

```
> (format str "Beliebiger String")
NIL
```

Wenn wir uns nun die Datei anschauen, ist unsere Ausgabe nicht notwendigerweise schon darin enthalten, da viele Systeme ihre Ausgabe puffern, und erst eine gewissen Menge an Zeichen zwischenspeichern, bevor sie sie in die Dateien schreiben. Wirklich sicher sein können wir uns erst, wenn wir den Stream geschlossen haben:

```
> (close str)
NIL
```

Man sollte einen Stream generell schließen, wenn man ihn nicht mehr braucht, da viele Systeme nur begrenzt viele Streams unterhalten können.

Analog zum Schreiben öffnen wir Dateien zum Lesen mit `:direction :input`. Man kann jede Eingabefunktion auf eine Datei anwenden. Hier ist ein Beispiel, in dem `read-line` benutzt wird, um eine Zeile Text von der Datei einzulesen:

```
> (read-line str)
"Beliebiger String"
NIL
```

11.2 Übungen:

1. An dieser Stelle erinnere Dich noch einmal an die Aufgabe 3. aus Kapitel 2. Denn nun hast Du das nötige Wissen, um den Dialog richtig komfortabel zu gestalten. Dazu soll ein extra Dialogfenster eingerichtet werden. Dazu solltest Du das obige Programm so abändern, daß beim Start des Programms ein entsprechendes Fenster geöffnet wird, in dem die gegenwärtige Situation (wie oben) ausgegeben wird, und in dem der Benutzer die Anweisungen an das Programm schreibt. Dem Benutzer soll es möglich sein, seine Eingaben in umgangssprachlicher Weise einzugeben.

   ```
   >  (Lege den Klotz-a auf den Klotz-b)
   ```

12 Objektorientierung mit CLOS

Im vorliegenden Text werden kurz die wesentlichsten Eigenschaften von CLOS anhand einiger Beispiele demonstriert.

12.1 Definition von Klassen und Objekterzeugung

Die Definition von Klassen entspricht weitgehend der Definition von Strukturen. Möchte man zum Beispiel einen Punkt im dreidimensionalen Raum als Struktur definieren, würde dies etwa folgendermaßen aussehen:

```
(defstruct 3d-point-struct x y z)
```

Eine entsprechende Klassendefinition sähe dann folgendermaßen aus:

```
(defclass 3d-point-class () (x y z))
```

Die zweite Liste definiert die Slots [15] der Klasse, die erste (hier leere) Liste wird zunächst nicht berücksichtigt. Was ist nun mit dieser Klassendefinition gewonnen? Nichts! Im Gegenteil: bei der Definition einer Struktur werden automatisch verschiedene Funktionen zum Erzeugen der Struktur und den Zugriff auf Slots generiert. Dies passiert bei dem oben aufgeführten Aufruf von `defclass` nicht. Um eine Instanz, also einen Speicherbereich, der die in der Klasse definierten Slots aufnehmen kann, zu erzeugen, muß die Funktion `make-instance` verwendet werden:

```
  (make-instance '3d-point-class)
```

12.2 Zugriff auf Instanzvariablen

Nach diesem Aufruf steht ein neues Objekt vom Typ `3d-point-class` zur Verfügung, dessen Slots mit den gewünschten Werten gefüllt werden können. Im folgenden werden die Vorgehensweisen bei Strukturen und Klassen skizziert:

[15] Die Bezeichnung *Slot* ist nur im Kontext von Lisp üblich. Der Rest der Welt redet hier eher von Instanz- oder Objektvariablen.

```
;;; Struktur 3d-point-struct erzeugen und danach x auf 1 setzen
(let ((p (make-3d-point-struct)))
  (setf (3d-point-struct-x) 1))

;;; Selbiges f\"ur die entsprechende Klasse:
(let ((p (make-instance '3d-point-struct)))
  (setf (slot-value p 'x) 1))
```

Wie man an diesem Beispiel sieht, existieren für das Erzeugen und Manipulieren eines Objekts nur grundlegende Funktionen. Die Definition einer Klasse erzeugt also tatsächlich nur die Klasse, nicht aber die von Strukturen bekannten Manipulationsfunktionen. Man kann Lisp aber leicht überreden, diese Funktionen nachzuliefern. Als Beispiel sei die folgende neue Definition der Klasse `3d-point-class` gegeben:

```
(defclass 3d-point-class ()
  ((x :accessor point-x)
   (y :accessor point-y)
   (z :accessor point-z)))
```

Diese Definition veranlaßt das Lispsystem, drei Funktionen `point-x`, `point-y` und `point-z` zu erzeugen, mit denen auf die zugeordneten Slotwerte eines Objekt zugegriffen werden kann. Somit kann man jetzt schreiben:

```
;; Objekt vom Typ 3d-point-class erzeugen und Slot x auf 1 setzen:
(let ((p (make-instance '3d-point-class)))
  (setf (point-x p) 1))
```

Wer es noch eleganter haben möchte, kann für den lesenden und schreibenden Zugriff auf Instanzvariablen je eigene Zugriffsfunktionen erzeugen lassen. Dadurch kann das `(setf <accessor> <wert>)` Konstrukt, das auch im letzten Beispiel verwendet wurde, entfallen. Mit der Klassendefinition

```
(defclass 3d-point-class ()
  ((x :reader get-x :writer set-x)
   (y :reader get-y :writer set-y)
   (z :reader get-z :writer set-z)))
```

läßt sich ein 3D-Punkt dann folgendermaßen erzeugen und initialisieren:

```
;; Objekt vom Typ 3d-point-class erzeugen und Slot x auf 1 setzen:
(let ((p (make-instance '3d-point-class)))
  (set-x 1 p))
```

Zu beachten ist hier, daß die Reihenfolge der Argumente gegenüber dem `setf`-Aufruf vertauscht ist. Würde Lisp nicht hin und wieder kleine Fallen stellen, wäre das Leben zu einfach...

12.3 Initialisierungsausdrücke

Wenn mit `make-instance` ein neues Objekt definiert wird, ist die Belegung der Instanzvariablen undefiniert. Man kann aber bei der Definition einer Klasse sehr einfach einen Standardwert für die einzelnen Objektvariablen festlegen. Dies geschieht mittels des Schlüsselarguments `:initform`. Für die Klasse `2d-punkt` wäre es zum Beispiel sinnvoll, jeden nicht weiter spezifizierten Punkt per Default im Koordinatenursprung zu plazieren. Die folgende Klassendefinition erledigt dies:

```
(defclass 2d-punkt ()
  (( x :accessor point-x :initform 0)
   ( y :accessor point-y :initform 0)))
```

Diese Klassendefinition sorgt dafür, daß ein Aufruf `(make-instance '2d-punkt)` den Punkt (0,0) liefert. Dies bedeutet aber unter anderem, daß jeder Punkt, der nicht im Koordinatenursprung liegt, zweimal initialisiert werden müßte. Um dies zu vermeiden, stellt CLOS die Möglichkeit zur

Verfügung, Initialisierungsargumente für den Konstruktoraufruf zu spezifizieren. Um einen Punkt (1,2) mit dem dem Aufruf `(make-instance '2d-punkt :x 1 :y 2)` direkt erzeugen zu können, ist einfach folgende Erweiterung der Klassendefinition anzugeben:

```
(defclass 2d-punkt ()
  (( x :accessor point-x :initform 0 :initarg :x)
   ( y :accessor point-y :initform 0 :initarg :y)))
```

Natürlich wäre CLOS kein Bestandteil von Common Lisp, wenn es nicht noch eine Menge weiterer Schlüsselargumente gäbe. So kann man mittels des Schlüssel `:type` den Typ einer Instanzvariablen einschränken, um den Compiler bei der Codeoptimierung zu unterstützen. Was der Compiler aber wirklich mit dieser Hilfe anfängt, ist ganz allein der jeweiligen Lisp-Implementation überlassen. Schließlich kann jeder Instanzvariablen ein eigener Dokumentationsstring, definiert durch den Schlüssel `:documentation`, mitgegeben werden.

12.4 Methoden

Bislang haben sich Klassen bzw. Objekte vor allem als umständlich handzuhabende Strukturen präsentiert. Sinnvoll nutzbar werden Objekte erst durch Methoden, die in CLOS als eine Verallgemeinerung des Funktionskonzepts von Lisp implementiert sind. CLOS unterscheidet sich durch seine Behandlung von Methoden am stärksten von anderen objektorientierten Sprachen, da sie nicht direkt mit einer Klasse verknüpft sind.

Im folgenden sollen einige einfache geometrische Objekte im zweidimensionalen Raum modelliert werden. Um das Beispiel kurz zu halten, werden nur Punkte, Kreise und Rechtecke betrachtet. Die entsprechenden Klassendefinitionen könnten zum Beispiel wie folgt aussehen:

```
;; Punkte zeichnen sich durch ihre x- und y-Koordinaten aus.
```

```
(defclass 2d-punkt ()
 ((x :accessor point-x)
  (y :accessor point-y)))

;; Ein Kreis wird durch seinen Mittelpunkt und Radius bestimmt.
(defclass kreis ()
  ((mitte :accessor kreis-mitte :type 2d-punkt)
   (radius :accessor kreis-radius)))

;; Ein Rechteck kann \"uber die Punkte der linken oberen und rechten
;; untern Ecke definiert werden.
(defclass rechteck ()
  ((linksoben :accessor rechteck-lo :type 2d-punkt)
   (rechtsunten :accessor rechteck-ru :type 2d-punkt)))
```

Das Beispiel zeigt unter anderem, daß Klassen (natürlich) auch andere Klassen enthalten können, da zum Beispiel der Mittelpunkt eines Kreises als Instanz durch einen `2d-punkt` repräsentiert wird. (Die Angabe von `:type` ist optional und dient hier vor allem als Dokumentation). Das Einbetten von Klassen wird als *Aggregation* bezeichnet und stellt in einigen Fällen eine bessere Alternative zur weiter unten vorgestellten Vererbung dar. Abgesehen von diesem Detail enthält das Beispiel aber nichts neues und sollte deshalb ohne weiteren Kommentar verständlich sein.

Nun existieren aber geometrische (und andere) Objekte in einem leeren Raum, in dem sie unveränderlich und ewig verharren. Vielmehr kann man mit geometrischen Objekten eine Reihe nützlicher Dinge anstellen. Mit Punkten können alle Vektoroperationen durchgeführt werden. Von Kreisen und Rechtecken kann der Flächeninhalt berechnet oder durch Skalierung verändert werden, und vieles mehr. Als Lispexperte [16] wird man diesem Umstand sicher sofort dadurch begegnen, daß man sich einen Satz von Funktionen definiert, die

[16] Gemeint ist hier ein Lispexperte der alten Schule, der die Existenz objektorientierter Softwareentwicklung bislang gekonnt ignoriert hat.

diese Operationen implementieren. Ein Ausschnitt aus einer solchen Funktionssammlung:

```
;; Zwei Punkte 'punkt1' und 'punkt2' addieren
(defun add (punkt1 punkt2)
  (let ((p (make-instance '2d-punkt)))
  (setf (point-x p) (+ (punkt-x punkt1) (punkt-x punkt2))
        (point-y p) (+ (punkt-x punkt1) (punkt-y punkt2)))
  p))

;; Punktkoordinaten 'p' um den Faktor 's' skalieren
(defun scale (p s)
  (setf (point-x p) (* (punkt-x p) s))
  (setf (point-y p) (* (punkt-y p) s)))

;; Fl\"ache eines Kreises 'kreis' berechnen
(defun flaeche (kreis)
  (* pi (expt (kreis-radius k) 2)))
```

Da es durchaus Sinn macht, auch die Fläche eines Rechtecks zu berechnen, wird eine zweite Funktion geschrieben, die die Flächenberechnung für Rechtecke erledigt. Da bereits eine Funktion namens Fläche existiert und zudem zu erwarten ist, daß man seiner Sammlung geometrischer Objekte weitere Objekte hinzufügen wird, deren Flächeninhalt man berechnen möchte, hält sich der Lisp-Experte an die gute alte Konvention, Funktionen durch den Typ, auf den sie angewendet werden können/sollen/... zu präfigieren. Das Ergebnis dieser Strategie verlangt viel Geduld beim Eintippen des Codes und sieht zudem eher verheerend aus:

```
(defun kreis-flaeche (ein-kreis) (...))
(defun rechteck-flaeche (ein-rechteck) (...))
(defun polygon-flaeche (ein-polygon) (...))
(defun dreieck-flaeche (ein-dreieck) (...))
```

Unter der Annahme, daß man eine (wirklich kleine) Sammlung von etwa 10 Klassen mit jeweils 10 verschiedenen Funktionen implementieren möchte, kommt man schnell auf 100 zu implementierende Funktionen [17]. Dazu kommt der Unsicherheitsfaktor, daß `kreis-flaeche` nur mit einem Kreis, nicht aber mit einem Rechteck funktioniert und deshalb sorgfältig auf die korrekte Parameterversorgung geachtet werden muß. Der (etwas ignorante) Lisp-Experte hat mit seiner Vorgehensweise also nicht nur einen Weg gefunden, besonders viel Code zu produzieren, der Code ist zusätzlich noch besonders fehleranfällig. Glückwunsch!

Das Problem der korrekten Parameterversorgung kann sehr einfach CLOS übertragen werden und nebenbei spart man auch noch etwas Tipparbeit. Das Zauberwort heißt `defmethod`, das zunächst als Variante von `defun` betrachtet werden kann. `defmethod` erlaubt es im Gegensatz zu `defun` die Angabe der Klasse, für die der Funktionsaufruf gültig ist. Dadurch kann man mehrere Objektfunktionen bzw. Methoden gleichen Namens definieren. Die Entscheidung, welche Variante aufgerufen wird, geschieht transparent durch das CLOS. Nach den diversen Codefetzen folgt nun ein vollständigeres Beispiel, das die meisten der bislang vorgestellten CLOS-Konstrukte verwendet:

```
;; Ein 2D-Punkt besteht einfach aus seinen X- und Y-Koordinaten.
(defclass 2d-punkt ()
  ((x :reader get-x :writer set-x :initform 0 :initarg :x)
   (y :reader get-y :writer set-y :initform 0 :initarg :y)))

;; Erzeugt einen neuen Punkt, dessen Koordinaten die Summe der
;; Koordinaten der beiden urpr\"unglichen Punkte sind.
(defmethod add ((self 2d-punkt) (p 2d-punkt))
  (make-instance '2d-punkt
   :x (+ (get-x p) (get-x p))
      :y (+ (get-y p) (get-y p))))
```

[17] Die Zahl 100 ist hier eher das Ergebnis einer Milchmädchenrechnung, die den Worst Case abdeckt. Näherungsweise könnte diese Zahl aber stimmen.

```
;; Subtraktion zweier Punkte.
(defmethod sub ((self 2d-punkt) (p 2d-punkt))
  (make-instance '2d-punkt
   :x (- (get-x self) (get-x p))
   :y (- (get-y self) (get-y p))))

;; Verschiebt einen Punkt um die angegebenen Offsets.
(defmethod move-by ((self 2d-punkt) xoff yoff)
  (progn (set-x  (+ (get-x p) xoff))
         (set-y  (+ (get-y p) yoff))
   self))


;; Ein Kreis wird \"uber seinen Mittelpunkt und Radius definiert.
(defclass kreis ()
   ((mitte :reader get-mitte :writer set-mitte
           :initform 0 :initarg :mitte)
     (radius :reader get-radius :writer set-radius
     :initform 0 :initarg :radius)))

  (defmethod flaeche ((self kreis))
    (* pi (expt (get-radius self) 2)))

  (defmethod scale ((self kreis) s)
    (set-radius (* (get-radius k) s))
    self)


;; Eine m\"ogliche Definition einer Rechteckklasse. Hier wird ein
;; Rechteck durch seine linke obere und rechte untere Ecke definiert.
(defclass rechteck ()
  ((linksoben :reader get-lo :writer set-lo
        :initform nil :initarg :lo)
```

```
  (rechtsunten :reader get-ru :writer set-ru
 :initform nil :initarg :ru)))
 (defmethod flaeche ((self rechteck))
  (let ((rect (sub (get-ru self) (get-lo self))))
      ; rect enth\"alt die Ausdehnung (H\"ohe, Breite) des Rechtecks
      (* (get-x rect) (get-y rect))))
```

Der rein äußerliche Unterschied zwischen `defmethod` und `defun` liegt einzig in der Möglichkeit, optional den Argumenttypen zu spezifizieren, indem als Parameter ein Paar `(<parametername> <parametertyp>)` angegeben wird. Beim Parametertypen muß es sich nicht unbedingt um eine selbstdefinierte Klasse handeln. Man kann mit Typbezeichnern wie unter anderem mit `NUMBER`, `FLOAT`, `INTEGER`, `RATIONAL` oder `COMPLEX` auf die internen Lisp-Typen referieren. Die grobe Regel zur Auswahl einer Methode lautet dabei, daß die speziellste zu Verfügung stehende Variante ausgewählt wird, wie das folgende Beispiel zeigen soll:

```
> (defmethod test-1 ((a-number number)) 'a-number)
> (defmethod test-1 ((a-float float)) 'a-float)
> (test-1 1)
A-NUMBER
> (test-1 7/5)
A-NUMBER
> (test-1 1.0)
A-FLOAT
```

Für die übrigen Details sollte man sich einen Lisp-Interpreter schnappen und ein wenig mit Methoden- und Klassendefinitionen herumexperimentieren. Nur bekommt man ein Gefühl für das Verhalten des verwendeten Interpreters. Wer die bis hier aufgeführten Beispiele ausprobiert, wird wahrscheinlich eine Menge neuer Meldungen und Ausgabeformate zu Gesicht bekommen [18].

[18] Wer eine aktuelle Clisp-Version benutzt, sollte einmal folgendes ausprobieren: zuerst `test-1` mit einem nichtnumerischen Argument aufrufen und danach eine Variante von `test-1` ohne eine Spezifikation des Parametertypen definieren.

Wenn der Parametertyp nicht spezifiziert wird, kann der jeweilige Parameter einen beliebigen Lisptyp übernehmen. Es kann so zum Beispiel leicht eine weitere Methode `test-1` definiert werden, die den Aufruf mit nichtnumerischen Argumenten abfängt.

Ein kleiner Tip: Man sollte darauf achten, daß Methoden, die keinen speziellen Rückgabewert haben, da ihre Aufgabe durch Nebeneffekte erfüllt wird [19], das Objekt zurückliefern, mit dem sie aufgerufen wurden. Dadurch kann man Methodenaufrufe schachteln und kürzeren, besser lesbaren Code schreiben.

Das Beispiel der geometrischen Objekte zeigt einige konzeptuelle Unterschiede zwischen den drei Klassen `2d-punkt`, `kreis` und `rechteck`. Während `kreis` und `rechteck` Punkte enthalten, und damit komplexere Klassen bilden, stellt ein Punkt gewissermaßen eine atomare Klasse dar. Zum anderen unterscheiden sich die Klassen durch ihre Methoden: von `kreis` und `rechteck` können Flächeninhalte berechnet werden, während Vektoroperationen auf Kreise und Rechtecke nicht sinnvoll anwendbar sind. Eine objektorientierte Modellierung einer Domäne sollte diese Zusammenhänge durch geeignete Mittel zum Ausdruck bringen. Eine Möglichkeit kann die in diesem Abschnitt bereits angesprochene Aggregation sein. Eine andere Möglichkeit wird im folgenden Abschnitt vorgestellt.

12.5 Vererbung

Überladene Methoden bekommen erst im Kontext der Klassenvererbung ihre eigentliche Bedeutung. Ein Teil des zugrundeliegenden Mechanismus ist mit der Beispielmethode `test-1` bereitsverwendet worden. Dort wurde eine allgemeine Version für `NUMBER` und eine spezielle Version für `FLOAT` definiert. Für die internen Lisptypen existiert, wie das Beispiel bereits andeutet, eine Typhierarchie, die Spezialisierungs- bzw. Verallgemeinerungsrelationen zwischen den einzelnen Typen ausdrückt. Derartige Hierarchien können auch für eigene Klassen definiert werden. Dabei kommt das zweite Argument von

[19] Methoden zur Ausgabe von Objekten sind ein klassisches Beispiel für diesen Fall.

`defclass` ins Spiel, das bislang stets als leere Liste angegeben wurde. Gibt man hier den Namen einer Klasse *super* an, legt man dadurch fest, daß die neue Klasse *sub* eine Spezialisierung der angegebenen Klasse *super* darstellt. Solange keine spezielleren Methoden definiert werden, können alle für *super* gültigen Methoden auch auf die neue Klasse *sub* angewendet werden.

Spezialisierung eine Klasse bedeutet auch, daß eine Subklasse alle Eigenschaften der Superklasse aufweist. Dies manifestiert sich in der objektorientierten Vererbung dadurch, daß eine Subklasse automatisch alle Instanzvariablen bzw. Slots der Superklasse übernimmt:

```
> (defclass test-1 () (x y))
#<STANDARD-CLASS TEST-1>
> (defclass test-2 (test-1) ())
#<STANDARD-CLASS TEST-2>
> (make-instance 'test-2)
#<TEST-2 #x20236C61>
> (setq obj (make-instance 'test-2))
#<TEST-2 #x20238549>
> (setf (slot-value obj 'x) 1)
1
> (setf (slot-value obj 'y) 2)
2
> (slot-value obj 'x)
1
> (slot-value obj 'y)
2
```

Da syntaktisch keine Möglichkeit besteht, Slot-Attribute für geerbte Instanzvariablen zu definieren, werden auch diese vererbt:

```
> (defclass test-1 () ((x :initform 0 :accessor tx)
                       (y :initform 0 :accessor ty)))
#<STANDARD-CLASS TEST-1>
```

```
> (defclass test-2 (test-1) ())
#<STANDARD-CLASS TEST-2>
> (setq obj (make-instance 'test-2))
#<TEST-2 #x202408FD>
> (tx obj)
0
> (ty obj)
0
```

Spätestens hier wird wohl klar, daß es sich bei der ‘Klassenhierarchie’ geometrischer Objekte um keine Hierarchie handelt. Wie oben bereits erwähnt wurde, ist es nicht vorteilhaft, `2d-punkt` als Schwesterklasse von `rechteck` und `kreis` zu behandeln. Somit bleiben nur noch zwei Klassen übrig, um etwas hierarchieartiges aufzubauen. Eine Möglichkeit besteht, die Gemeinsamkeiten dieser beiden Klassen durch Vererbung auszudrücken besteht darin, ein Rechteck sozusagen als eine ‘Erweiterung’ eines Kreises zu betrachten. Dazu muß die oben angegebene Definition eines Rechtecks ein wenig geändert werden:

```
(defclass kreis () (mitte breite))
(defclass rechteck (kreis) (hoehe))
```

Dieser Ansatz bring aber keinen besonderen Vorteil. Zum einen wirkt es unnatürlich, ein Rechteck als Spezialfall eines Kreises zu betrachten [20]. Zum anderen wird der Gewinn, bei den Klassendefinitionen durch die Tatsache geschmälert, daß sich wahrscheinlich die meisten Operationen, die für einen Kreis definiert wurden, nicht für ein Rechteck übernehmen lassen.

[20] Man könnte dieses Beispiel fortführen, indem man ein Quadrat als noch spezielleren Fall eines Kreises betrachtet. Dann würde man ein Quadrat von einem Kreis ableiten, ohne neue Instanzvariablen einführen zu müssen. Unter diesen Vorraussetzungen kann dann ein Rechteck als Spezialfall eines Quadrats betrachtet werden. Führt man dieses Design konsequent für eine größere Hierarchie fort, ist die Intuition wahrscheinlich schon über Bord gegangen, bevor man seinen Geniestreicht vollständig in den Rechner getippt hat.

Der wesentlichste Schwachpunkt dieser Lösung liegt in der grundsätzlichen Betrachtungsweise der Vererbung. Beim Design von Klassenhierarchien sollte stets die Vererbung von Schnittstellen im Vordergrund stehen, nicht aber die Vererbung von Implementationen. Das heißt, daß man sich beim Entwurf eine Klasse nicht, wie oben geschehen, darauf konzentrieren sollte, welche Bestandteile man von einer anderen Klasse übernehmen kann, sondern wie Aufgaben auf eine möglichst einheitliche Art und Weise unter den verschiedenen Klassen verteilt werden können. Es sollte also etwas wie eine abstrakte Klasse namens `2d-flaechen-objekt` existieren, die eine Menge von Methoden definiert, die von Unterklassen definiert werden. Diese Klasse wird dann möglichst anstelle konkreter Unterklassen wie `kreis` oder `rechteck` verwendet. Dies könnte dann als Lispcode folgendermaßen aussehen:

```
(defclass 2d-flaechen-objekt () ())
(defclass kreis (2d-flaechen-objekt) (mitte radius))
(defclass rechteck (2d-flaechen-objekt) (linksoben rechtsunten))

(defmethod flaeche ((self 2d-flaechen-objekt)) ...)
(defmethod skaliere ((self 2d-flaechen-objekt)) ...)
(defmethod verschiebe ((self 2d-flaechen-objekt)) ...)
```

Auch diese Lösung ist eher merkwürdig, da sich für die angegebenen Methoden keine sinnvolle Implementation angeben läßt. Selbst wenn man Teile der Operationen auf den abgeleiteten Objekten in die Basisklasse verlagern könnte [21], besteht die Gefahr, daß einfach unbemerkt eine Methode von `2d-flaechen-objekt` verwendet, weil zum Beispiel die Methode für die Berechnung einer Kreisfläche nach einem Cut-und-Paste-Experiment aus dem Quelltext verschwunden ist. Den Lispinterpreter stört das Verschwinden der Methode nicht weiter, da er ja auf die entsprechende Methode der Basisklasse zurückgreifen kann. Die Schuld für das Scheitern dieses Ansatzes liegt nicht beim Programmierer, der beschließt, ein Protokoll für Objekte festzulegen, deren Fläche man berechnen kann. Die Schuld liegt vielmehr bei CLOS, das

[21] Falls hier jemandem etwas sinnvolles einfällt, soll sie/er sich bitte sofort bei mir melden!

das hier benötigte Konzept einer abstrakten Klasse oder zumindest einer abstrakten Methode nicht zur Verfügung stellt. Eine sehr lispige Art, dieses Problem zu umgehen bietet ein Makro der folgenden Art:

```
(defmacro abstractmethod (name args)
  `(defmethod ,name ,args
     (error "Eine abstrakte Methode wurde aufgerufen")))
```

Danach kann man dann eine abstrakte Methoden folgendermaßen simulieren:

```
(defclass 2d-flaechen-objekt () ())
(defclass kreis (2d-flaechen-objekt) (mitte radius))
(defclass rechteck (2d-flaechen-objekt) (linksoben rechtsunten))

(abstractmethod flaeche ((self 2d-flaechen-objekt)) ...)
(abstractmethod skaliere ((self 2d-flaechen-objekt)) ...)
(abstractmethod verschiebe ((self 2d-flaechen-objekt)) ...)
```

Nachteil dieser Lösung ist, daß Fehler, also versehentliche Aufrufe abstrakter Methoden, sich erst zur Laufzeit bemerkbar machen.

12.6 Zugriffsbeschränkungen

Im Normalfall bieten objektorientierte Sprachen für die Elemente eines Objekts verschiedene Stufen der Zugriffsbeschränkung, die in den üblichsten OOP-Sprachen durch Qualifizierer wie `private`, `protected`, `public`, etc. ausgedrück werden. Zudem kann in einigen Sprachen darüberhinaus auch der Zugriff auf Instanzvariablen durch spezielle Methoden kontrolliert werden (z.B `property` in Delphi, `attribute` in CORBA-IDL). Wie lassen sich diese Konzepte in CLOS umsetzen? Die kurze Antwort lautet "überhaupt nicht". Da kurze Antworten bisweilen unbefriedigend sind, kommt hier die längere Version:

Im Abschnitt über den Zugriff auf Instanzvariablen wurden die Zugriffsattribute `:reader` und `:writer` eingeführt. Um Zugriffsbeschränkungen zu simulieren, gibt man einfach nur eins der Attribute an. Hält man sich an die Konvention, nur die in der Methodendefinition erzeugten Slot-Accessors zu verwenden, kann man so Zugriffsbeschränkungen simulieren. CLOS unterstützt das Information Hiding aber nicht in konsequent, da man mittels `slot-value` immer auf Instanzvariablen zugreifen kann. Die Skopuseinschränkung durch Qualifizierer wie `protected`, `private` oder `public` ist in CLOS überhaupt nicht vorgesehen und eine Simulation dieser Qualifizierer ist nicht ohne weiteres möglich [22].

12.7 Weitere Features von CLOS

CLOS bietet eine Menge weiterer Features, von denen hier aber nur einige der Vollständigkeit halber kurz erwähnt werden:

- `:before`-, `:after`- und `:around`- Methoden: Diese Methoden tauchen hier an erster Stelle auf, weil sie meiner Meinung das obskurste sind, was CLOS zu bieten hat. Definiert man eine Methode mit den oben vorgestellten Mitteln, so definiert man im CLOS-Jargon "nur" die *Primary Method*. Definiert man weitere Methoden gleichen Namens und mit gleichen Parametertypen und läßt dem Methodennamen eins der Schlüsselwörter `:before`, `:after` oder `:around` folgen, dann werden die jeweiligen Methoden, vor, nach oder vor und nach der Primary Method aufgerufen.

- CLOS basiert auf *MOP*, dem Meta Object Protocol. MOP liefert die Basisfunktionalität zum Definieren von Objekten und Methoden. Funktionen des MOP können, analog zum Beispiel zur Introspektion in Java, Objective-C oder Smalltalk, dazu genutzt werden, die Eigenschaften von Objekten (Vererbungsbeziehungen, Slotnamen etc.) zu erfragen.

[22] Ich habe allerdings den Verdacht, daß Skopusqualifizieren durch Manipulationen des MOP (s.u.) simuliert werden können

Durch Eingriffe in diese Ebene des Objektsystems kann zudem dessen Verhalten an das anderer Objektsysteme angepaßt werden.

- *Mehrfachvererbung* ist in CLOS möglich. Dazu muß bei der Definition einer Klasse einfach eine Liste von Elternklassen angegeben werden. Da Mehrfachvererbung aber eine neue Klasse von Problemen [23] einführt, wird hier nicht weiter darauf eingegangen. Mehrfachvererbung kann zudem in den meisten Fällen durch Alternativen wie Aggregation oder Delegation umgangen werden. Da CLOS keine Möglichkeiten zur reinen Spezifikation von Schnittstellen zur Verfügung stellt (`interface` in Java oder Delphi, `@protocol` in Objective-C), können in CLOS durchaus Fälle auftreten, in denen Mehrfachvererbung die sinnvollste Alternative zur Modellierung von Klassenrelationen darstellt.

13 Pattern Matching und Unifikation

13.1 Pattern Matching

Pattern Matching (Mustervergleich) ist der Vergleich symbolischer Ausdrücke. Pattern Matching findet an vielen Stellen in der Programmierung seine Anwendung, so z.B. bei der berprüfung von Eingaben eines Benutzers auf seine Korrektheit usw. Einfache Mustervergleiche lassen sich in LISP relativ leicht implementieren, etwa in Form einer Funktion `match`, die folgendes leistet:

```
>  (match '(a b c) '(a b c))
T
```

Könnte man mit der Prozedur `match` nur gleiche Muster erkennen, so wäre sie ziemlich nutzlos. Was man wirklich möchte ist folgendes:

[23] Probleme treten typischerweise dann auf, wenn für zwei Klassen, die gemeinsam eine Subklasse beerben Methoden oder Instanzvariablen gleichen Namens definiert sind.

```
>  (match '(a ,x c) '(a (e r q) c))
T
```

Betrachtet man `,x` als Variable, deren Inhalt frei gewählt werden kann, so läßt sich das Pattern `(a ,x c)` mit dem Wert `(a (e r q) c))` matchen, wenn `,x` den Wert `(e r q)` erhält. Aus der Ausgabe von `t` allein ist noch nicht klar, welchen Wert `,x` denn nun erhalten muß, damit das Pattern und der Wert übereinstimmen. Deshalb sollte nun `,x` den entsprechenden Wert besitzen.

```
>  x
(E R Q)
```

Folgende Definition der Prozedur `match1` leistet das Gewünschte:

```
(defun match1 (*pattern* *wert*)
    (cond ((consp *pattern*)
            (cond ((equal (car *pattern*) 'comma)
                        (set (cadr *pattern*) *wert*))
                  (T (and
                        (consp *wert*)
                        (match1 (car *pattern*)
                        (car *wert*))
                     (match1 (cdr *pattern*)
                             (cdr *wert*))))))
          (T (equal *pattern* *wert*))))
```

Das hier verwendete `set` unterscheidet sich zum einen von `setq` dadurch, daß es sein erstes Argument auswertet und dann zuweist.

```
>  (setq x 'y)               Ausgabe: Y
>  (setq x 1)                Ausgabe: 1
>  x                         Ausgabe: 1
```

```
>  y                          Ausgabe: nil

>  (setq x 'y)                Ausgabe: Y
>  (set x 1)                  Ausgabe: 1
>  x                          Ausgabe: Y
>  y                          Ausgabe: 1
```

Der größte Unterschied besteht allerdings darin, daß `setq` die lexikalischen Variablen belegt und `set` die dynamischen Variablen, die es so (in COMMON LISP) nicht gibt. Daher belegt `set` hier nur globale Variablen, was diese Definition von `match` unschön macht.

Eine Definition von `match`, die dieses Problem der globalen (besser nicht dynamischen) Variablen löst, ist die folgende:

```
(setq match2 '(lambda (*pattern* *wert*)
                (cond ((consp *pattern*)
                         (cond ((equal (car *pattern*) 'comma)
                                  (set (cadr *pattern*) *wert*))
                               (T (and
                                    (consp *wert*)
                                    (match2 (car *pattern*)
                                    (car *wert*))
                                  (match2 (cdr *pattern*)
                                          (cdr *wert*))))))
                      (T (equal *pattern* *wert*)))))
```

Den Unterschied zwischen `match1` und `match2` macht man sich am besten an einem Beispiel klar.

```
>  (defun test1 ()
       (let (x y)
            (match1 '(a ,x c ,y) '(a b c d))
            (print x)
```

```
            (print y)))
>  (defun test2 ()
       (let (w z)
            (match2 '(a ,w c ,z) '(a b c d))
            (print w)
            (print z)))

>  (test1)
nil
nil
nil
>  (test2)
B
D
D
```

Daran sieht man, daß `match1` die lexikalischen Variablen und `match2` die dynamischen Variablen bindet.

14 Prologinterpreter

Ein gutes Beispiel, um sich die Verwaltung von Variablen mittels LISP anzuschauen, ist der hier beschriebene kleine Prolog-Interpreter. Er wurde ursprünglich von Ken Kahn in MacLisp geschrieben und für Common-Lisp von David Betz modifiziert. Dieser Interpreter wurde nicht dafür geschrieben, mit ihm wirklich Prolog zu implementieren, sondern rein für Test- und Lehrzwecke. Der Interpreter kommt ohne das Produzieren von Nebeneffekten aus, was allerdings dazu führt, daß er sehr langsam ist. Außerdem stellt er nur die grundlegenden Sprachelemente von Prolog zur Verfügung. Der Prolog-Interpreter arbeitet nach dem Prinzip „downward success continuation“.

```
(defun prolog (database &aux goal)
```

```
 (do ()
  ((not (progn (princ "Query?") (setq goal (read)))))
  (prove
   (list (rename-variables goal '(0)))
   '((bottom-of-environment))
   database
   1
)))
```

`prove` - Prüft die Konjunktion der Goals in der akuellen Umgebung

```
(defun prove (list-of-goals environment database level)
  (cond ((null list-of-goals) ;; Erfolg, wenn keine Goals mehr
         (print-bindings environment environment)
         (not (y-or-n-p "More?")))
        (t
         (try-each database database
            (cdr list-of-goals)
            (car list-of-goals)
            environment
            level))))


(defun try-each (database-left database goals-left goal
                 environment level
                 &aux assertion new-enviroment)
  (cond ((null database-left) nil) ;; Anfrage fehlgeschlagen,
                       ;; da keine Klausel mehr vorhanden ist
        (t (setq assertion     ;; naechste Klausel
             (rename-variables (car database-left)
               (list level)))
           (setq new-environment
             (unify goal (car assertion) environment))
```

```
           (cond ((null new-environment) ;; failed to unify
                    (try-each (cdr database-left) database
                      goals-left goal
                      environment level))
                 ((prove (append (cdr assertion) goals-left)
                     new-environment
                     database
                     (+ 1 level)))
                 (t (try-each (cdr database-left) database
                      goals-left goal
                      environment level))))))

(defun unify (x y environment &aux new-environment)
    (setq x (value x environment))
    (setq y (value y environment))
    (cond ((variable-p x) (cons (list x y) environment))
          ((variable-p y) (cons (list y x) environment))
          ((or (atom x) (atom y))
             (cond ((equal x y) environment)
                   (t nil)))
          (t (setq new-environment
               (unify (car x) (car y) environment))
             (cond (new-environment
               (unify (cdr x) (cdr y) new-environment))
                   (t nil)))))
```

Ist `x` oder `y` eine Variable, so kann man sie einfach unifizieren, indem die Variable als Wert die andere Struktur erhält. Ist eines der beiden ein Atom, so kann man sie nur unifizieren, wenn sie gleich sind. Handelt es sich um komplexere Strukturen, so wird versucht, die `car`s zu unifizieren. Klappt dieses, so wird `unify` auf den `cdr` mit der neuen Umgebung angewendet.

```
(defun value (x environment &aux binding)
```

```
    (cond ((variable-p x)
              (setq binding (assoc x environment :test #'equal))
              (cond ((null binding) x)
                    (t (value (cadr binding) environment))))
          (t x)))
```

Wenn `x` eine Variable ist, wird seine Bindung geholt. Ist die Bindung nil, dann ist `x` frei und wird so ausgegeben. Anderenfalls wird der Wert geholt und ausgegeben. '`:test #'equal`' bedeutet, daß beim Suchen nach dem Schlüssel `equal` als Vergleichsfunktion benutzt wird und nicht `eql`, die defaultmäßig benutzt wird.

```
(defun variable-p (x)
   (and x (listp x) (eq (car x) '?)))
```

`x` ist eine Variable, wenn es eine Liste ist und mit einem `?` beginnt.

```
(defun rename-variables (term list-of-level)
    (cond ((variable-p term) (append term list-of-level))
          ((atom term) term)
          (t (cons (rename-variables (car term) list-of-level)
             (rename-variables (cdr term) list-of-level)))))

(defun print-bindings (environment-left environment)
    (cond ((cdr environment-left)
            (cond ((= 0 (nth 2 (caar environment-left)))
                     (prin1 (cadr (caar environment-left)))
                     (princ " = ")
                     (print (value (caar environment-left)
                            environment))))
             (print-bindings (cdr environment-left) environment))))
```

Die Hauptarbeit des Prolog-Interpreters wird in den drei Funktionen `prove`, `try-each` und `unify` geleistet. Die anderen Funktionen sind Hilfsfunktionen.

Die Funktionen `prove` und `try-each` rufen sich wechselseitig auf, wobei sie die Funktion `unify` benutzen, die wie der Name schon ausdrückt, dazu da ist, Ausdrücke zu unifizieren.

```
;; ein Datenbasisbeispiel:
(setq db '((((father madelyn ernest))
           ((mother madelyn virginia))
           ((father david arnold))
           ((mother david pauline))
           ((father rachel david))
           ((mother rachel madelyn))
           ((grandparent (? grandparent) (? grandchild))
                  (parent (? grandparent) (? parent))
                  (parent (? parent) (? grandchild)))
           ((parent (? parent) (? child))
                  (mother (? parent) (? child)))
           ((parent (? parent) (? child))
                  (father (? parent) (? child))))


;; the following are utilities
(defun y-or-n-p (prompt)
 (princ prompt)
 (eq (read) 'y)
)

(prolog db)
```

Ruft den Prolog-Interpreter mit der angegebenen Datenbasis auf.

Desweiteren werden in `prolog` noch die Variablen umbenannt und die Funktion `prove` aufgerufen.

Diese erhält als erstes Argument die Liste der umbenannten Goals, als zweites die aktuelle Umgebung (environment), die zu Beginn nur aus der Endekennung `bottom-of-environment` besteht, als drittes die aktuelle Datenbasis und als viertes und letztes Argument den Level, welcher dazu dient, die Variablen in eindeutiger Weise umzubenennen.

`prove` testet zuerst, ob die Liste der Goals leer ist. Sollte das der Fall sein, so war die Anfrage erfolgreich und es werden die Bindungen der Variablen ausgegeben, was die Funktion `print-bindings` erledigt. Sollten noch Goals übrig sein, so wird `try-each` aufgerufen.

Das erste Argument `database-left` besteht aus den Klauseln, die noch übrig bleiben (versucht werden müssen), um das aktuelle Goal zu beweisen. `database` ist die Datenbasis mit der die Goals bewiesen werden müssen. `goals-left` enthält die Goals, die noch übrigbleiben, wenn `goal` bewiesen worden ist. `environment` enthält die aktuelle Umgebung (Belegung der Variablen).

Zuerst wird in `try-each` getestet, ob noch Klauseln vorhanden sind, um die Anfrage beweisen zu können. Ist dies nicht der Fall, so ist die Anfrage nicht zu beweisen `(fail)`. Anderenfalls erhält `assertion` als Wert die nächste Klausel, deren Variablen zuvor umbenannt werden, und die neue Umgebung `(new-environment)` entsteht aus der Unifikation des zu beweisenden Goals mit dem Kopf der Klausel in der alten Umgebung.

In dem folgenden `cond` wird zuerst getestet, ob die Unifikation nicht geklappt hat, d.h. `new-environment` nil ist. In diesem Fall ruft sich `try-each` mit der um eine Klausel kleineren Datenbasis auf und das Spiel beginnt von neuem. Es wird also die nächste Klausel getestet. Hat dagegen die Unifikation geklappt, so muß geprüft (`prove` ...) werden, ob die Bedingungen (Subgoals) der Klausel bewiesen werden können.

Sollte dieses nicht gelingen, so muß wie im Falle des Scheiterns der Unifikation auch, die nächste Klausel versucht werden.

Was das für Prolog ja sehr wichtige Backtracking angeht, so bekommt man es bei dieser Art Interpeter quasi geschenkt. Denn man nutzt einfach die Rekursion der Lisp-Funktionen aus. Dadurch, daß in `try-each` immer `new-environment` benutzt wird, ist bei einem Rücksetzen in der nächst höheren Rekursionsstufe immer noch das „alte“ `environment` vorhanden. Man braucht sich also nicht darum zu kümmern, beim Rekursionsabstieg (Beweisversuch eines Goals) gemachte Bindungen wieder rückgängig zu machen.

15 Ausschnitte aus dem *Jargon-File*

Die nachfolgenden Zitate stammen aus:
'The on-line hacker Jargon File, version 3.2.0, 21 MAR 1995'

LISP: n. [from 'LISt Processing language', but mythically from 'Lots of Irritating Superfluous Parentheses'] AI's mother tongue, a language based on the ideas of (a) variable-length lists and trees as fundamental data types, and (b) the interpretation of code as data and vice-versa. Invented by John McCarthy at MIT in the late 1950s, it is actually older than any other High-Level-Language still in use except FORTRAN. Accordingly, it has undergone considerable adaptive radiation over the years; modern variants are quite different in detail from the original LISP 1.5. The dominant HLL among hackers until the early 1980s, LISP now shares the throne with C.

All LISP functions and programs are expressions that return values; this, together with the high memory utilization of LISPs, gave rise to Alan Perlis's famous quip (itself a take on an Oscar Wilde quote) that 'LISP programmers know the value of everything and the cost of nothing'.

One significant application for LISP has been as a proof by example that most newer languages, such as COBOL and Ada, are full of unnecessary crocks. When the Right Thing has already been done once, there is no justification for bogosity in newer languages.

The '-P' convention: Turning a word into a question by appending the syllable 'P'; from the LISP convention of appending the letter 'P' to denote a predicate (a boolean-valued function). The question should expect a yes/no answer, though it needn't.

At dinnertime:
Q: 'Foodp?'

A: ‘Yeah, I’m pretty hungry.’ or ‘T!’

At any time:
Q: ‘State-of-the-world-P?’
A: (Straight) ‘I’m about to go home.’
A: (Humorous) ‘Yes, the world has a state.’

[One of the best of these is a Gosperism. Once, when we were at a Chinese restaurant, Bill Gosper wanted to know whether someone would like to share with him a two-person-sized bowl of soup. His inquiry was: ‘Split-p soup?’]

cdr: /ku’dr/ or /kuh’dr/ vt. [from LISP] To skip past the first item from a list of things (generalized from the LISP operation on binary tree structures, which returns a list consisting of all but the first element of its argument). In the form ‘cdr down’, to trace down a list of elements: ‘Shall we cdr down the agenda?’ Usage: silly.

Historical note: The instruction format of the IBM 7090 that hosted the original LISP implementation featured two 15-bit fields called the ‘address’ and ‘decrement’ parts. The term ‘cdr’ was originally ‘Contents of Decrement part of Register’. Similarly, ‘car’ stood for ‘Contents of Address part of Register’.

The cdr and car operations have since become bases for formation of compound metaphors in non-LISP contexts. GLS recalls, for example, a programming project in which strings were represented as linked lists; the get-character and skip-character operations were of course called CHAR and CHDR.

Real Programmer: n. [indirectly, from the book ‘Real Men Don’t Eat Quiche’] A particular sub-variety of hacker: one possessed of a flippant attitude toward complexity that is arrogant even when justified by experience. The archetypal ‘Real Programmer’ likes to program on the bare metal and is very good at same, remembers the binary opcodes for every machine he has ever programmed, thinks that HLLs are sissy, and uses a debugger to edit his code because full-screen editors are for wimps. Real Programmers aren't satisfied with code that hasn't been bummed into a state of tenseness just short of rupture. Real Programmers never use comments or write documentation: 'If it was hard to write', says the Real Programmer, 'it should be hard to understand.' Real Programmers can make machines do things that were never in their spec sheets; in fact, they are seldom really happy unless doing so. A Real Programmer's code can awe with its fiendish brilliance, even as its crockishness appalls. Real Programmers live on junk food and coffee, hang line-printer art on their walls, and terrify the crap out of other programmers – because someday, somebody else might have to try to understand their code in order to change it. Their successors generally consider it a Good Thing that there aren't many Real Programmers around any more. For a famous (and somewhat more positive) portrait of a Real Programmer, see 'The Story of Mel, a Real Programmer' in Appendix A. The term itself was popularized by a 1983 Datamation article 'Real Programmers Don't Use Pascal' by Ed Post, still circulating on Usenet and Internet in on-line form.

Literatur

[1] Anderson, Carbett, Reiser: *Essential Lisp*, Addison-Wesley, 1987

[2] D. Bentz (1984): XLISP Version 1.7, PD-Programm.

[3] D. Bentz (1987): XLISP Version 2.0, PD-Programm.

[4] Brooks, R.: *LISP: Programmieren in Common Lisp*, München 1987

[5] Church, A. (1932). A set of postulates for the foundation of logic. *Annals of mathematics*, 2, (33), S.346-366

[6] Church, A. (1933). A set of postulates for the foundation of logic. *Annals of mathematics*, 2, (34), S.839-864

[7] B. Enders, H. Gust (1992): *G_ED, G_WINDOW Benutzerhandbuch*, Software-Dokumentation, FB Sprach- und Literaturwissenschaft, Universität Osnabrück, 1992

[8] Gazdar, G.; Mellish, C.: *Natural Language Processing in Lisp*, Addison-Wesley, 1989

[9] H. Gust (1992): *G_LOG Benutzerhandbuch*, Software-Dokumentation, FB Sprach- und Literaturwissenschaft, Universität Osnabrück, 1992

[10] H. Gust, A. Tischler, T.Thelen (1996): *Einführung in Lisp mit G_LISP*, Software-Dokumentation, FB Sprach- und Literaturwissenschaft, Universität Osnabrück, 1996.

[11] Hekmatpour, S.: *Introduction to LISP and symbol manipulation*, Cambridge University Press 1988

[12] Kantrowitz, M. (Hrsg.): *comp.lang.lisp Frequently Asked Questions*, monatlich in den Usenet-Newsgruppen comp.lang.lisp, comp.answers

[13] Steel, G.L.jr.: , *COMMON LISP*, Digital Press

[14] Steel, G.L.jr.; Gabriel, R.P.: *The evolution of lisp*, in: The second ACM SIGPLAN History of programming languages conference, SIGPLAN Notices 28, 3 (March 1993), p. 231-270.

[15] Winston, P.H.; Horn, B.K.: LISP. Addison-Wesley

Index

λ-Kalkül, 6, 70
' (Quote), 11
, (Comma), 23, 74
,@ (Comma-At), 24, 74
` (Backquote), 23, 74
=, <u>42</u>
&aux, 30
&key, 30
&optional, 28
&rest, 29
and, 20
append, 23, 48
apply, 41
aref, 68
assoc, 59
atom, 33
backquote, 23
car, 12
cdr, 12
comma-at, 24
comma, 23
cond, 16
consp, 96
cons, 22
dappend, 54
defmacro, 73
defun, 14, 70
delete, 49
dlist, 53
do*, 37
dolist, 37
dotimes, 37
do, 36
eql, 32, <u>42</u>
equal, 20, 35, <u>42</u>
eq, <u>42</u>
eval, 60
flatc, 77
flatsize, 77
get, 63
go, 38
if, 17, 74
lambda, 70, 72
lastlist, 54
last, 25
length, 24
let*, 33
let, 33
listp, 33
list, 23
make-array, 68
make-pathname, 78
mapcan, 41
mapcar, 39
mapc, 40
maplist, 40
mapl, 40
match, 95
nconc, 48
nil, 16, 46

`null`, 18
`open`, 77
`or`, 20
`prin1`, 38, 75
`princ`, 38, 75
`print`, 18, 38, 75
`prog`, 38
`putprop`, 64
`quote`, 11
`read-char`, 76
`remprop`, 65
`return`, 38
`reverse`, 24
`rplaca`, 47
`rplacd`, 47
`setf`, 50, 64
`setq`, 13
`set`, 96
`subst`, 25
`symbol-plist`, 64
`terpri`, 75
`t`, 16
`write-char`, 76
`zerop`, 37

Aggregation, 84
Atom, 8
Ausgabe, 38

Bindung
- dynamische, 69
- lexikalische, 69

Common Lisp, 5
cons-Zelle, 22, 46

Differenzlisten, 51
- letztes Element (lastlist), 54
- umwandeln (dlist), 53
- verketten (dappend), 54

Fehler durch Seiteneffekte, 56
funktional, 15
Funktionen, 15
- Anwendung auf Listen, 39
- Anwendung einer Liste als Argumente, 41
- aufrufen, 10
- definieren, 14
- Definition, 69

Gleicheitstests, 41

Induktionsbeweis, 56

Jargon-File, 5

Klötzchenwelt, 66

Lambdaausdruck, 73
Lambdalisten, 28
- Syntax, 31

lexical closure, 72
Listen, 9, 11, 21
- Assoziations-, 58
- destruktive Prozeduren, 46
- Differenzlisten, 51
- Eigenschafts-, 63
- Lambda-, 28

Listen, Lambda, 31

Makros, 73

Parameter
  Hilfs-, 30
  optional, 28
  restliche, 29
  Schlüsselworte, 30
  Vergleichs-, 32
Pattern-Matching, 95
place form, 50
Präfixnotation, 11
Prozeduren, 15

read-eval-loop, 6
referentielle Transparenz, 55

Schleifen
  do, 36
  dolist, 37
  dotimes, 37
  prog, 38
Scoperegeln, 69
Seiteneffekte, 15, 55
Stream, 77

Variablen, 13
  globale, 13
  lokale, 30, 33
Variablenbindung, 72
Vergleichsoperatoren, 35, <u>41</u>
Vergleichsparameter, 32
Verifikation, 56

Zugriffsform, 50
Zugriffsschlüssel, 58